UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Escola d'Enginyeria de Barcelona Est

TRABAJO DE FIN DE GRADO

Grado en Ingeniería Electrónica Industrial y Automática

SISTEMA DE VISIÓN POR COMPUTADOR DE BAJO COSTE PARA ROBOTS UR

Memoria y Anexos

Autor: Joan Tur Ruiz
Director: Sebastian Tornil Sin

Convocatòria: Mayo 2023

Resum

En aquest projecte es tracta el disseny d'un sistema de visió per computador per robots de Universal Robots basat en Raspberry Pi amb l'objectiu de tenir un cost reduït. El motiu darrere d'aquest baix cost és que el sistema sigui senzill de replicar en centres educatius.

En primera instància es fa una introducció a la plataforma Raspberry Pi i als robots col·laboratius de UR. Partint d'aquesta base, s'exposa el desenvolupament i programació en Python del sistema de visió, tractant els següents apartats: calibració òptica i geomètrica de la càmera, processat de la imatge i detecció d'objectes amb la seva posició i orientació. De la mateixa manera es detalla el desenvolupament del sistema de comunicacions. També s'explica el procés de disseny i fabricació del sistema de fixació al robot, amb els plànols i models CAD generats.

Un cop muntat el sistema, es comprova la seva funcionalitat muntant-lo sobre un robot UR3e i implementant una aplicació d'exemple.

Resumen

En este proyecto se trata el diseño de un sistema de visión por computador para robots de Universal Robots basado en Raspberry Pi con el objetivo de que tenga un coste reducido. El motivo detrás del bajo coste es que el sistema sea sencillo de replicar en centros educativos.

En primera instancia se hace una introducción a la plataforma Raspberry Pi y a los robots colaborativos de UR. Partiendo de esta base, se expone el desarrollo y programación en Python del sistema de visión, tratando los siguientes apartados: calibración óptica y geométrica de la cámara, procesado de la imagen y detección de objetos con su posición y orientación. Del mismo modo se detalla el desarrollo del sistema de comunicaciones. También se explica el proceso de diseño y fabricación del sistema de sujeción al robot, con los planos y modelos CAD generados.

Una vez ensamblado el sistema, se comprueba su funcionalidad montándose sobre un robot UR3e e implementando una aplicación de ejemplo.

Abstract

This project involves the design of a computer vision system for Universal Robots-based robots using Raspberry Pi, with the goal of having a reduced cost. The reason behind this low cost is to make the system easy to replicate for educational centers.

Initially, an introduction is made to the Raspberry Pi platform and UR collaborative robots. Based on this foundation, the development and programming of the vision system in Python is explained, covering the following sections: optical and geometric calibration of the camera, image processing and object detection with its position and orientation. Similarly the development of the communication system is also detailed. The design and manufacturing process of the robot fastening system is also explained, along with the generated CAD plans and models.

Once the system is assembled, its functionality is verified by mounting it on a UR3e robot and implementing a sample application.

Glosario

Abreviaturas:

TFG: Trabajo de Final de Grado
UR: Universal Robots
PC: “Personal Computer”; ordenador personal
EEBE: Escola d'Enginyeria Barcelona Est
RAM: “Random Access Memory”; memoria de acceso aleatorio
HDMI: “High-Definition Multimedia Interface”; interfaz multimedia de alta definición
USB: “Universal Serial Bus”; bus universal en serie
GPIO: “General Purpose Input/Output”; entrada/salida de propósito general
CSI: “Camera Serial Interface”; interfaz serie para cámaras
DSI: “Display Serial Interface”; interfaz serie para displays
VNC: “Virtual Network Computing”; Computación Virtual en Red
TCP: “Tool Center Point”; punto central de la herramienta
CAD: “Computer Assisted Design”; diseño asistido por ordenador
RGB: “Red, Green and Blue”; rojo, verde y azul
HSV: “Hue, Saturation and Value”; tonalidad, saturación y brillo
RTDE: “Real Time Data Exchange”; intercambio de datos en tiempo real
Mw2c: Matriz homogénea que transforma un punto expresado en el sistema de referencia del plano de trabajo (“work plane”) al sistema de referencia de la cámara.
Rw2c: Vector de rotación extraído de la matriz homogénea Mw2C.
Tw2c: Vector de traslación extraído de la matriz homogénea Mw2c.
Mw2b: Matriz homogénea que transforma un punto expresado en el sistema de referencia del plano de trabajo (“work plane”) al sistema de referencia de la base del robot.
Rw2b: Vector de rotación extraído de la matriz homogénea Mw2b.
Tw2b: Vector de traslación extraído de la matriz homogénea Mw2b.
Mo2b: Matriz homogénea que transforma un punto expresado en el sistema de referencia del objeto al sistema de referencia de la base del robot.
Ro2b: Vector de rotación extraído de la matriz homogénea Mo2b.
To2b: Vector de traslación extraído de la matriz homogénea Mo2b.
Mb2g: Matriz homogénea que transforma un punto expresado en el sistema de referencia de la base del robot al sistema de referencia de la pinza o “gripper”.
Rb2g: Vector de rotación extraído de la matriz homogénea Mb2g.
Tb2g: Vector de traslación extraído de la matriz homogénea Mb2g.
Mg2o: Matriz homogénea que transforma un punto expresado en el sistema de referencia de la pinza o “gripper” al sistema de referencia del objeto.
Rg2o: Vector de rotación extraído de la matriz homogénea Mg2o.
Tg2o: Vector de traslación extraído de la matriz homogénea Mg2o.

Índice

1. Introducción

La importancia tanto de la robótica como de la visión por computador en la industria actual da lugar a una necesidad de técnicos formados en estos campos. Por ello es esencial que los centros educativos dispongan de las herramientas adecuadas para formar en estas materias. Por la parte de robótica la herramienta idónea es un robot colaborativo (por ejemplo el UR3e de Universal Robots), ya que evita la instalación de costosas y voluminosas medidas de seguridad como jaulas y escáneres de proximidad. En cuanto a la visión por computador la solución más versátil es una cámara con disposición *eye-on-hand*, ya que permite configurar con facilidad múltiples posiciones de captura. No obstante, las opciones en el mercado de este tipo de cámaras son de precio demasiado elevado para los presupuestos de la gran mayoría de instituciones educativas.

Dado que en el momento de plantear este TFG me hallaba realizando prácticas extracurriculares en Universal Robots Spain SL, Carlos Pérez (Responsable de Educación e Investigación en Universal Robots) sugirió el diseño de un sistema de visión *eye-on-hand* de bajo coste, motivando así este proyecto. Con este trabajo se pretende incentivar a los centros educativos a recrear el sistema en sus aulas, democratizando así el acceso a cámaras *eye-on-hand*.

1.1. Objetivos del trabajo

El objetivo de este trabajo es el diseño e implementación de un sistema de visión de bajo coste para robots colaborativos UR.

Para lograr el objetivo de bajo coste la computación habrá de basarse en la plataforma Raspberry Pi (Raspberry Pi 3 o 4). Estos ordenadores monoplaca de coste reducido proporcionan la potencia computacional necesaria para procesar las imágenes. Además dan soporte a sus propios módulos de cámara, facilitando su uso. Adicionalmente, su popularidad hace que muchos centros (como la EEBE) ya dispongan de ellos en sus laboratorios y por tanto estén familiarizados con su uso.

El sistema de visión deberá tener una configuración *eye-on-hand* para mayor versatilidad en diferentes aplicaciones. Una cámara *eye-on-hand* consiste en tener el sensor óptico montado en el elemento terminal del robot, moviéndose conjuntamente con él (figura 1.1). De esta manera se puede configurar fácilmente diferentes puntos de captura con sus respectivos planes de trabajo. Esta configuración también permite desarrollar aplicaciones de alineamiento, como por ejemplo moverse respecto a un código QR hasta una posición concreta relativa a él, aunque estas no se tratarán en el presente trabajo. El montaje de la cámara también deberá permitir el acople de un

elemento terminal (como una pinza, tal como se muestra en la figura 1.1), de manera que la cámara quede entre el robot y dicho elemento terminal.

En cuanto al código del sistema de visión, se pretende que logre distinguir diversos tipos de piezas entre sí con la mayor fiabilidad posible. Una vez hallada la pieza, deberá calcular su posición y orientación relativa a la base del robot para así poder realizar un comando de pick and place. El código se basará en la librería de código abierto OpenCV.

Respecto al sistema de comunicaciones, el objetivo es tener una doble comunicación robot-Raspberry y Raspberry-PC. Para la comunicación raspberry-robot, deberá hacerse por vía ethernet o Wi-Fi (a elección del usuario) y ser capaz de transmitir variables (coordenadas, strings, numéricas y booleanas). En cuanto a la comunicación raspberry-PC, se pretende que funcione simplemente para visualizar y controlar remotamente el código que ejecuta Raspberry desde un PC.

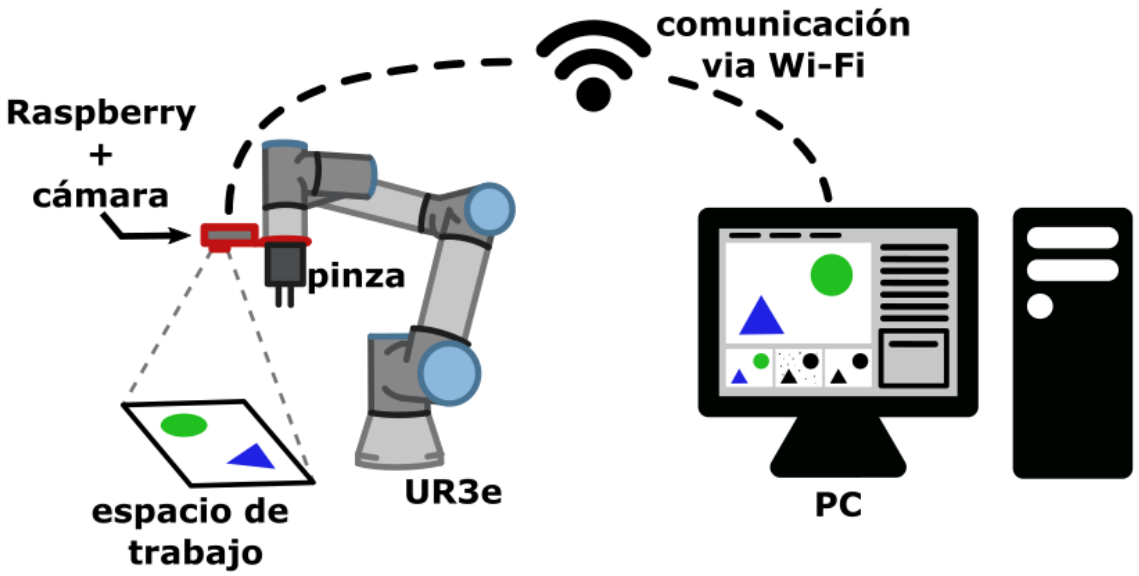


Figura 1.1. Dibujo del sistema

Para lograr estos objetivos se tendrán que llevar a cabo una serie de tareas, las cuales se explicitan a continuación:

- Diseñar y fabricar el sistema de sujeción al robot
- Programar la calibración óptica y geométrica de la cámara
- Programar el reconocimiento y localización de objetos
- Programar el sistema de comunicaciones

- Realizar pruebas del sistema completo

1.2. Alcance del trabajo

Se pretende cumplir con todos los objetivos expuestos anteriormente, de manera que se obtenga un sistema de visión *eye-on-hand* funcional que permita realizar operaciones de pick and place con posiciones de pick relativas al objeto. Dicho sistema funcionará sobre cualquier robot e-series de Universal Robots, concretamente se testeará sobre un UR3e.

2. Componentes utilizados

En este apartado se detallan los componentes necesarios para la realización del proyecto. Se describe cada uno de ellos y se especifica su función en el sistema. También se explica la elección de cada componente concreto y se dan alternativas de otros productos similares que podrían realizar la misma función. Cabe destacar que en este apartado no se incluye el sistema de sujeción al robot (el cual cuenta con su propio apartado) ya que se diseñó y fabricó desde cero mediante impresión 3D.

La lista de componentes utilizados es la siguiente:

- UR3e
- Raspberry Pi 3 Model B
- Raspberry Camera Module v1
- Ordenador / dispositivo móvil
- Router inalámbrico

2.1. UR3e

El UR3e es un robot colaborativo de la empresa Universal Robots. Se trata de un brazo articulado con seis grados de libertad (6 articulaciones rotativas), lo que le permite una gran capacidad de posicionamiento y orientación del elemento terminal en el espacio de trabajo. Dicho espacio de trabajo (mostrado en la figura 1.2) tiene una geometría cuasi esférica de 500 mm de radio, el menor dentro de la gama e-series de Universal Robots. Esto podría parecer una desventaja, pero su tamaño compacto lo hacen ideal para las aulas y laboratorios donde no se dispone de mucho espacio. También significa que es el robot más económico dentro de la gama e-series, por lo que se ajusta mejor a los reducidos presupuestos de los centros educativos. En cuanto a su capacidad de carga, la cual es de 3 kg, es suficiente para los ejercicios educativos más comunes, en los que se suelen manipular piezas pequeñas y ligeras mediante pinzas o ventosas que no suelen superar los 2 kg de peso.

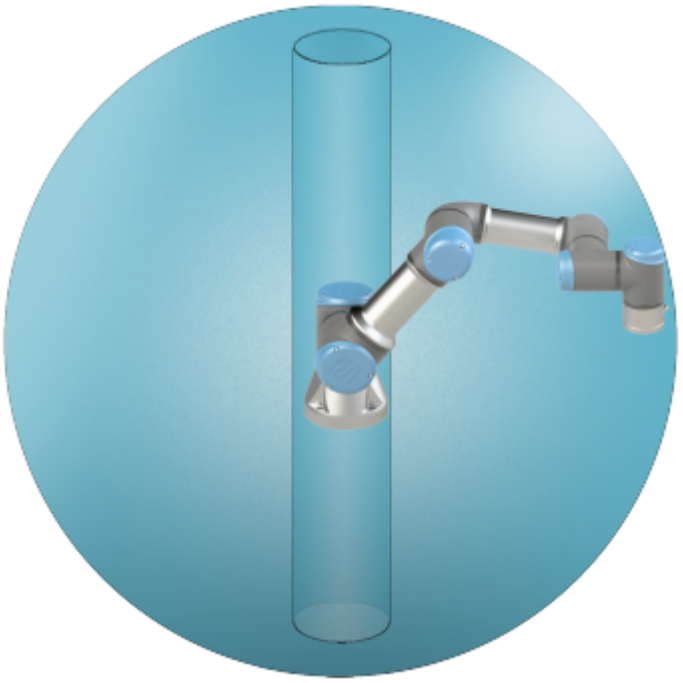


Figura 1.2. Espacio de trabajo del UR3e [1]

Otra característica crucial de este robot es el hecho de que se trata de un robot colaborativo, por lo que está diseñado para compartir el mismo espacio de trabajo que el operario y permite la interacción entre ambos. Su estructura mecánica de materiales ligeros está diseñada para evitar atrapamientos y cuenta con bordes redondeados para reducir el daño en caso de golpes al operario. También cuenta con sensores de par y fuerza que bloquean el robot en caso de superar valores predeterminados, lo que permite evitar el riesgo de aplastamiento y reducir el impacto de posibles golpes. Es por esto que, a diferencia de un robot industrial tradicional, los robots colaborativos no necesitan de una jaula de protección ni otras medidas de seguridad que elevan el coste y volumen de la instalación.

Otra funcionalidad que aportan los sensores de fuerza y par antes mencionados es el guiado manual del robot. El guiado manual permite al operario mover al robot directamente con las manos, enseñándole así las posiciones necesarias de una manera rápida e intuitiva. Esto agiliza en gran medida el proceso de programación.

En cuanto a la programación del robot, toda la gama e-series cuenta con una interfaz gráfica llamada PolyScope (figura 1.3). Dicha interfaz da acceso a una programación sencilla mediante nodos de programa, a la vez que permite a los usuarios más avanzados utilizar el lenguaje URScript para escribir programas de una manera más tradicional. También da acceso a la configuración de

toda clase de parámetros de la instalación del robot (seguridad, entradas/salidas, protocolos de comunicación, ...).

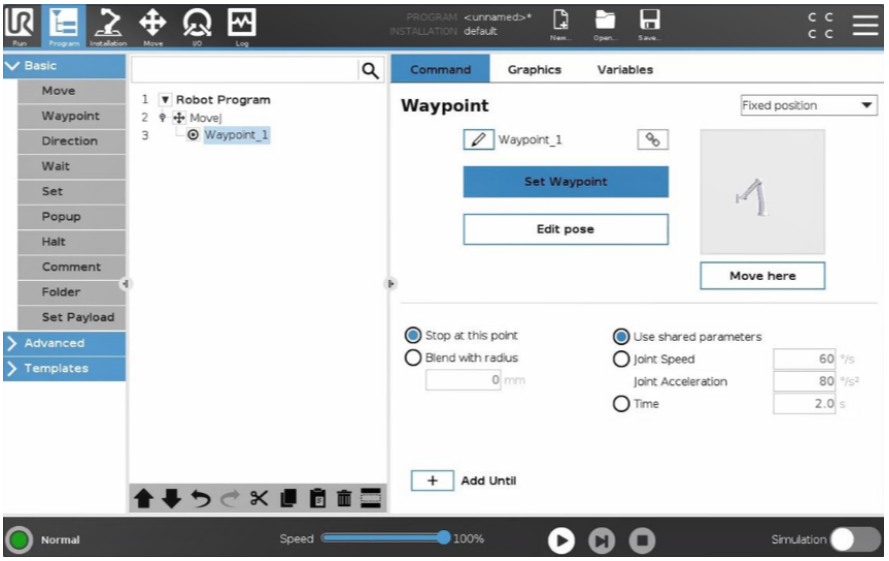


Figura 1.3. Captura de la interfaz gráfica Polyscope

En última instancia, se ha elegido este modelo de robot ya que es del que se dispone en el laboratorio de robótica A5.4 de la EEBE. No obstante, dado que el proyecto se desarrollará estando en convenio de prácticas en la empresa Universal Robots, el sistema se podrá testear en todos los robots de la gama e-series (UR3e, UR5e, UR10e y UR16e). Debido a que todos comparten el mismo software y la misma sujeción de elemento terminal el sistema se podrá implementar en cualquiera de ellos sin necesidad de apenas cambios.

2.2. Raspberry Pi 3 Model B

Para ejecutar tanto el procesado de imagen como la interfaz gráfica se utilizará la Raspberry Pi 3 Model B (figura 1.4). Se trata de un ordenador monoplaca que cuenta con un procesador de cuatro núcleos Broadcom BCM2837 de 64 bits trabajando a 1.2GHz y una RAM de 1 GB [2]. Esta potencia de procesado es suficiente para ejecutar el procesado de imagen en un tiempo aceptable, teniendo en cuenta que en aplicaciones educativas no se requiere un tiempo de ciclo excesivamente corto.

Figura 1.4. Raspberry Pi 3 Model B

La placa cuenta con las siguientes conexiones: un puerto HDMI, un puerto Ethernet, 4 puertos USB 2.0, 40 pines GPIO, un puerto CSI para la conexión de la cámara, un puerto DSI para conectar un display, una conexión jack de 4 polos y un puerto para tarjeta Micro SD. En cuanto a conexiones inalámbricas, cuenta con un chip BCM43438 con LAN inalámbrica (Wi-Fi) y Bluetooth. Para su alimentación dispone de un puerto micro-USB que se conecta a una fuente de 5.1 V y hasta 2.5 A.

En cuanto al sistema operativo, se ha decidido optar por el oficial de Raspberry: Raspberry Pi OS. Dicho sistema operativo ha de instalarse en una tarjeta micro SD desde un ordenador para luego insertarse en la placa, donde la tarjeta micro SD hará a la vez de memoria ROM. Para este proyecto se ha usado una tarjeta micro SD de 32 GB, pero cualquiera de mayor capacidad serviría igual.

Con unas dimensiones de 86 x 56 x 18 mm, la Raspberry Pi 3 Model B es lo suficientemente compacta como para montarse entre el elemento terminal y el robot.

Cabe comentar que en el momento en el que se escribe este documento Raspberry cuenta con una nueva gama superior de ordenadores monoplaca, las Raspberry Pi 4. Las principales diferencias con la Raspberry Pi 3 son que cuenta con un mejor procesador y con la posibilidad de hasta 8 GB de RAM. En cuanto a las dimensiones y la conectividad són prácticamente idénticas, es por eso que se podría utilizar este nuevo modelo en el sistema sin necesidad de cambios mayores. La única razón por la que se usa el modelo antiguo es porque ya se disponía de uno al iniciar el

proyecto, y por la escasez global de stock en ese momento no se pudo conseguir una Raspberry Pi 4.

También se planteó el uso de la Raspberry Pi Zero W por su tamaño reducido. Pero tras diversas pruebas se llegó a la conclusión de que no disponía de la potencia computacional suficiente para ejecutar el procesado de imágenes de forma eficiente.

2.3. Raspberry Camera Module v1

Para capturar las imágenes se utilizará la Raspberry Camera Module v1 (figura 1.5). Se trata de una cámara producida por la propia Raspberry y diseñada para conectarse directamente a sus ordenadores monoplaca mediante cable CSI. Cuenta con un sensor RGB OmniVision OV5647 de 5 megapíxeles de resolución (2592 × 1944 pixels) [3] . Tiene un campo de visión horizontal de 53.50 ± 0.13 grados y uno vertical de 41.41 ± 0.11 grados. El tamaño del módulo es de 25 x 23.862 mm, haciéndola lo suficientemente compacta como para usarse en la configuración *eye-on-hand*.

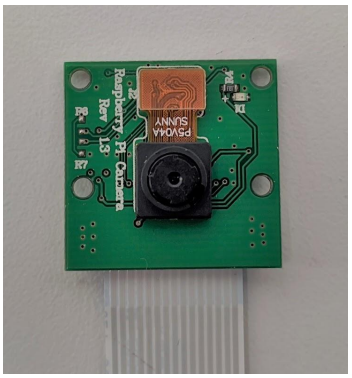


Figura 1.5. Raspberry Camera Module v1

Cabe mencionar que en el momento en el que se escribe este documento Raspberry tiene disponibles dos nuevos modelos de cámara: Camera Module v2 y Camera Module 3. Su principal diferencia respecto a la primera versión radica en el sensor, que pasa a ser de 8 megapixels (Sony IMX219) en el caso de la versión 2 y de 12 (Sony IMX708) en el caso de la tercera. Cualquiera de estas dos nuevas versiones se puede utilizar en lugar de la primera, con posibles mejoras en la precisión del sistema a costa de un ligeramente peor tiempo de ejecución (ya que las imágenes son más pesadas). La versión 3 también cuenta con un modelo gran angular, pero para esta aplicación no interesa ya que implica mayor distorsión en los bordes de la imagen.

La única razón por la que se ha utilizado la Raspberry Camera Module v1 y no las versiones más nuevas es, de nuevo, por su disponibilidad a la hora de iniciar el proyecto. Dicho esto, se ha

realizado todo el diseño del sistema teniendo en cuenta la posibilidad de implementar las nuevas versiones de las cámaras sin la necesidad de cambios mayores.

2.4. Ordenador

Para poder acceder al Raspberry OS y controlar el sistema remotamente se necesita un dispositivo con VNC Viewer instalado. En este caso no se escoge ningún producto concreto, ya que mientras pueda ejecutar VNC Viewer, tenga pantalla y tenga conexión Wi-Fi cualquier dispositivo sirve. Se recomienda el uso de un ordenador portátil o de sobremesa (Windows, macOS o Linux) ya que el teclado y ratón son los métodos de entrada más cómodos. Dicho esto, un dispositivo móvil Android, iPhone o iPad también sirven para visualizar la interfaz, aunque sea más tedioso usar la pantalla táctil como único método de entrada.

2.5. Router inalámbrico

El router inalámbrico es un componente opcional y sirve únicamente para conectar el robot UR3e a la misma red Wi-Fi que la Raspberry, permitiendo así la comunicación inalámbrica entre ambos. Si se dispone de otro método de conexión del robot a la red wifi (p.e. conexión por cable ethernet al router principal) este componente no es necesario. En caso de no disponer de un punto de acceso o simplemente querer evitar conectar el robot a una red Wi-Fi, se puede utilizar una conexión directa de la Raspberry al UR3e mediante cable Ethernet.

Para este proyecto se ha utilizado el router inalámbrico TL-WR802N de la marca TP-Link (figura 1.6). Cuenta con una tasa de transferencia de 300Mbps, más que suficiente para los datos ligeros que se van a transmitir. Entre los diferentes modos de operación que admite se usa el modo cliente, que permite conectar un dispositivo sin conectividad inalámbrica (en este caso el UR3e) a una red Wi-Fi.

Figura 1.6. Router inalámbrico TL-WR802N [4]

3. Sistema de sujeción al robot

Para poder capturar imágenes desde posiciones precisas es necesario un elemento que ancle tanto la cámara como la Raspberry a la muñeca del robot, de manera que queden fijadas de manera firme y se puedan enseñar posiciones de captura de manera precisa. En este capítulo se detalla el proceso de diseño y fabricación de dicho sistema de sujeción al robot, señalando en cada pieza si hubo prototipos anteriores fallidos y la causa.

3.1. Criterios de diseño

Antes de empezar a diseñar en SolidWorks fue necesario establecer primero unos criterios de diseño que el sistema ha de cumplir. De esta manera se tiene claro desde el principio las características que debía tener cada componente y se evita la fabricación de prototipos fallidos.

Estos son los criterios de diseño que se establecen para el sistema de sujeción:

- **Tamaño compacto**: dado que irá montado en el robot para la configuración *eye-on-hand*, es necesario que sea lo más compacto posible. De esta manera no estorbará tanto en los movimientos del robot y se evitarán golpes. También se reduce así la cantidad de material necesario para su fabricación.
- **Ligereza**: teniendo en cuenta que la capacidad de carga del robot es de 3 kg, será necesario diseñar el sistema lo más ligero posible. De esta manera podrá aprovecharse la capacidad del robot para instalar elementos terminales y coger piezas de mayor masa. Para ello se evitará utilizar más material del necesario en zonas donde no proporcione una mejora estructural significativa.
- **Firmeza**: la función principal de este sistema es mantener la cámara en una posición fija relativa a la muñeca del robot. Para ello es necesario que la estructura sea lo más firme posible, evitando juegos y movimientos indeseados que modifiquen la posición de la cámara, ya que ello introduciría imprecisión al sistema. Para conseguir este objetivo hay que intentar trabajar con tolerancias lo más bajas posibles (para evitar juego entre las piezas) y diseñar una estructura rígida que impida deformaciones no deseadas.
- **Optimizado para impresión 3D**: dado que todas las piezas serán fabricadas mediante impresión 3D hay que optimizarlas para este proceso. Para ello se han de seguir dos directrices. La primera es que cada pieza ha de tener una cara plana y amplia que sirva como base estable durante la impresión. En segundo lugar, teniendo en cuenta la dirección de impresión, hay que evitar en la medida de lo posible tener material en

voladizo para minimizar el material de soporte necesario (idealmente no tendría que necesitar ningún soporte).

- **Capacidad de acoplar un elemento terminal**: puesto que se tiene una configuración *eye-on-hand*, el sistema ha de quedar entre la muñeca del robot y cualquier elemento terminal que se quiera instalar. Para ello es necesario que la estructura replique la geometría del acople que tiene la muñeca el UR3e para que se pueda encajar elemento terminal. De esta manera se podrá atornillar al robot tanto el elemento terminal como la estructura que sujeta la cámara con los mismos tornillos M6.
- **Acceso a todos los puertos de la raspberry:** todos los puertos de la raspberry han de ser accesibles una vez montado el sistema, en especial los puertos USB, Ethernet y CSI (para el cable de la cámara). De esta manera se permite usar la comunicación por cable ethernet sin necesidad de desmontar el sistema. También resulta conveniente tener acceso a los puertos USB para transferir archivos cómodamente, especialmente útil durante la fase de desarrollo del proyecto. Por último, este criterio da pie a usar el mismo sistema de sujeción para desarrollar otras aplicaciones que necesiten utilizar dichos puertos.
- **Protección de los componentes**: además de sujetar los componentes en su sitio, la estructura debe protegerlos en caso de un golpe. Para ello debe asegurarse que la estructura cubra los componentes en todas las direcciones, de manera que independientemente de la dirección del golpe sea la estructura la que reciba la colisión.
- **Facilidad de montaje**: para evitar confusión y roturas a la hora del montaje, este ha de ser sencillo e intuitivo. También ha de evitarse la dependencia de material extra (tornillería, bridas,...) para reducir el coste y la complejidad del sistema. Es por esto que se usará un mecanismo de click para unir las diferentes piezas.
- **Bordes redondeados**: dado que se montará sobre un robot colaborativo y por ende compartirá espacio de trabajo con el operario es necesario reducir el daño en caso de golpe. Es por esto que se evitarán las esquinas puntiagudas y los bordes afilados dentro de lo posible, intentando que todos los bordes sean redondeados.

3.2. Diseño final

Teniendo en cuenta todos los criterios de diseño mencionados en el apartado anterior, se diseñó en Solidworks el modelo que muestra la figura 3.1. El sistema consta de cuatro partes, tal como se muestra en la vista explosionada de la figura 3.2, las cuales son: la sujeción de la cámara, la sujeción raspberry-robot, la tapa protectora y el acople para el elemento terminal. Cada una de estas partes se explicará en detalle en los siguientes apartados. Los planos de cada una de las piezas se encuentran en el anexo B.

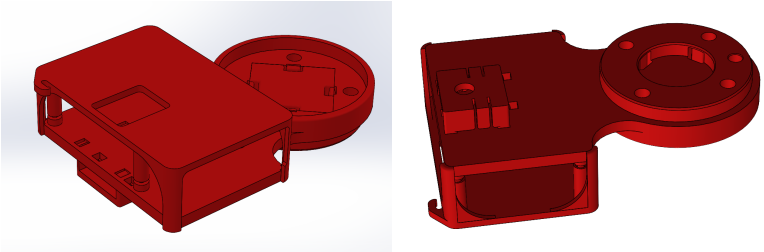


Figura 3.1. Modelo CAD del sistema de sujeción

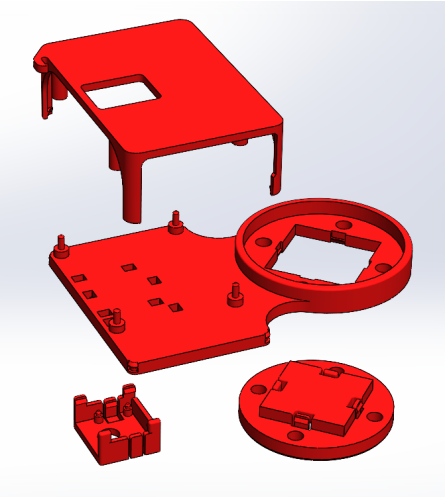


Figura 3.2. Vista explosionada del modelo CAD del sistema de sujeción

La decisión de este sistema por partes se hizo para cumplir con dos de los criterios de diseño: la optimización para impresión 3D y la facilidad de montaje. Al dividir el sistema en varias piezas resulta más sencillo tener en cada una de ellas una superficie amplia y plana que sirva de base durante la impresión, ya que puedes orientar cada una de ellas como convenga para la impresión. También facilita evitar un exceso de material en voladizo que necesite soporte, especialmente separando la sujeción raspberry-robot del acople de elemento terminal, tal como se explicará más adelante. En cuanto a la facilidad de montaje, el separar las piezas permite encajar la cámara y la raspberry por separado para luego ensamblar el conjunto, ya que encajar ambas en una misma pieza resultaría más incómodo.

Para llegar a este modelo definitivo se pasó por un proceso de prototipado y re-diseño de cada uno de sus componentes, tal como se explica más adelante para cada una de las piezas.

3.2.1. Sujeción raspberry-robot

Esta pieza sirve como base de todo el sistema, es la parte que se ancla al robot para que el resto de piezas se encajen en esta. Como se puede apreciar en la figura 3.3 la pieza consta de dos zonas: la que se queda entre el robot y el elemento terminal (de forma redonda) y la que sujeta la raspberry y la cámara (de forma rectangular). Se observa como se ha diseñado de manera que la raspberry quede lo más pegada posible al robot, haciendo el conjunto más compacto. También se han redondeado todas las esquinas para cumplir con el criterio de diseño.

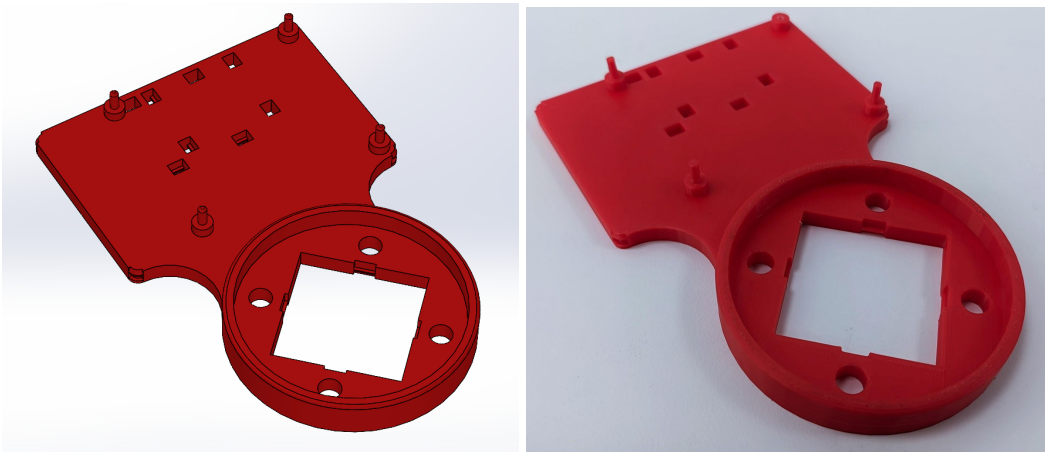


Figura 3.3. Sujeción raspberry-robot: modelo CAD (izquierda) y parte real (derecha)

Se empezará explicando la zona que se queda entre el robot y el elemento terminal, en la cual destaca un gran agujero cuadrado en el centro. Este agujero es el punto de anclaje del acople para el elemento terminal. Se consideró incluir el acople en la misma pieza formando un solo sólido, pero ello implicaba la necesidad de una excesiva cantidad de material de soporte, incumpliendo uno de los criterios de diseño. Tal como marca la figura 3.4 en amarillo, toda la zona de sujección de la raspberry quedaría en voladizo y necesitaría material de soporte. Es por ello que se decidió imprimir ese elemento a parte, consiguiendo así que esta pieza se imprimiera sin necesidad de ningún soporte.

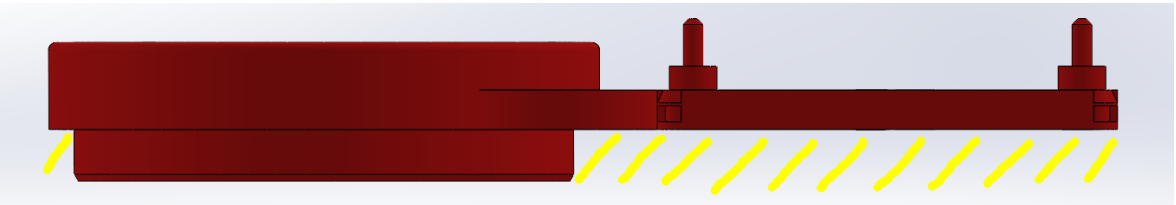


Figura 3.4. Material de soporte necesario si no se separa el acople para el elemento terminal

Obviando ahora el agujero, esta parte de la pieza está diseñada para encajar en la muñeca de cualquier robot UR incluido el UR3e. Cuenta con cuatro agujeros para pasar los tornillos M6 que sujetan todo el conjunto al robot, con un diámetro 0.5 mm mayor al del tornillo para que puedan pasar con facilidad. También dispone de un reborde de 3 mm de grosor que cubre la muñeca del robot, de manera que la pieza quede fija mientras se atornilla el conjunto, facilitando la operación.

En cuanto a la zona rectangular, cuenta con los elementos necesarios para encajar la cámara, la raspberry y la tapa protectora. Para la cámara cuenta con dos sets de cuatro agujeros con la forma necesaria para encajar la cámara mediante el sistema de click (el cual se detalla más adelante). Se cuenta con dos sets de agujeros para poder instalar la cámara tanto en horizontal como en vertical, a elección del usuario, sin tener que cambiar ninguna pieza. En cuanto a la raspberry, se dispone de cuatro pilares que se alinean con los cuatro agujeros de la placa, de manera que al insertarse limitan su movimiento sobre el plano horizontal. También tienen la función de mantener la placa a 3 mm de altura para salvar los puntos de soldadura y el lector de tarjetas microSD de su lado inferior. Para encajar la tapa protectora, además de los pilares ya mencionados que sirven de alineación, las esquinas del rectángulo cuentan con la forma necesaria para usar el sistema de click (figura 3.5).

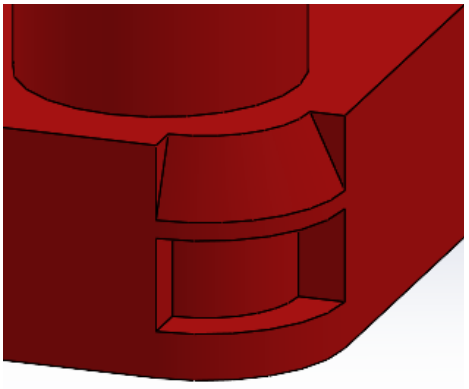


Figura 3.5. Detalle de las esquinas del rectángulo

Para esta pieza solo se necesitó un prototipo que funciona correctamente con el robot UR3e para el que fue diseñado, al igual que con el UR5e. Sin embargo con el UR10e y UR16e, al tener una muñeca más gruesa, la Raspberry chocaba con ella al no estar lo suficientemente separada. Es por esto que también se ha elaborado un segundo diseño que soluciona este problema, dejando más margen entre la zona redonda que encaja en la muñeca del robot y la zona rectangular que sujeta la Raspberry.

3.2.2. Sujeción de la cámara

Esta pieza tiene la función de sujetar el módulo de cámara para poder anclarla a la sujeción raspberry-robot. Dado que se requiere que la cámara quede lo más firme posible, esta pieza es la que necesita las tolerancias más ajustadas, lo que causó que fuera la que tuvo más prototipos fallidos tal como se detallará más adelante. Tras todo el proceso de prototipado y re-diseño se obtuvo el modelo que se muestra en la figura 3.6.

Figura 3.6. Sujeción de la cámara: modelo CAD (izquierda) y parte real (derecha)

Para asegurar una sujeción firme se han usado tres métodos. El primero es hacer las paredes interiores de la pieza de las mismas dimensiones exactas del módulo de cámara, de manera que encaje sin ningún tipo de juego. En segundo lugar se han usado cuatro pilares (igual que en la sujeción de la raspberry) que alinean la cámara y acaban de impedir, junto a las paredes, su desplazamiento sobre el plano horizontal. Estos pilares también sirven de apoyo para el módulo, manteniendo la lente en la altura deseada. Para restringir el desplazamiento vertical se han utilizado unas pestañas diseñadas para que el módulo pueda entrar en sentido pero una vez dentro impida su salida en el sentido contrario (figura 3.7).

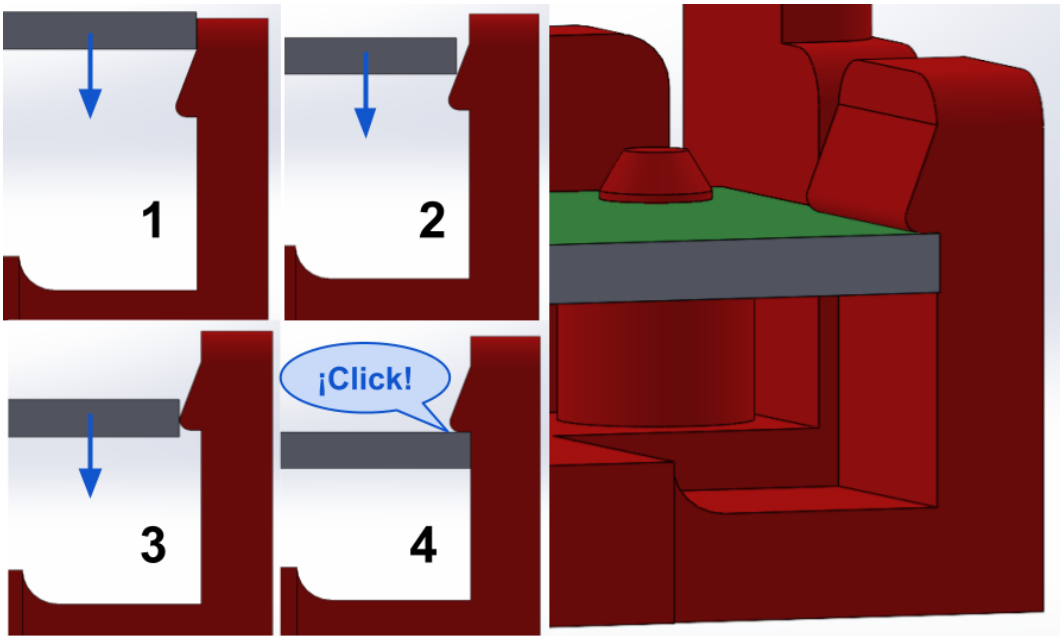


Figura 3.7. Detalle de la pestaña que sujeta el módulo de cámara

Estas pestañas también sufrieron un proceso de rediseño, ya que en el primer prototipo eran demasiado rígidas y no cedían al intentar insertar el módulo de cámara. Para solucionarlo se alargaron los surcos que separan la pestaña del resto de la pieza, extendiéndose por el plano horizontal tal como se aprecia en la figura 3.8. Al alargar la pestaña se aumentó su flexibilidad hasta permitir la inserción del módulo, pero sin ser demasiado flexible como para dejarlo salir una vez dentro. Como se puede observar en la figura 3.8 se consiguió que el módulo de cámara encaje de forma precisa en la pieza.

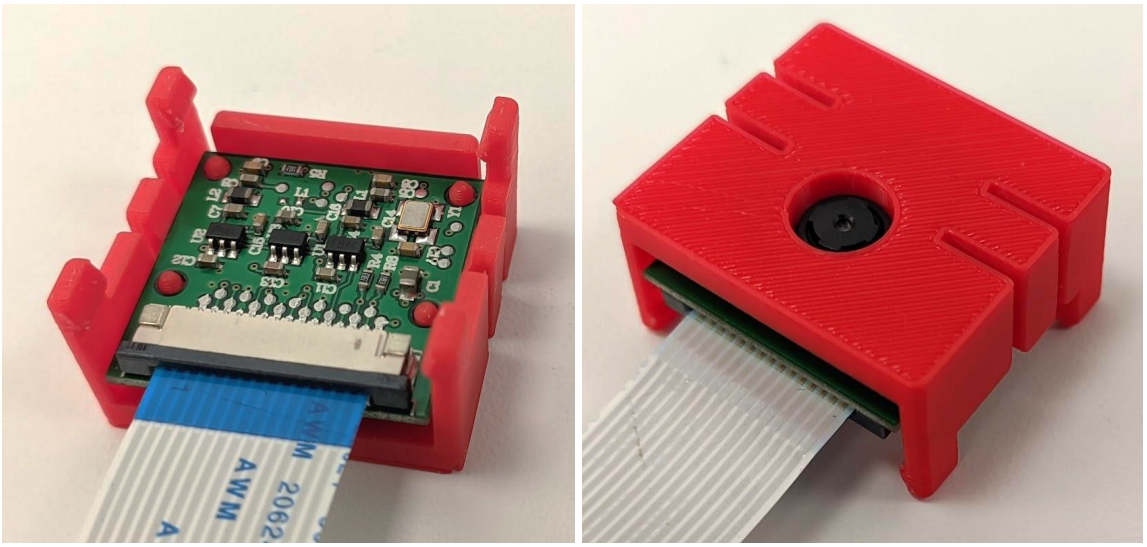


Figura 3.8. Sujeción de la cámara con el módulo de cámara montado

Para encajar la pieza a la sujeción raspberry-robot se hace uso de un mecanismo con una pestaña muy similar, en este caso de un tamaño ligeramente mayor. Estas pestañas se insertan en unos agujeros con un chaflán que asiste en la flexión durante la inserción. Una vez insertadas las pestañas quedan sujetas por un saliente dentro del agujero (ver detalle de la figura 3.9). Cuando se encajan las piezas con este mecanismo se produce un distintivo sonido de “click”, es por esto que se le llama mecanismo click.

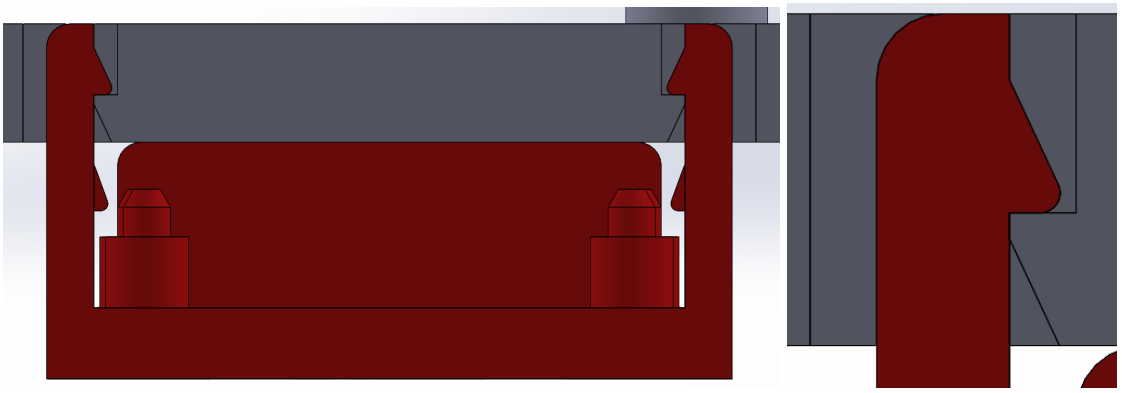


Figura 3.9. Vista de sección del montaje de la sujeción de la cámara a la sujeción raspberry-robot (izquierda) y detalle del mecanismo de click (derecha)

Para poder hacer este diseño se tuvieron que tomar medidas manuales de la Raspberry Camera Module v1, ya que la web de Raspberry no proporciona planos oficiales y los disponibles en la web no encajaban con el módulo real (concretamente en la posición de los agujeros) y se necesitaban medidas precisas. Dadas las pequeñas dimensiones del módulo la toma de medidas fue complicada, lo que implicó la impresión de dos prototipos fallidos hasta encontrar las medidas exactas.

Cabe mencionar que también se han diseñado los modelos CAD para las nuevas versiones del Raspberry Camera Module con las medidas de los planos oficiales disponibles en la web de Raspberry [5]. Aunque no se han podido testear, por lo que no se puede garantizar que encajen a la perfección y no sea necesario ajustar algunas medidas.

3.2.3. Tapa protectora

La función principal de esta pieza es proteger la Raspberry Pi de golpes, así como mantenerla sujeta al resto del sistema. Se puede apreciar en la figura 3.10 que cuenta con una ventana rectangular, cuya función es simplemente dejar pasar el cable CSI de la cámara para evitar doblarlo en exceso.

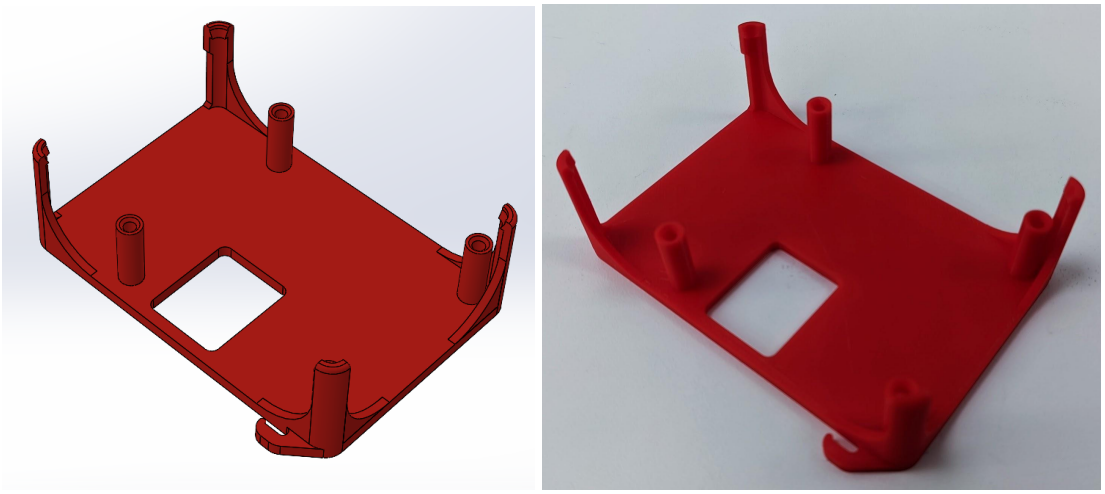


Figura 3.10. Tapa protectora: modelo CAD (izquierda) y parte real (derecha)

Para asegurar la protección de la Raspberry Pi se diseñó esta pieza con dimensiones ligeramente superiores a las de la placa en todas las direcciones, sin sobrepasarse para mantener el criterio de diseño de un tamaño compacto. De esta manera se asegura que independientemente de la dirección del golpe será la tapa la que reciba el golpe en vez de la placa. Dicho esto, la pieza tiene un diseño de laterales abiertos para dar acceso a los puertos de la Raspberry Pi (figura 3.11), lo que disminuye la capacidad de protección. Se decidió optar por esta accesibilidad en detrimento de la protección ya que no se consideró necesario un nivel de protección tan elevado teniendo en cuenta que la cámara no debería entrar en contacto con ningún objeto, al contrario que el elemento terminal.

Figura 3.11. Sistema completo ensamblado

Como se comentó en el apartado 3.2.1, la sujeción raspberry-robot cuenta con cuatro pilares que limitan el movimiento de la Raspberry Pi sobre el plano horizontal. La tapa, por su parte, tiene

cuatro columnas con agujeros en la punta que se alinean con los pilares mencionados anteriormente. De esta manera, entre las dos piezas se acaba de limitar el movimiento de la placa en el eje vertical, tal como muestra la figura 3.12.

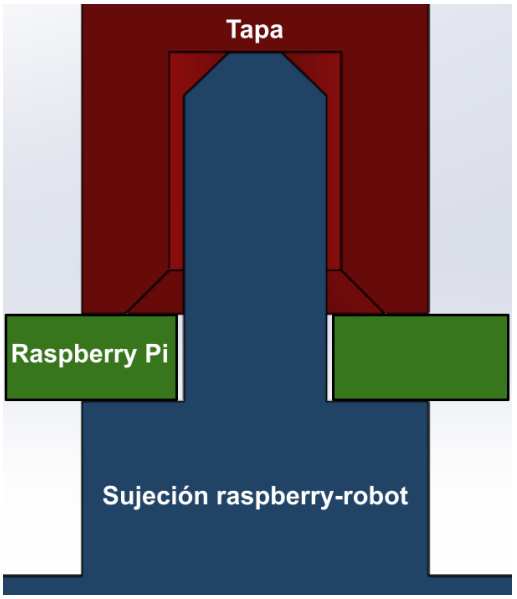


Figura 3.12. Detalle de los pilares de sujeción de la Raspberry Pi

Para anclar la tapa a la sujeción raspberry-robot se hace uso otra vez del ya mencionado sistema de click. La tapa cuenta con cuatro patas en sus esquinas, las cuales cuentan con la pestaña del mecanismo de click en sus extremos. En este caso se ha adaptado el mecanismo a la curvatura de las esquinas redondeadas, tal como muestra la figura 3.13, pero el funcionamiento sigue siendo el mismo.

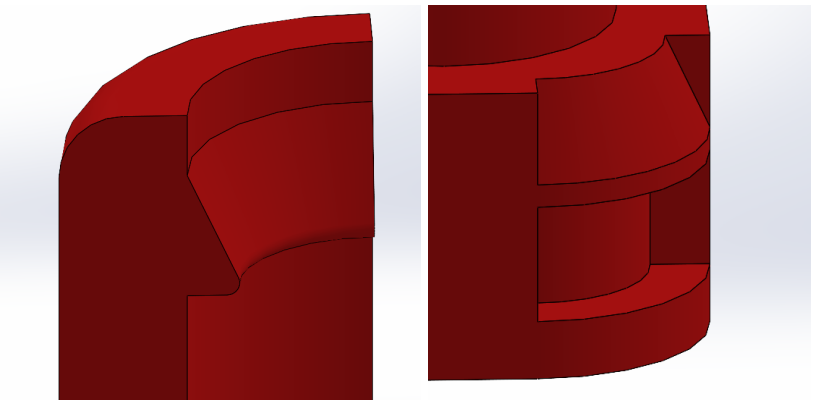


Figura 3.13. Detalle del mecanismo de click adaptado a las esquinas redondeadas

Esta pieza pasó por un par de prototipos fallidos. Para el primer prototipo, con tal de ahorrar material y aligerar la pieza, se decidió aplicar una matriz hexagonal a toda la superficie de la tapa, tal como muestra la primera imagen de la figura 3.14. Sin embargo esto hizo la pieza muy flexible (como se observa en la segunda imagen de la figura 3.14) y se desencajaba con mucha facilidad de la sujeción raspberry-robot. Tampoco se consideró el cable de la cámara, por lo que no se añadió su ventana a la tapa.

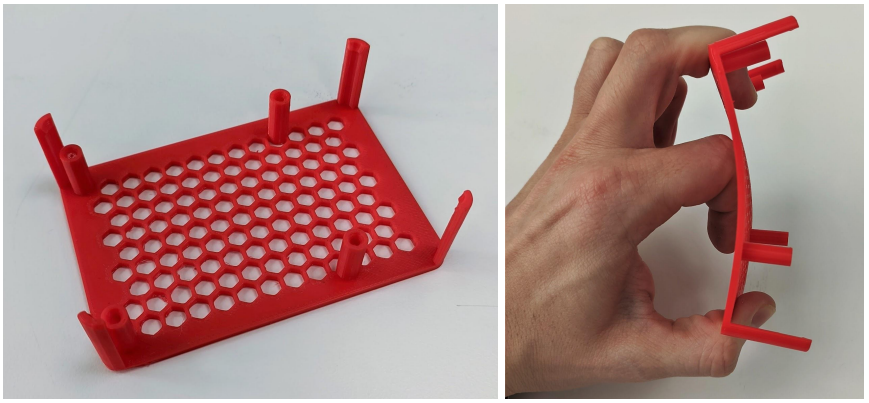


Figura 3.14. Primer prototipo de la tapa protectora

Para aumentar la rigidez de la pieza, el segundo prototipo se diseñó con una superficie sólida y un grosor 1mm mayor. También se le añadió la ventana para el cable de la cámara. Si bien este prototipo aguantaba bastante mejor, las patas seguían siendo demasiado flexibles y se desencajaban con cierta facilidad.

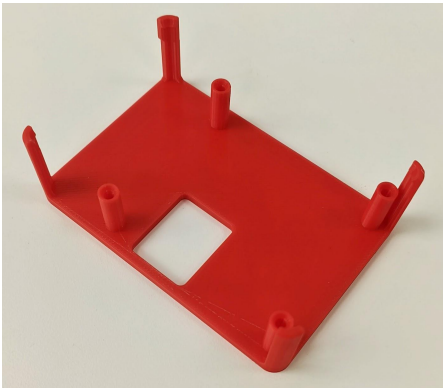


Figura 3.15. Segundo prototipo de la tapa protectora

Para el diseño final se reforzaron estas patas añadiendo redondeos a las esquinas de sus bases, consiguiendo una rigidez considerablemente mayor y evitando por fin que la tapa se desencajara. A este último diseño también se le añadió una nueva función: sujetar el cable de alimentación de la

raspberry. Con los dos primeros prototipos de esta tapa el cable quedaba colgando del puerto micro-USB y según como se orientaba el robot el propio peso del cable causaba su desconexión. Para aguantar el peso del cable y solucionar este problema se añadió a la tapa una sujeción del cable (figura 3.16). Esta sujeción se ha diseñado con una entrada menor al diámetro del cable, de manera que para introducirlo hay que ejercer una ligera presión que deforma levemente el cable. Una vez dentro, el cable vuelve a su forma original y queda atrapado. Para sacarlo solamente hay que ejercer la misma presión que al entrar, la cual es lo suficientemente grande como para que el cable no se salga por su cuenta. Cabe detallar que en este proyecto se ha usado la antigua fuente de alimentación oficial de raspberry, la cual tiene un cable con unas dimensiones diferentes al de la fuente actual, tal como se aprecia en la figura 3.16. Para poder recrear el proyecto con la nueva fuente de alimentación, se ha diseñado también una variante de la sujeción del cable con las dimensiones adecuadas.

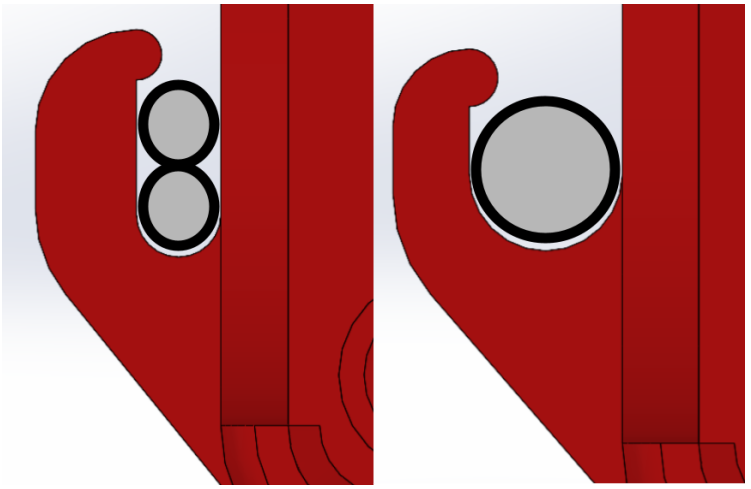


Figura 3.16. Detalle de la sujeción del cable de alimentación antiguo (izquierda) y nuevo (derecha)

3.2.4. Acople para elemento terminal

Esta pieza tiene la única función de servir de punto de acople para el elemento terminal. En la figura 3.17 se observa como se ha diseñado con la misma geometría y dimensiones que la muñeca de los robots e-series de Universal Robots (incluyendo el UR3e) con sus cuatro agujeros para los tornillos M6, el agujero para el pivote alineador y el hueco central.

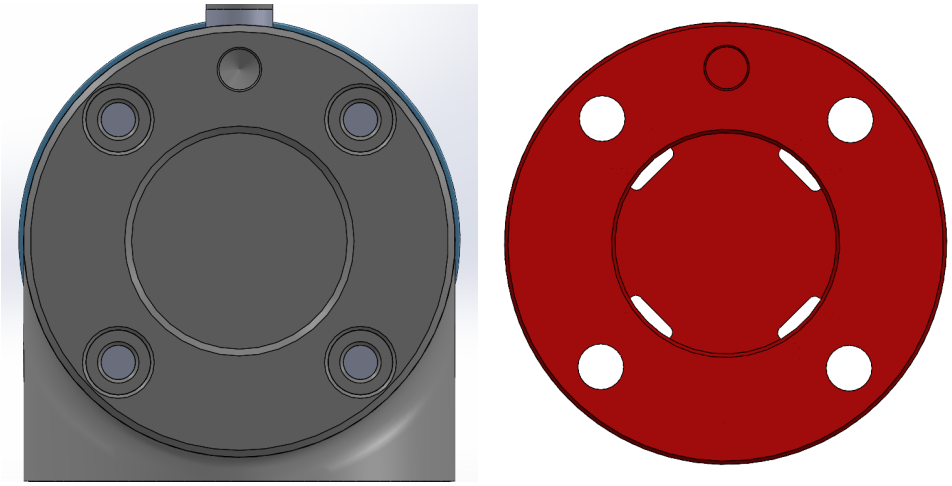


Figura 3.17. Muñeca de robot UR3e (izquierda) y pieza de acople para el elemento terminal (derecha)

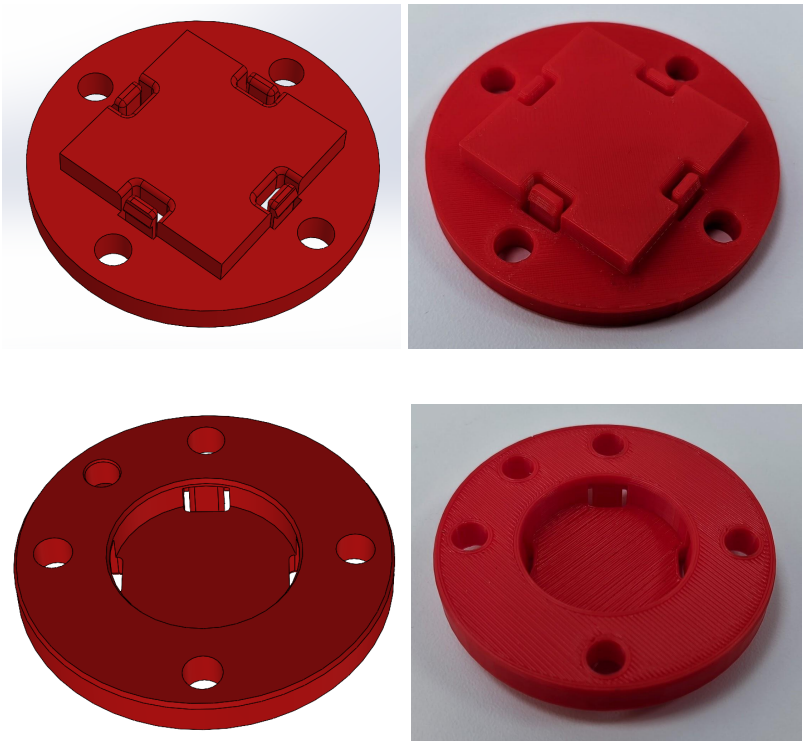


Figura 3.18. Acople para el elemento terminal: modelo CAD (izquierda) y parte real (derecha)

Tal como se ha explicado en el apartado 4.2.1, esta pieza se ha separado de la sujeción raspberry-robot para permitir la impresión de esta segunda sin soportes. Sin embargo no se puede evitar la necesidad de material de soporte para el hueco central del acople, haciendo de esta la única pieza que necesita dicho soporte (concretamente para el hueco central).

Otro beneficio de separar esta pieza es que permite poder montarla en distintas orientaciones. Para encajar el acople a la sujeción raspberry-robot se dispone de un cuadrado con un mecanismo de click en el centro de cada uno de sus lados. Como el cuadrado es simétrico se puede montar en cuatro posiciones diferentes. Así el pivote alineador del acople puede quedar en cuatro orientaciones diferentes respecto a la cámara, a elección del usuario.

3.3. Fabricación

Para producir todas las piezas se usó la tecnología de fabricación aditiva o impresión 3D. Concretamente se usó un filamento de PLA (ácido poliláctico) por su amplia disponibilidad, bajo coste y facilidad de impresión. Se consideró el uso de materiales más resistentes, pero dado que las piezas no se ven sometidas a esfuerzos mecánicos demasiado elevados se concluyó que estos no eran necesarios.

En cuanto a los parámetros de impresión, se usó una velocidad de impresión de 60 mm/s y una altura de capa de 0.1 mm para una máxima calidad de detalle. Esta calidad es solamente necesaria para los mecanismos de click, si el slicer lo permite se puede configurar para que solo aplique esa altura a las capas que incluyen dichos mecanismos, pudiendo usar una altura mayor (por ejemplo 0.2 mm) para el resto de capas y conseguir reducir así el tiempo de impresión. Para el relleno de las piezas se usó un patrón triangular al 20% de relleno (figura 3.19). Respecto al resto de parámetros, se usaron los valores por defecto que tiene el slicer para el material PLA. Cabe recordar que el acople para el elemento terminal necesita material de soporte y por tanto es necesario activar la generación de soportes para esa pieza en concreto.

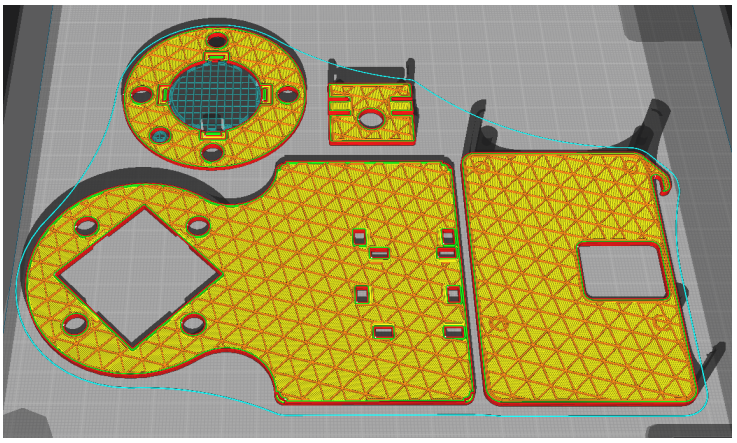


Figura 3.19. Patrón de relleno para la impresión 3D

Con estos parámetros de impresión el sistema completo gasta 70 g de material, de los cuales 1 g corresponde al material de soporte y 69 g a las piezas en sí. Si a esto se le suman los 54 g de la

Raspberry y la cámara se obtiene un peso total del sistema de 123 g. Por tanto ocupa tan solo el 4.1 % de la carga útil del UR3e (3 kg), cumpliendo así el criterio de diseño de ligereza.

3.4. Montaje del sistema

Una vez impresas todas las piezas hay que seguir una serie de pasos para montar el sistema. El primero es encajar el acople del elemento terminal a la sujeción raspberry-robot, teniendo la orientación en la que se quiera tener el pivote de alineamiento respecto a la cámara. Para encajarlo simplemente hay que alinear el cuadrado con su respectivo agujero y presionar hasta escuchar un click. El segundo paso es encajar el módulo de cámara en su respectiva sujeción. Para ello se debe alinear con las tres paredes de la sujeción y luego presionar ligeramente (asegurándose que los pilares están bien alineados con los agujeros) hasta escuchar un click. Una vez encajado debe quedar como se muestra en la figura 3.20.

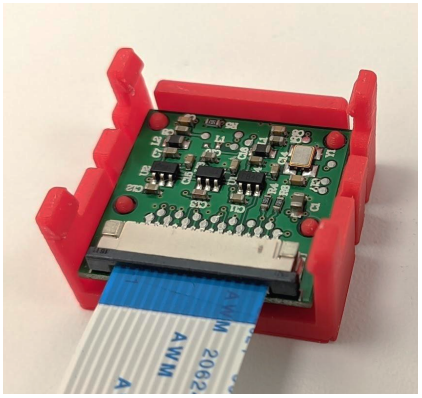


Figura 3.20. Segundo paso del montaje: encajar la cámara en su soporte

A continuación se ha de encajar la sujeción de la cámara (con la cámara ya insertada) en la sujeción raspberry-robot. Una vez elegida la orientación de la cámara (horizontal o vertical) hay que alinear las cuatro pestañas de la sujeción de la cámara con sus respectivos cuatro agujeros en la sujeción raspberry-robot y presionar hasta escuchar un click. En caso de escoger la configuración horizontal, debe quedar como muestra la figura 3.21.

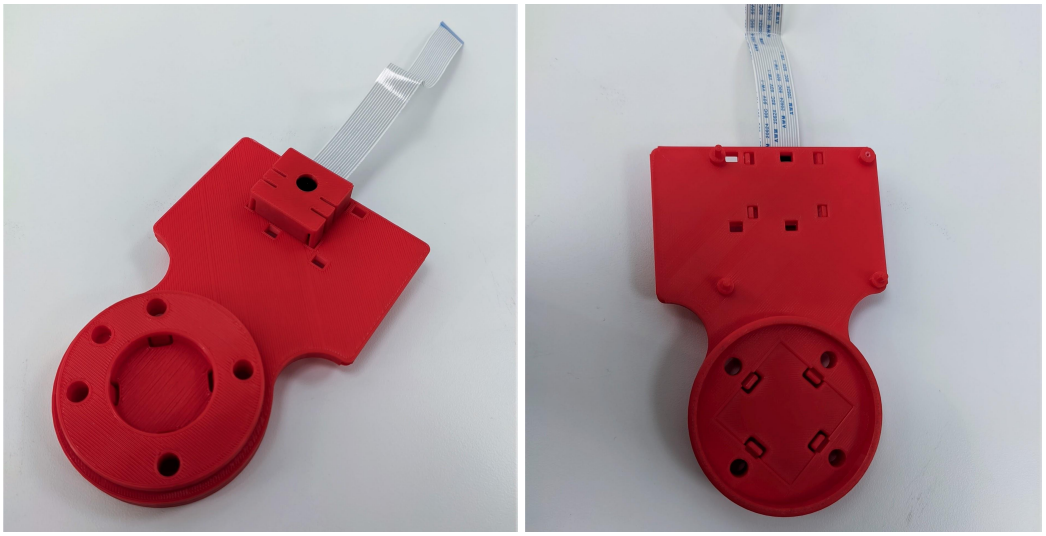


Figura 3.21. Tercer paso del montaje: encajar la sujeción de la cámara en la sujeción raspberry-robot

Para el cuarto paso simplemente hay que colocar la Raspberry Pi en la sujeción raspberry-robot, asegurándose de que los pilares de alineación queden insertados en los agujeros de la placa. Una vez hecho debería quedar como muestra la figura 3.22.

Figura 3.22. Cuarto paso del montaje: colocar la Raspberry Pi en la sujeción raspberry-robot

El siguiente paso consiste en conectar el cable CSI de la cámara a su correspondiente puerto en la Raspberry Pi. Pero antes de hacerlo hay que pasarlo por la ventana de la tapa, de manera que quede como en la figura 3.23.

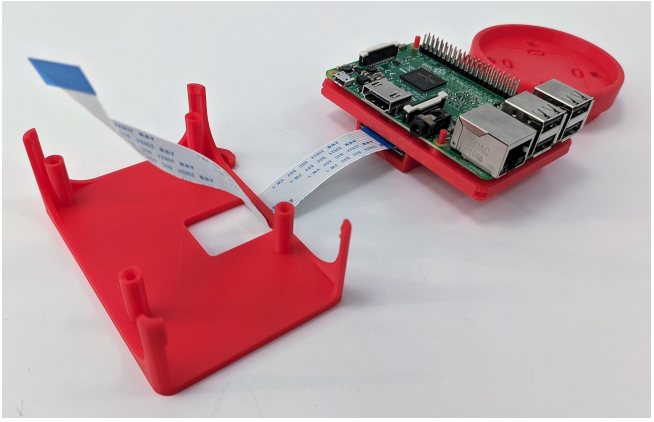


Figura 3.23. Quinto paso del montaje: pasar el cable CSI a través de la ventana de la tapa

Una vez pasado el cable por la ventana de la tapa ya se puede conectar a la placa tal como muestra la figura 3.24.

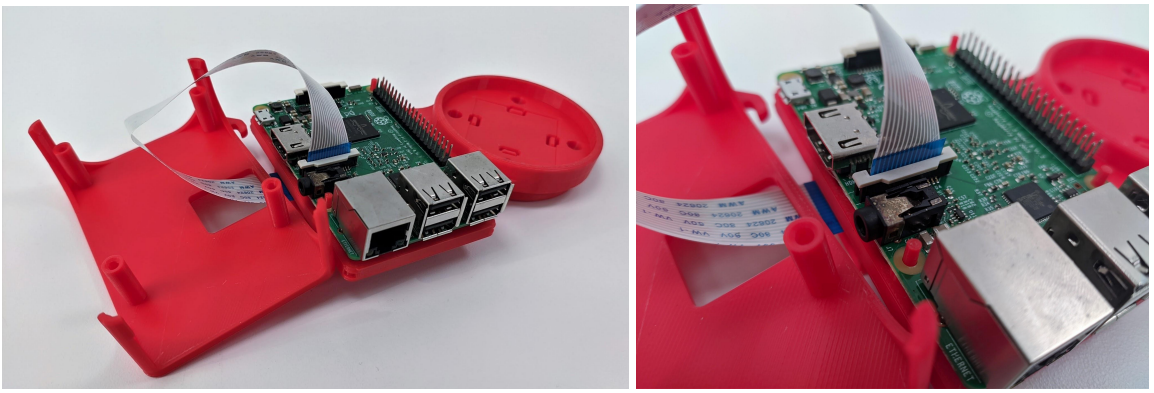


Figura 3.24. Sexto paso del montaje: conectar el cable CSI a su puerto de la Raspberry Pi

Una vez conectada la cámara ya se puede girar la tapa para colocarla en su sitio. Para encajar la tapa a la sujeción raspberry-robot hay que alinear las cuatro columnas de la tapa con los cuatro pilares que sujetan la Raspberry Pi, una vez alineados solo hay que presionar en cada una de las cuatro esquinas hasta escuchar el click del mecanismo, quedando el conjunto tal como se muestra en la figura 3.25.

Figura 3.25. Séptimo paso del montaje: encajar la tapa de protección

Una vez encajada la tapa, solo queda conectar el cable de alimentación, el cual hay que insertar primero en su sujeción ejerciendo una ligera presión hasta quedar como muestra la figura 3.26. Este último paso también se puede realizar después de montar el sistema sobre el robot.

Figura 3.26. Octavo paso del montaje: encajar el cable de alimentación en su sujeción

Una vez seguidos estos pasos ya se tiene el sistema completamente ensamblado y listo para montarse sobre el robot UR3e.

Figura 3.27. Sistema de visión ensamblado

Para instalar el sistema entre el robot y un elemento terminal primero hay que mover la muñeca del robot de manera que quede orientada verticalmente hacia arriba; este paso no es imprescindible, pero resulta más cómodo para realizar los pasos que siguen. A continuación hay que encajar el sistema de visión a la muñeca del robot haciendo uso del reborde, que debe quedar recubriendo la muñeca. Una vez encajado hay que girarlo a la orientación deseada, asegurándose de que los cuatro agujeros quedan alineados con las roscas de la muñeca del robot, como se observa en la figura 3.28.

Figura 3.28. Primer paso de la instalación: encajar el sistema de visión a la muñeca del robot

El siguiente paso es encajar el elemento terminal, en este caso una pinza, al acople del sistema de visión. Distintos elementos terminales tienen diferentes sistemas de acople, en este caso la pinza usa solamente el hueco central del acople (figura 3.29). Para encajarla simplemente hay que ejercer presión hasta conseguir su inserción, asegurándose otra vez de alinear los agujeros de los tornillos. Si las tolerancias son muy ajustadas igual cuesta más de insertar hasta el fondo, en ese caso se acabará de apretar con la ayuda de los tornillos.

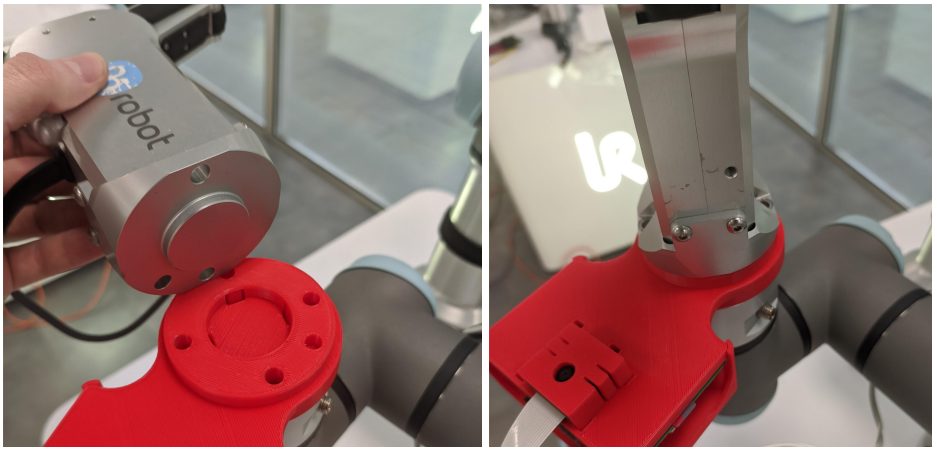


Figura 3.29. Segundo paso de la instalación: encajar el elemento terminal al acople

Una vez encajado y alineado el elemento terminal, se ha de atornillar todo con unos tornillos M6 de la longitud adecuada, en este caso la pinza admite el uso de solamente dos tornillos. Hecho esto ya se puede conectar el elemento terminal al robot, en este caso haciendo uso conector de la muñeca. Realizados estos pasos la instalación queda como en la figura 3.30.

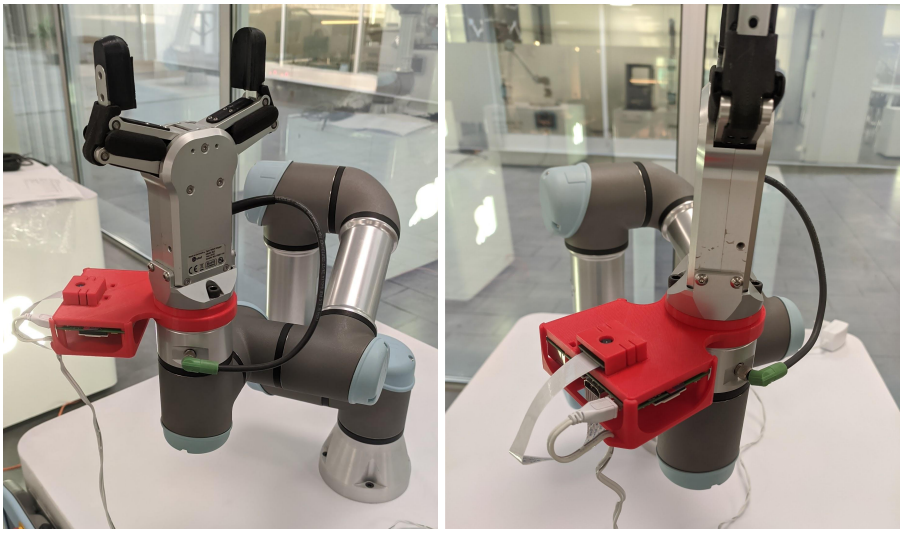


Figura 3.30. Instalación robot-cámara-pinza

Como paso adicional, se recomienda sujetar el cable de alimentación de la Raspberry Pi al brazo del robot para que no quede colgando. En este caso se usan unas tiras de velcro, quedando como se observa en la figura 3.31. Si el elemento terminal también tuviese cables o tuberías neumáticas se podría amarrar todo con las mismas tiras de velcro.

Figura 3.31. Sujeción del cable de alimentación al brazo del robot

4. Sistema de visión

En este capítulo se presenta el sistema de visión por computador desarrollado para el robot, basado en la librería OpenCV. Se aborda la calibración de la cámara para corregir la distorsión de la imagen y calcular la posición del plano de trabajo, el procesado de la imagen para detectar objetos y el cálculo de la posición del objeto respecto al robot.

4.1. OpenCV

OpenCV (Open Source Computer Vision Library) es una librería de software de código abierto diseñada para procesamiento de imágenes y visión artificial. Proporciona herramientas para diversas tareas como detección y seguimiento de objetos, reconstrucción 3D, segmentación de imágenes, calibración de cámaras, entre muchas otras. Es compatible con varios lenguajes de programación, como C++, Python, Java y MATLAB; en este proyecto concreto se usa la versión para Python 3.

Esta librería contiene la mayoría de las funciones necesarias para realizar las tareas requeridas de nuestro sistema de visión: calibración óptica de la cámara, calibración geométrica del plano de trabajo, procesado de la imagen (binarizado, filtrado, detección de contornos y sus propiedades) y cálculo de la posición del objeto. Es este motivo, junto a su naturaleza de código abierto, por lo que se ha escogido la librería OpenCV como la base de este sistema de visión.

Su instalación en la Raspberry Pi se realiza de forma sencilla ejecutando el siguiente comando en una terminal: sudo apt-get install python-opencv. Cabe recordar que es necesario tener Python 3 previamente instalado.

4.2. Calibración del sistema

Es esencial calibrar óptica y geométricamente la cámara antes de utilizarla en el sistema de visión, ya que esto permite corregir las distorsiones ópticas que pueden afectar la imagen capturada y obtener una representación fiel de la escena. Además, la calibración geométrica permite obtener la relación entre los píxeles de la imagen y su correspondiente posición en el mundo real, lo que es fundamental para poder medir distancias y posiciones con precisión.

4.2.1. Calibración óptica

La calibración óptica de la cámara tiene como objetivo corregir las posibles distorsiones de la imagen causadas por la lente. Existen dos tipos principales de distorsiones ópticas: la distorsión radial y la distorsión tangencial.

La distorsión radial se produce cuando los rayos de luz que pasan a través de los bordes de la lente se curvan más que los que pasan a través del centro, lo que provoca que los objetos aparezcan deformados en la imagen. Esta distorsión puede ser de tipo barril, cuando los objetos se curvan hacia el exterior, o de tipo cojín, cuando se curvan hacia el interior.

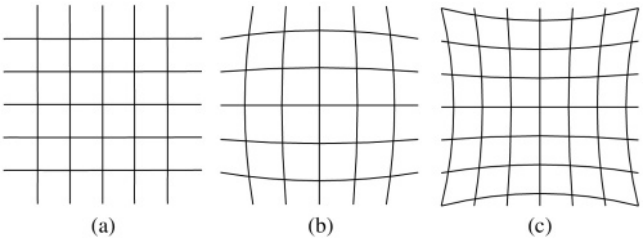


Figura 4.1. Distorsión radial: imagen de entrada (a), distorsión tipo barril (b) y distorsión tipo cojín (c) [6]

La distorsión tangencial, por otro lado, se produce cuando la lente no está perfectamente paralela o centrada al plano de la imagen. Esto provoca que los objetos parezcan inclinados o sesgados en la imagen, tal como representa la imagen 4.2.

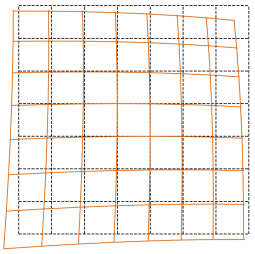


Figura 4.2. Distorsión tangencial [7]

Estas distorsiones se ven definidas por cinco coeficientes de distorsión, los cuales se calculan mediante la calibración óptica y pueden ser usados para corregir dichas distorsiones. Aparte de estos coeficientes, la calibración también calcula los parámetros intrínsecos de la cámara (distancia focal y centro óptico), necesarios para la corrección. Dado que estos coeficientes y parámetros son únicos para cada cámara sólo es necesario calcularlos una vez mediante calibración óptica, una vez obtenidos se pueden reutilizar para cualquier proyecto que use esa misma cámara sin necesidad de volver a calibrarla.

Para encontrar estos parámetros con OpenCV primero es necesario capturar varias imágenes (se recomiendan mínimo diez) de un patrón de calibración conocido, en este caso de un tablero de ajedrez de 9x6 (figura 4.3), desde diferentes ángulos y posiciones.

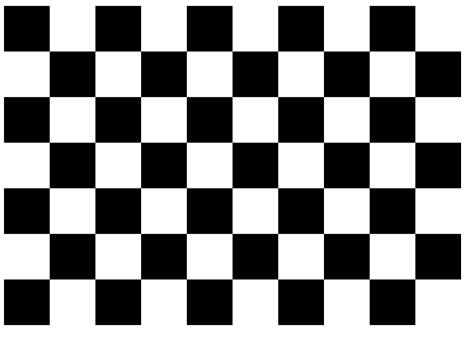


Figura 4.3. Patrón de tablero ajedrez de 9x6 [8]

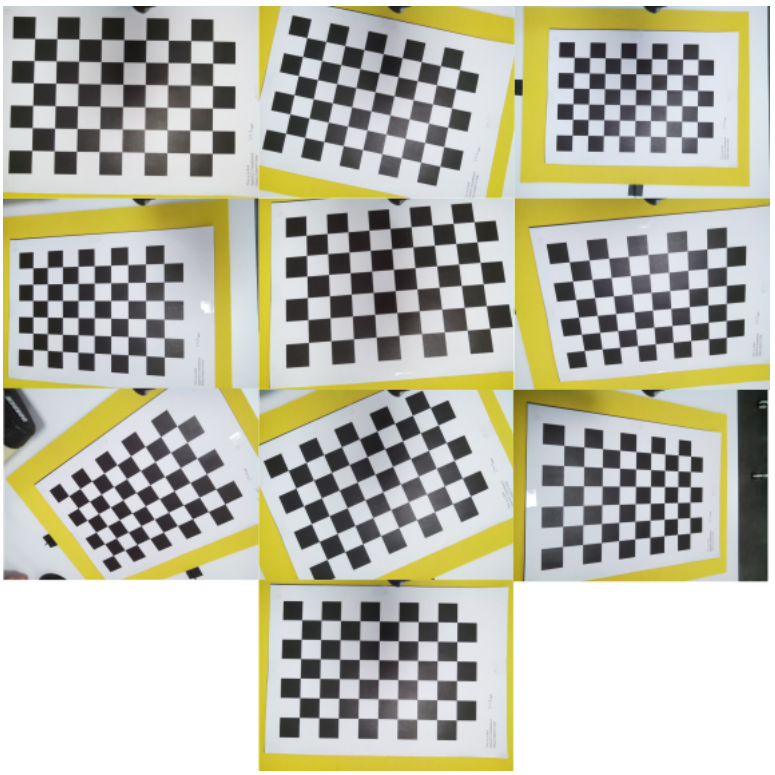


Figura 4.4. Capturas del patrón de calibración desde distintas posiciones

Para capturar dichas imágenes se moverá manualmente el robot (usando el modo “freedrive”) con la cámara a distintas posiciones, desde donde se hará una captura y se mostrará en pantalla. En cada imagen se utiliza la función “cv.findChessboardCorners()” [9] para detectar las esquinas del patrón, si no las encuentra la captura no es válida y se muestra en escala de grises con texto rojo

para informar al operario que debe repetirla desde otra posición. En caso de sí encontrarlas se muestra la imagen a color y con las esquinas marcadas para que el operario pueda verificar que sea correcta y pasar a la siguiente captura.

Figura 4.5. Captura correcta (izquierda) y captura incorrecta (derecha) del patrón de calibración

Una vez encontradas las esquinas se utiliza la función “cv.cornerSubPix()” [9] para refinar sus coordenadas en la imagen y aumentar la precisión. Las posiciones relativas de estas esquinas entre sí también son conocidas (cuadrícula de 9x6). Con ambos conjuntos de datos se pueden calcular los coeficientes de distorsión usando la función “cv.calibrateCamera()” [9], que además también calcula los parámetros intrínsecos de la cámara mencionados anteriormente. Una vez obtenidos estos datos se usa la función “numpy.savez()” para guardarlos en un archivo “calibration.npz”, de esta manera se pueden cargar desde cualquier otro programa que requiera usarlos.

Con estos parámetros de calibración es posible visualizar las imágenes con la distorsión corregida. Sin embargo con la cámara que se usa en este proyecto apenas se aprecia la diferencia a simple vista, ya que la distorsión es muy sutil. La verdadera utilidad de esta corrección reside en aumentar la precisión a la hora de calcular las coordenadas del plano de trabajo y cualquier objeto sobre él.

En el anexo C.1 se puede encontrar el código Python comentado que implementa este proceso de calibración óptica al completo. El mismo código también implementa la calibración geométrica que se explica en el siguiente apartado.

4.2.2. Calibración geométrica

Dado que en este proyecto se dispone solamente de una cámara, sin ningún otro sensor, no es posible obtener datos de profundidad de la escena para saber la posición de un objeto en el espacio 3D. Por ello se ha de implementar un proceso de calibración geométrica que permita obtener la relación entre los píxeles de la imagen y su correspondiente posición en el mundo real. Esta relación es esencial para poder calcular posiciones con precisión de manera que el robot las pueda alcanzar de forma fiable.

La calibración geométrica en este proyecto consiste en calcular la posición del plano de trabajo respecto al robot ("work plane to base", w2b) para luego proyectar los píxeles de la imagen sobre dicho plano, tal como muestra la figura 4.6. De esta manera se obtiene una relación entre un píxel de la cámara y la posición del punto que dicho píxel representa respecto al sistema de coordenadas del robot. Esta calibración, a diferencia de la óptica, no se puede reutilizar para varios proyectos, ya que cada plano de trabajo nuevo requiere de su propia calibración.

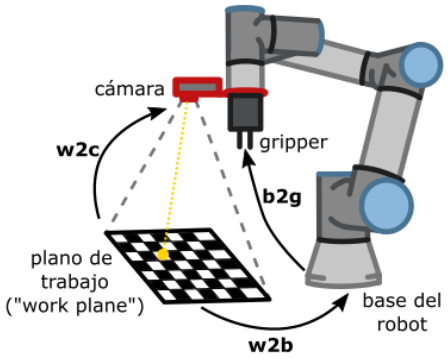


Figura 4.6. Representación de la calibración geométrica

Para calcular la posición del plano de trabajo respecto al robot (w2b) se hará uso de la función "cv.calibrateRobotWorldHandEye()", la cual requiere de los siguientes cuatro parámetros de entrada [9]:

- **Rw2c**: Vector de rotación extraído de la matriz homogénea que transforma un punto expresado en el sistema de referencia del plano de trabajo ("work plane") al sistema de referencia de la cámara (w2c).
- **Tw2c**: Vector de traslación extraído de la matriz homogénea que transforma un punto expresado en el sistema de referencia del plano de trabajo ("work plane") al sistema de referencia de la cámara (w2c).

- **Rb2g**: Vector de rotación extraído de la matriz homogénea que transforma un punto expresado en el sistema de referencia de la base del robot al sistema de referencia de la pinza o "gripper" (b2g).
- **Tb2g**: Vector de traslación extraído de la matriz homogénea que transforma un punto expresado en el sistema de referencia de la base del robot al sistema de referencia de la pinza o "gripper" (b2g).

Ya que los vectores de rotación y traslación se obtienen de forma conjunta, los cuatro parámetros se pueden simplificar en dos transformaciones: w2c ("work plane to cámara") y b2g ("base to gripper"). Para la calibración se necesitan al menos diez de estas transformaciones obtenidas desde diferentes puntos de captura.

Para obtener estas transformaciones se necesitarán diez capturas del patrón de tablero de ajedrez, las cuales se pueden obtener durante la calibración óptica o capturarse independientemente. Es importante que dicho patrón se coloque en el plano de trabajo que se quiera utilizar en la aplicación, ya que se acabarán calculando las coordenadas de éste respecto a la base del robot. En cada captura se leerá la posición del TCP ("Tool Center Point") del robot mediante comunicación RTDE (explicada más adelante en el capítulo 7). Estas posiciones representan transformaciones g2b ("gripper to base"), por lo que hay que realizar la conversión a transformaciones b2g usando el método explicado en el anexo A. Hecho esto ya se tendrían los parámetros de la transformación b2g (Tb2g y Rb2g).

El proceso de obtención de las transformaciones w2c es similar al de la calibración óptica. Primero, para cada captura se han de hallar y refinar las esquinas del tablero de ajedrez en la imagen. También se necesitan las posiciones relativas de estas esquinas entre sí (cuadrícula de 9x6). Estos dos conjuntos de datos se introducen esta vez en la función "cv.solvePnP()" [9], la cual también necesita los parámetros de calibración óptica obtenidos anteriormente. Esta función calcula directamente los vectores de traslación y rotación de la transformación w2c (Tw2c y Rw2c), en otras palabras proporciona la posición del plano de trabajo relativa a la cámara. Cabe destacar que el vector de traslación por defecto tiene unidades iguales a la longitud del lado de un cuadrado del tablero de ajedrez, para pasarlo a metros hay que multiplicarlo por el valor en metros de dicha longitud (24.5 mm en este caso), el cual se hubo de medir a mano con una regla de precisión.

Con los parámetros ya obtenidos se puede usar la función "cv.calibrateRobotWorldHandEye()", la cual calculará la matriz de rotación y el vector de traslación propios de la matriz homogénea que transforma un punto expresado en el sistema de referencia de la base de robot al sistema de referencia del plano de trabajo (b2w). Dado que necesitamos la transformación inversa, del plano de trabajo a la base del robot, volvemos a utilizar el método del anexo X para obtener los vectores

de rotación y traslación de la transformación w2b (Tw2b y Rw2b). Estos vectores se guardan para su posterior uso a la hora de calcular la posición de un objeto respecto a la base del robot.

Ahora que se tiene la posición del plano de trabajo respecto a la base del robot (w2b) solo falta proyectar los píxeles de la cámara sobre dicho plano. Para ello se utilizará una matriz de homografía, la cual sirve para representar una transformación perspectiva entre dos planos. En otras palabras, la matriz de homografía se utiliza para transformar los píxeles de la imagen a los puntos correspondientes en el plano de trabajo (figura 4.7).

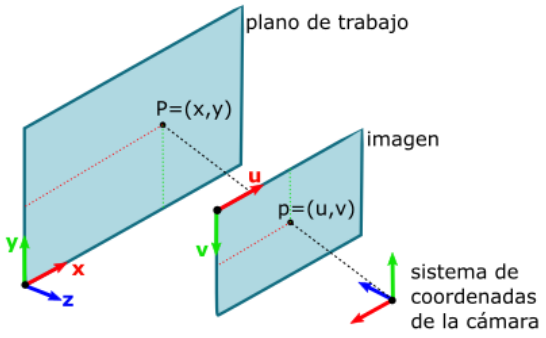


Figura 4.7. Transformación de un píxel de la imagen (p) a un punto del plano de trabajo (P)

Para obtener esta matriz de homografía se necesita una vez más una captura del patrón de tablero de ajedrez, esta vez tomada desde la posición de captura que se desee utilizar en la aplicación. Para no tener que volver a tomarla se usa la primera de las diez capturas usadas anteriormente, por lo que hay que asegurarse que esta se haga desde la posición de captura deseada. En esta imagen se detectan las esquinas del tablero como se hizo anteriormente. También hay que volver a usar las posiciones relativas de las esquinas entre sí, esta vez teniendo en cuenta la medida real de los cuadrados (24.5 mm). Una vez conseguidos estos datos, se pueden introducir en la función "cv.findHomography()" [9] para que calcule la matriz de homografía que permite transformar los píxeles de una imagen tomada desde la posición de captura a los puntos correspondientes en el plano de trabajo. Una vez más, guardamos esta matriz para su posterior uso.

Con los vectores de rotación y traslación de la transformación w2b y la matriz de homografía obtenidos, la calibración geométrica está completa. Se pueden usar estos parámetros en conjunto para calcular la posición de un objeto representado en una imagen respecto al sistema de coordenadas de la base del robot. Primero se proyecta el objeto de la imagen sobre el plano de trabajo, obteniendo unas coordenadas en el sistema de referencia de dicho plano, para luego transformar esas coordenadas al sistema de referencia del robot. Este proceso se verá más en detalle en el capítulo 5 de este documento.

Como se comentó en el apartado anterior, en el anexo C.1 se encuentra el código Python comentado que implementa este proceso de calibración geométrica al completo.

4.3. Procesado de la imagen

El procesado de imágenes es esencial en una aplicación de visión por computador, ya que permite extraer información útil de las imágenes para su posterior análisis y toma de decisiones. Para este proyecto concreto el procesado debe conseguir reconocer distintos objetos y obtener su posición en la imagen.

Para realizar el procesado de una imagen se ha de seguir la siguiente secuencia de pasos: binarizar la imagen, operar morfológicamente sobre la imagen binarizada, detectar objetos (contornos) en la imagen y calcular sus características. Es por ello que este capítulo se divide en apartados que tratan cada uno de estos pasos.

Se explicarán todas las diferentes herramientas de procesado que se han implementado en este proyecto, pero eso no quiere decir que se utilicen todas en cualquier aplicación de visión (a diferencia de las herramientas de calibración del apartado anterior). El usuario ha de elegir por su cuenta cuales son las herramientas adecuadas para su aplicación concreta, para asistirle en dicha elección se explicará la utilidad y posibles inconvenientes de cada herramienta.

4.3.1. Binarizado

El binarizado de una imagen consiste en convertir una imagen a color o en escala de grises a una imagen binaria en la que cada píxel toma un valor binario: 0 (negro) o 1 (blanco). Se utiliza para simplificar la imagen y reducir su complejidad, lo que facilita su posterior análisis.

En este proyecto han implementado dos métodos de binarizado: el binarizado clásico a partir de una imagen en escala de grises y el binarizado por color a partir de una imagen RGB.

El binarizado clásico consiste en asignar a cada píxel el color blanco o negro dependiendo de si su valor de intensidad de gris se encuentra por encima o por debajo de cierto umbral. Para escoger este umbral se suele utilizar el histograma de la imagen. El histograma de una imagen es una gráfica en la que el eje horizontal representa los posibles valores de intensidad de gris que puede tomar un píxel (0 a 255 si es una imagen de 8 bits) y el eje vertical el número de píxeles en la imagen que toman ese valor concreto (figura 4.8). Observando el histograma, un umbral de binarizado adecuado sería el valor correspondiente al valle más pronunciado (en este ejemplo

ronda el valor 140). Se escoge el valor valle ya que és que separa el histograma en los dos conjuntos más diferenciados.

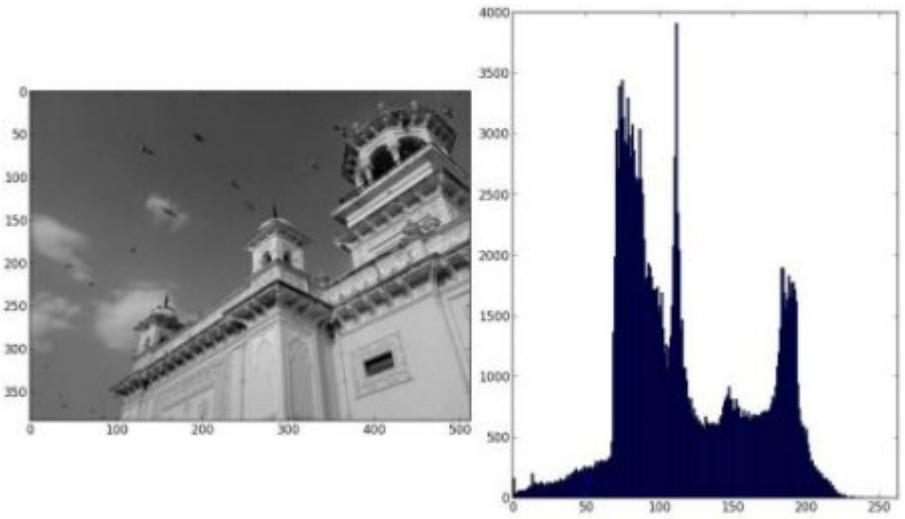


Figura 4.8. Imagen en escala de grises (izquierda) y su histograma [10]

Si bien este valor umbral se puede escoger a mano, también existe la posibilidad de usar el algoritmo de Otsu para encontrar el umbral óptimo que separa los píxeles de la imagen en dos clases. Este algoritmo utiliza el histograma para analizar la varianza entre las dos clases (dos mitades del histograma) y busca el umbral que minimiza la varianza dentro de cada clase y maximiza la varianza entre las clases.

Para binarizar una imagen de manera clásica con OpenCV se utiliza la función “cv.threshold()”, a la cual se le puede introducir un valor umbral concreto o indicar con el parámetro “cv.THRESH_OTSU” que utilice el algorítmo de Otsu [10].

Si la imagen en escala de grises presenta ruido este se puede ver reflejado en la binarización en forma de píxeles blancos en zonas negras y píxeles negros en zonas blancas, es lo que se conoce como el efecto sal y pimienta. Para mitigar sus efectos, una posible solución es aplicar un filtro gaussiano a la imagen original para suavizar dicho ruido antes de binarizarla. El filtro aplica una función de convolución con una distribución de probabilidad gaussiana a cada píxel de la imagen, lo que suaviza los cambios bruscos de intensidad, reduciendo el ruido y los detalles finos. Con OpenCV se puede aplicar un filtro gaussiano utilizando la función “cv.GaussianBlur()” [10].

La figura 4.9 muestra una comparativa entre diferentes métodos de binarizado clásico. En el primer caso se aplica una binarización con un valor umbral escogido a mano. En el segundo caso se aplica la binarización mediante el algoritmo de Otsu. En el tercer caso, primero se aplica un

filtro gaussiano y luego se aplica la binarización con Otsu. Se observa cómo el filtrado mejora el resultado reduciendo el efecto sal y pimienta.

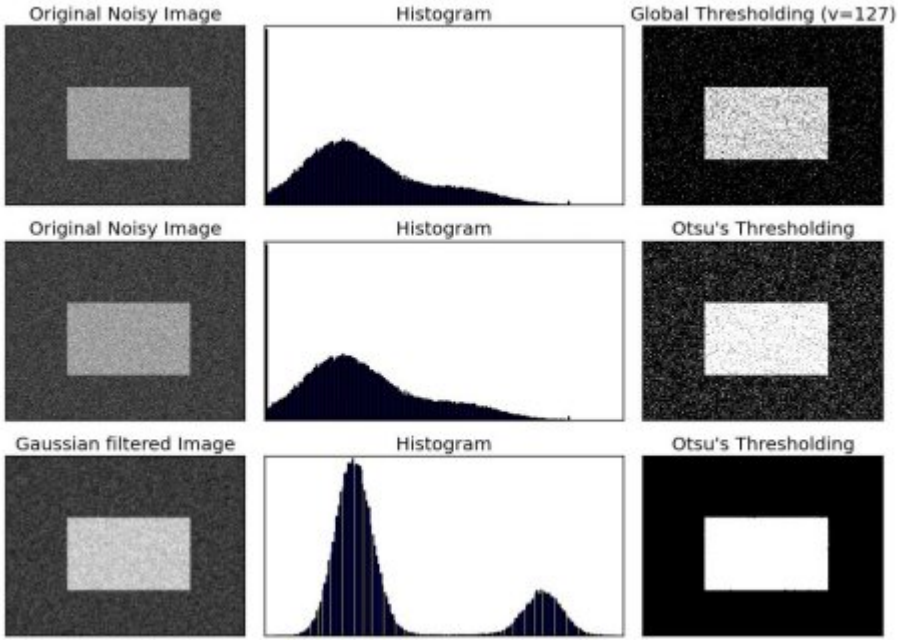


Figura 4.9. Comparativa entre diferentes métodos de binarizado clásico [10]

Por otra parte, el binarizado por color consiste en separar el objeto del fondo a partir de la tonalidad de color de uno de estos. Para utilizar este método primero hay que convertir la imagen RGB (“Red, Green and Blue”) a un formato HSV (“Hue, Saturation and Value”) mediante la función “cv.cvtColor(dst, cv.COLOR_BGR2HSV)”. Una vez en este formato, se puede usar el color del fondo o del objeto para establecer un rango que incluya las variaciones de tonalidad (“hue”) de ese color. Si las tonalidades del fondo y el objeto son similares (por ejemplo un verde claro sobre un verde oscuro) también se pueden usar los valores de saturación (“saturation”) y brillo (“value”) para acabar de acotar el rango. Establecido ese rango se utiliza la función “cv.inRange()” [10] para binarizar la imagen HSV, poniendo en blanco todos los píxeles dentro del rango y en negro todos los píxeles que estén fuera, separando objeto y fondo. Si se usa la tonalidad del fondo para el rango este será el que quede en blanco, para que sea el objeto el que esté en blanco se puede invertir la imagen binarizada usando la función “cv.bitwise_not()” [10].

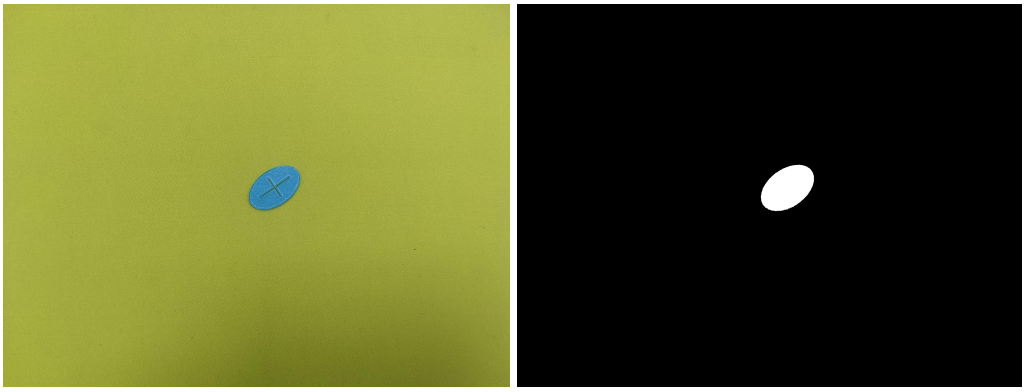


Figura 4.10. Binarizado por color: imagen de entrada (izquierda) e imagen binarizada (derecha)

Ambos tipos de binarizado tienen sus ventajas e inconvenientes según la escena que se esté capturando. El binarizado clásico es robusto en escenas con objetos oscuros sobre fondo claro y viceversa, dado que el blanco y el negro son colores muy comunes este método resulta muy útil simpre que se usen en contraste. Sin embargo, a este método le cuesta diferenciar entre dos colores con una intensidad similar, a pesar de que tengan una tonalidad diferente. Además, el binarizado clásico es bastante sensible a sombras que oscurezcan parcialmente la escena.

Por otra parte el binarizado por color es muy fiable en escenas donde se tenga un fondo y/u objeto con colores vívidos que contrasten, por ejemplo: azul sobre amarillo, rojo sobre blanco, verde sobre negro, negro sobre marillo, etc. Al basarse en la tonalidad de color tambien es más robusto ante sombras en la escena, ya que estas afectan principalmente al brillo o intensidad de un color y no su tonalidad. Su principal desventaja es que falla en cualquier escena en la que no haya color, es decir, con objetos y fondos blancos, negros o grises, ya que no hay una tonalidad diferenciada con la se pueda crear un rango.

Se puede concluir que ambos métodos de binarizado se complementan, siendo uno más robusto en situaciones donde el otro falla. A continuación se muestran una serie de escenas donde se han usado ambos métodos, dejando claras las ventajas y debilidades de cada uno.

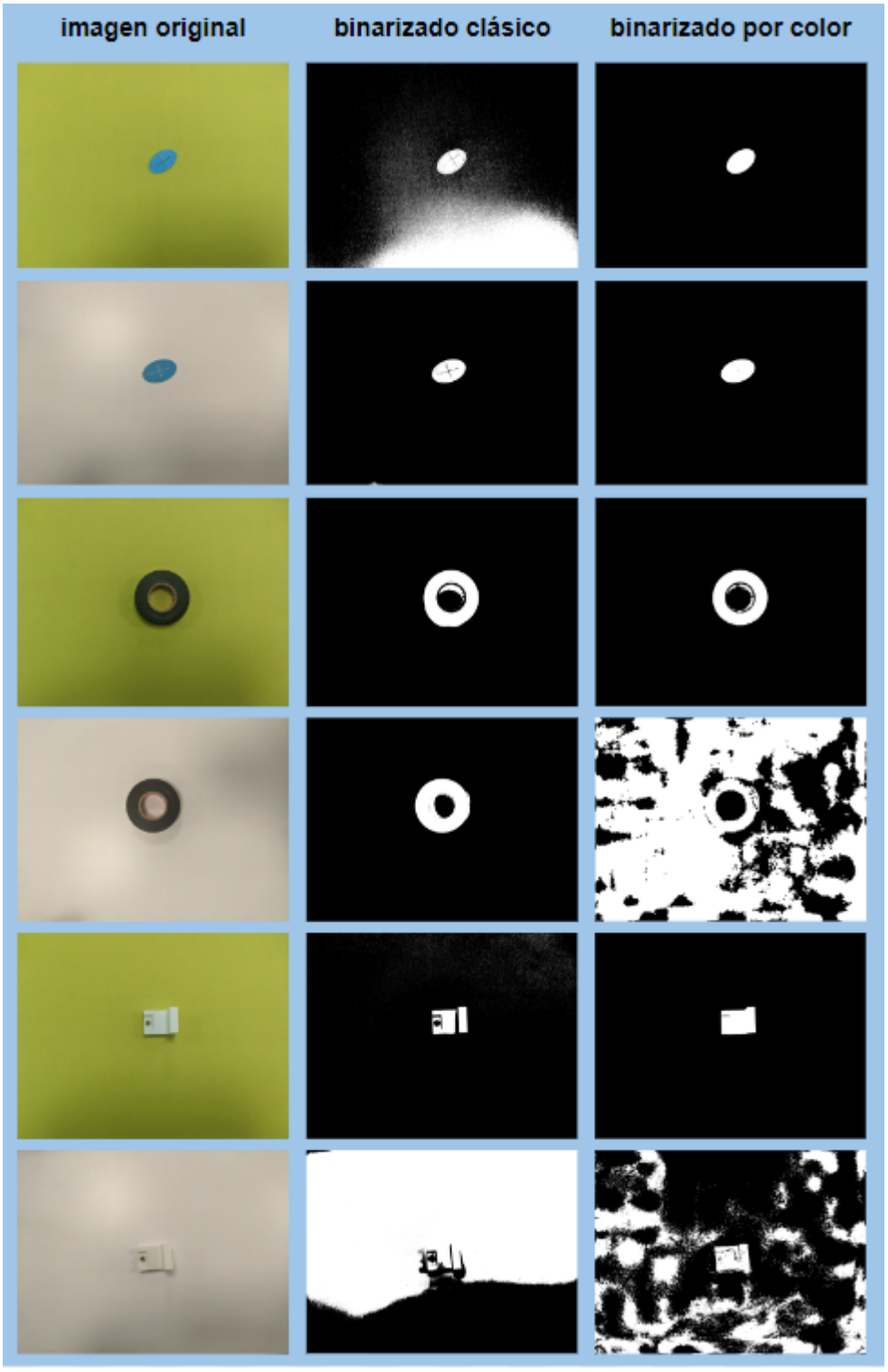


Figura 4.11. Comparativa entre binarizado clásico (centro) y binarizado por color (derecha)

Observando la figura 4.11 se puede apreciar que en la última escena ninguno de los métodos consigue binarizar la imagen de forma correcta. Esto se debe a que se trata de una pieza blanca sobre un fondo blanco, por lo que no hay contraste de intensidad ni de tonalidad. Es por esto que es tan importante planificar la escena de una aplicación, asegurándose que el objeto contraste bien con el fondo y que se tenga una iluminación que minimice las sombras y los reflejos.

4.3.2. Operaciones morfológicas

El proceso de binarización no es perfecto, es común que deje ruido y artefactos no deseados. Para acabar de retocar la imagen binarizada y eliminar esos desperfectos se hace uso de operaciones morfológicas, también conocidas como transformaciones morfológicas. Estas operaciones manipulan la forma y la estructura de los objetos en una imagen (se consideran objetos las partes de la imagen binarizada que queden en color blanco).

Para transformar la imagen estas operaciones hacen uso de elemento estructural, una especie de plantilla de forma geométrica que se va deslizando por la imagen pasando por todos y cada uno de los píxeles. El valor de cada píxel (0 o 1, negro o blanco) se decide segun la operación elegida, la cual tiene en cuenta el resto de píxeles que quedan bajo el elemento estructural. OpenCV cuenta con la función “cv.getStructuringElement()” [10] para crear elementos estructurales con forma rectangular, elíptica o de cruz. Esta función crea una matriz del tamaño que se le indique de píxeles, se ha de tener en cuenta que debe tener un número de filas y columnas impares para mantener una posición central, ya que esta es la posición del píxel sobre el que se realizará la operación. Dentro de la matriz la forma del elemento estructural se representa con unos dejando el resto de la matriz con ceros, tal como aparece en la figura 4.12. Cuando el elemento estructural está sobre un píxel concreto la operación a realizar tiene en cuenta todos los píxeles que quedan bajo un 1 de la matriz.

```
[1, 1, 1, 1, 1]    [0, 0, 1, 0, 0]    [0, 0, 1, 0, 0]
[1, 1, 1, 1, 1]    [1, 1, 1, 1, 1]    [0, 0, 1, 0, 0]
[1, 1, 1, 1, 1]    [1, 1, 1, 1, 1]    [1, 1, 1, 1, 1]
[1, 1, 1, 1, 1]    [1, 1, 1, 1, 1]    [0, 0, 1, 0, 0]
[1, 1, 1, 1, 1]    [0, 0, 1, 0, 0]    [0, 0, 1, 0, 0]
```

Figura 4.12. Elementos estructurales: rectángulo (izquierda), elipse (centro) y cruz (derecha) [10]

Las dos operaciones morfológicas básicas son la erosión y la dilatación, las cuales se pueden combinar para crear las operaciones de cierre y apertura.

En una transformación morfológica de erosión el valor de un píxel será 1 solo si todos los píxeles bajo el elemento estructural tienen valor 1, en caso contrario tomará el valor 0. De esta manera

los píxeles situados en los bordes de un objeto pasarán siempre a ser 0, ya que están en contacto con un píxel 0. El efecto de esta operación aplicada a todos los píxeles de la imagen es, como su nombre indica, erosionar los objetos de la imagen reduciendo su tamaño (figura 4.13). Para aplicar esta operación a una imagen se usa la función "cv.erode()" [10], dando como parámetros la imagen de entrada y el elemento estructural deseado.

Figura 4.13. Operación morfológica de erosión: imagen original (izquierda) e imagen erosionada (derecha) [10]

Esta operación resulta útil para separar objetos. Si dos objetos en una imagen están en contacto la binarización tiende a agruparlos en un solo objeto. Aplicando erosión es posible eliminar los píxeles del punto de contacto, separando los objetos en dos como muestra la figura 4.14. Un inconveniente de esta técnica es que reduce ligeramente el tamaño de ambos objetos

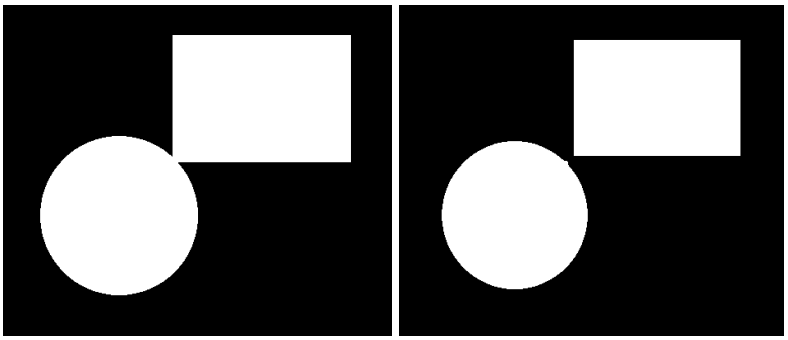


Figura 4.14. Separación de objetos mediante erosión: imagen original (izquierda) e imagen erosionada (derecha)

La operación de dilatación es la opuesta a la erosión, ya que aumenta el tamaño de los objetos (figura 4.15). En esta transformación un píxel toma el valor 1 si al menos uno de los píxeles bajo el elemento estructural tiene valor 1. En otras palabras, solo toma el valor 0 si todos los píxeles bajo el elemento estructural tienen valor 0. Se usa la función "cv.dilate()" para aplicar esta transformación

Figura 4.15. Operación morfológica de erosión: imagen original (izquierda) e imagen erosionada (derecha) [10]

Esta operación se puede usar para juntar dos partes de un mismo objeto que han quedado separadas durante la binarización, aunque provoca que este aumente de tamaño. Para contrarrestar este inconveniente se puede aplicar acto seguido una operación de erosión y así reducir de nuevo el tamaño. Dilatar una imagen para luego erosionarla es lo que se conoce como una operación de cierre, ya que cierra huecos en el objeto. OpenCV cuenta con la función "cv.morphologyEx()" [10], que al pasarle el parámetro "cv.MORPH_CLOSE" junto a la imagen y el elemento estructural aplica directamente la operación de cierre sin necesidad de dilatar y erosionar por separado.

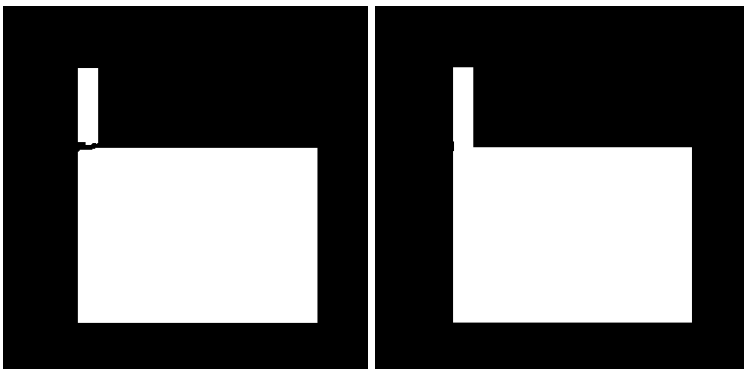


Figura 4.16. Operación morfológica de cierre: imagen original (izquierda) e imagen cerrada (derecha)

A parte de juntar dos partes separadas de un mismo objeto (figura 4.16) también resulta útil para eliminar posibles huecos y puntos negros en el objeto (ruido de pimienta), tal como se puede observar en la figura 4.17.

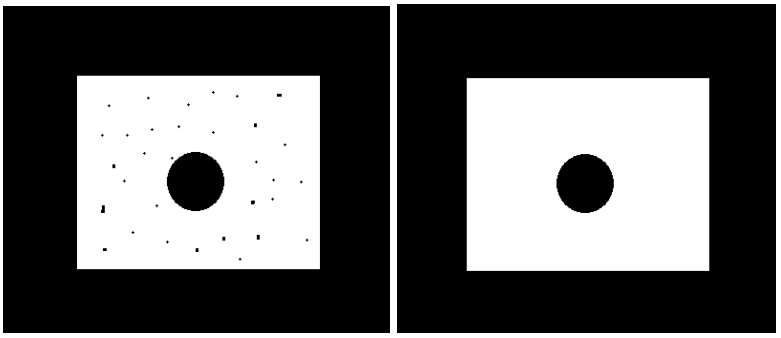


Figura 4.17. Eliminación de ruido mediante cierre: imagen original (izquierda) e imagen cerrada (derecha)

Si se invierte el orden de aplicación, primero erosión y luego dilatación, se obtiene la operación de apertura. Esta sirve para eliminar puntos blancos del fondo (ruido de sal), como se muestra en la figura 4.18.

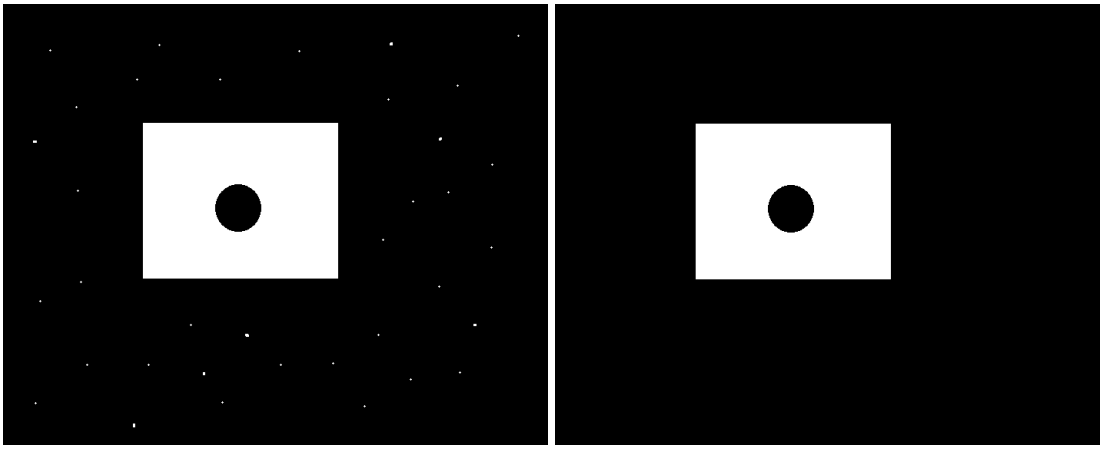


Figura 4.18. Eliminación de ruido mediante apertura: imagen original (izquierda) e imagen abierta (derecha)

Una de las prácticas más comunes es aplicar apertura seguida de cierre, eliminando así el ruido de sal y pimienta además de reparar objetos fragmentados.

4.3.3. Contornos

Ahora que se tiene una imagen binarizada con los defectos corregidos ya se pueden detectar los objetos que contiene. En una imagen binarizada se considera un objeto a una región de píxeles conectados que tienen un valor de 1 (blanco). Una imagen puede tener varios objetos si tiene diferentes regiones separadas entre sí. Este es el motivo por el que es tan importante eliminar el ruido de tipo sal (puntos blancos), ya que se detectarían como objetos independientes, dificultando trabajar con los objetos reales.

La librería OpenCV trata los objetos analizando su contorno, es por esto que para simplificar se referirá a los objetos como contornos, aunque cabe destacar que no son equivalentes exactos (como se explicará más adelante). La función “cv.findContours()" [10] encuentra todos los contornos de una imagen binarizada y los devuelve ordenados en una lista. Cada contorno viene definido por una matriz con las coordenadas en la imagen de los puntos que conforman forman dicho contorno. Para visualizarlos sobre la imagen original existe la función “cv.drawContours()" [10].

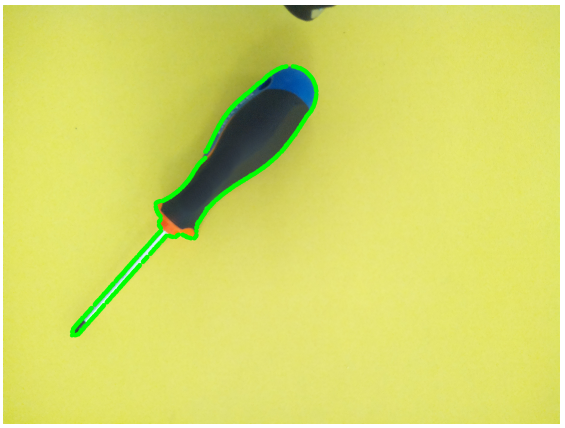


Figura 4.19. Contorno dibujado en verde sobre la imagen original

Una vez obtenido un contorno, OpenCV cuenta con diversidad de funciones que permiten extraer sus características. En este apartado se resumen las de uso más común, en la documentación de OpenCV se explicitan el resto de ellas [10].

La función más básica es “cv.moments()”, que calcula los momentos del contorno. Los momentos de una imagen es un concepto complejo cuya explicación detallada queda fuera del alcance de este proyecto. Simplificando, son valores numéricos de una imagen que sirven para calcular otras características de la misma: forma, tamaño, orientación, etc. El único uso directo que se les dará en este proyecto es el cálculo del centroide de un contorno. Este centroide se puede entender como el centro de masas del contorno y sirve para determinar su posición en la imagen. A continuación se muestra un fragmento de código que calcula ambas componentes (x, y) de dicho centroide:

```
M = cv.moments(cnt) # Calculamos momentos del contorno

cx = int(M['m10']/M['m00']) #Calculamos centroide.x

cy = int(M['m01']/M['m00']) #Calculamos centroide.y
```

Para el resto de características OpenCV cuenta con funciones dedicadas que evitan tener que trabajar con los momentos. Por ejemplo permite calcular el área (número de píxeles) de un contorno con la función “cv.contourArea()” [10], lo que da una idea del tamaño del objeto además de poder usarse para calcular otras características. Respectivamente también se puede calcular su perímetro con la función “cv.arcLength()" [10].

Es posible calcular la envolvente convexa de un contorno con la función “cv.convexHull()" [10]. En un contorno con concavidades su envolvente convexa es la forma que más se le ajusta evitando dichas concavidades. En la figura 4.20 se muestra un contorno cóncavo con su envolvente convexa representada en verde.

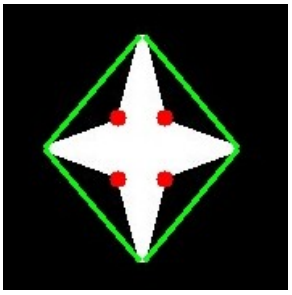


Figura 4.20. Envolvente convexa (verde) de un contorno (blanco) [10]

Si dividimos el área del contorno entre el área de su envolvente convexa se puede calcular lo que se conoce como su solidez. La solidez indica si un contorno tiene concavidades y como de pronunciadas son tomando valores entre 0 y 1, siendo 1 el valor de un contorno completamente convexo y tomando valores menores a medida que aumenta la concavidad. A continuación se muestra el código que realiza este cálculo:

```
area = cv.contourArea(cnt) # Calculamos el área del contorno

hull = cv.convexHull(cnt) # Calculamos su envolvente convexa

hull_area = cv.contourArea(hull) # Calculamos el área de la envolvente

solidez = float(area)/hull_area # Calculamos la solidez
```

También es posible calcular los rectángulos que delimitan un contorno. Existen de dos tipos: rectángulo recto y rectángulo rotado. El primero se obtiene con la función “cv.boundingRect()" [10], este no tiene en cuenta la rotación del contorno y se ve delimitado por sus extremos en ambos ejes (x, y). El rectángulo rotado viene dado por la función “cv.minAreaRect()" [10] y

representa el rectángulo con mínima área capaz de delimitar el contorno, esta vez sí teniendo en cuenta su orientación.

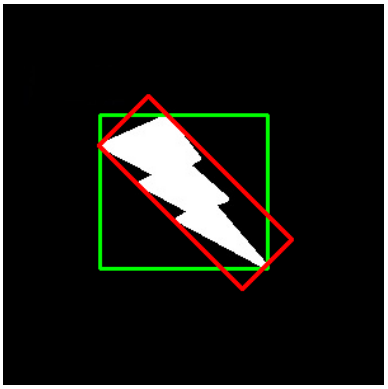


Figura 4.21. Rectángulo recto (verde) y rectángulo rotado (rojo) delimitadores de un contorno [10]

Con el rectángulo rotado es posible calcular la relación de aspecto del contorno dividiendo su anchura entre su altura, lo que resulta útil para distinguir objetos alargados en la imagen. El siguiente fragmento de código muestra este cálculo:

```
centro_x,centro_y,ancho,alto,angulo = cv.minAreaRect(cnt) # Calculamos rectángulo

#Calculamos relación de aspecto dividiendo el lado menor entre el mayor:

if alto>ancho: relacion_aspecto = float(ancho)/alto

else: relacion_aspecto = float(alto)/ancho
```

De manera similar al rectángulo también es posible calcular la elipse que se ajusta mejor a un contorno mediante la función “cv.fitEllipse()”. La principal utilidad de esto es que, entre otros valores, devuelve el ángulo de rotación del eje mayor la elipse, el cual se puede usar para calcular la orientación del objeto (se explicará más en detalle en el capítulo 5).

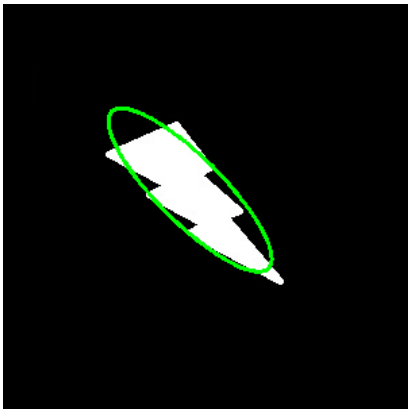


Figura 4.22. Elipse ajustada a un contorno [10]

Estas características se pueden utilizar para distinguir objetos entre sí. Un ejemplo sencillo: si en la aplicación se tienen dos objetos, uno cuadrado y otro alargado, se puede calcular la relación de aspecto de un contorno para determinar si es uno u otro. El siguiente código muestra cómo hacer la distinción de este ejemplo:

```
centro_x,centro_y,ancho,alto,ángulo = cv.minAreaRect(cnt) # Calculamos rectángulo

relacion_aspecto = float(ancho)/alto # Calculamos relación de aspecto

if relacion_aspecto > 0.9 : # si la relación de aspecto es elevada...

        obj = "cuadrado" # se trata del cuadrado

else # en caso contrario...

        obj = "alargado" # se trata del objeto alargado
```

Si se quiere detectar una forma concreta se podrían usar las características mencionadas anteriormente para delimitar unos rangos dentro de los cuales dicha forma siempre esté. Sin embargo OpenCV dispone de la función “cv.matchShapes()" [10] que permite comparar directamente dos contornos devolviendo una métrica de similitud, tomando valor 0 ante dos contornos idénticos y aumentando de valor a medida que aumenta la discrepancia. Si se quiere detectar un objeto, simplemente se tiene que enseñar una vez y guardar su contorno de manera que se puedan comparar futuros contornos con el original; si la métrica de similitud es próxima a 0 se puede asumir que se trata del mismo objeto.

Para calcular la similitud, esta función hace uso de los momentos invariantes de Hu. Se trata de un conjunto de siete momentos del contorno cuya característica principal es que son invariantes a la

traslación, la rotación y el escalado. Es esto lo que permite detectar la misma forma en distintas posiciones. Sin embargo que sean invariantes al escalado puede resultar un inconveniente, ya que si se tienen dos objetos de misma forma pero tamaño diferente (por ejemplo dos discos de distinto diámetro) no serán capaces de distinguirlo. Es por esto que se recomienda complementar la función “cv.matchShapes()" [10] con el cálculo del área del contorno para evitar este error.

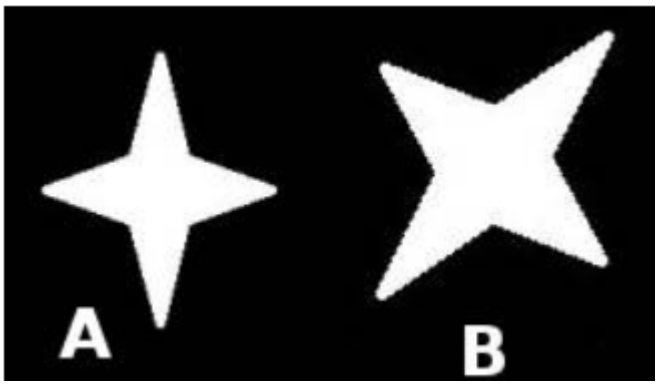


Figura 4.23. Similitud entre contorno A y contorno B: 0.001946 [10]

Otra debilidad de esta función viene dada por la naturaleza de los contornos y es que no es capaz de detectar agujeros en un objeto. Anteriormente se ha mencionado que un objeto y su contorno no son equivalentes exactos, esto se debe a que el contorno principal de un objeto solo tiene en cuenta su borde exterior, obviando los píxeles de su interior. Esto afecta sobretodo a objetos con agujeros, ya que sus contornos exteriores no los tienen en cuenta. De esta manera, si se tienen dos objetos de forma idéntica, uno de ellos con un agujero y el otro sin, la función “cv.matchShapes()" [10] será incapaz de distinguir sus contornos exteriores (figura 4.24).

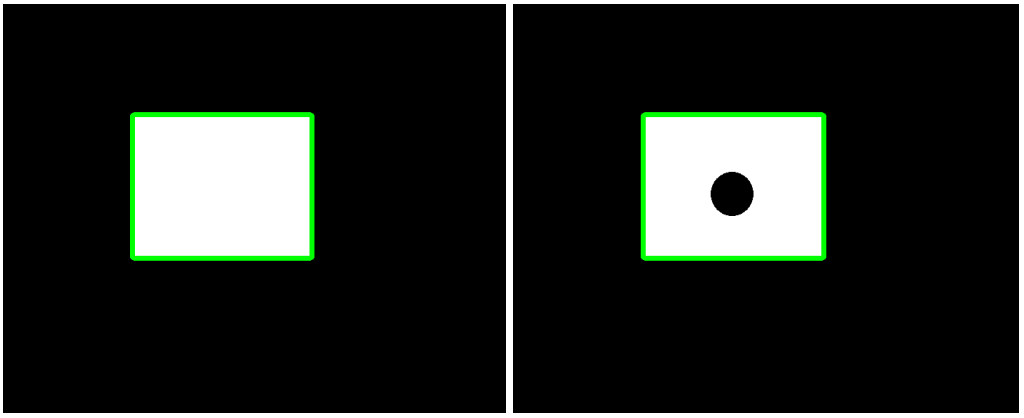


Figura 4.24. Similitud entre ambos contornos es 0, son idénticos

Es por esto que la función “cv.findContours()" [10] no devuelve solo los bordes exteriores de los objetos, sino también los interiores (los agujeros), tal como muestra la figura 4.25. En definitiva,

esta función devuelve los contornos de todo los bordes que haya entre los objetos (en blanco) y el fondo (en negro).

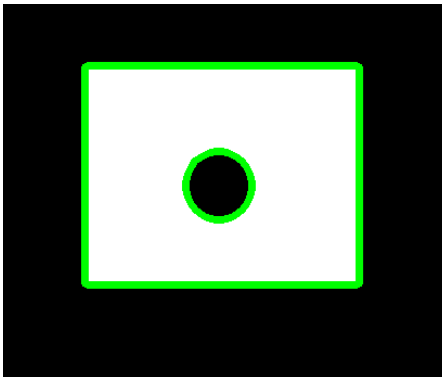


Figura 4.25. Contorno exterior e interior de un mismo objeto

Para poder distinguir un contorno exterior de uno interior “cv.findContours()" [10] proporciona junto al listado de contornos otro listado con su jerarquía. Esta jerarquía indica para cada contorno si se encuentra dentro de otro contorno y/o si tiene otro en su interior. De esta manera es posible saber si un objeto tiene agujeros, comprobando si su contorno exterior tiene algún otro en su interior.

5. Cálculo de posiciones

Una vez procesada la imagen y obtenidos los contornos de los objetos ya es posible usar los parámetros de la calibración geométrica para calcular la posición de dichos objetos en el mundo real. Si se quiere coger un objeto por un punto concreto (posición de "pick") también será necesario enseñar una posición relativa a dicho objeto para que el robot pueda repetirla independientemente de su posicionamiento sobre el plano de trabajo. En este capítulo se explicarán los métodos para obtener ambas posiciones.

Durante el proceso se considerarán cuatro sistemas de referencia: el del objeto, el del plano de trabajo, el de la base del robot y el de la pinza o "gripper"; representados en la figura 5.1. Estos sistemas son importantes ya que el cálculo de posiciones consiste en transformar puntos de uno a otro.

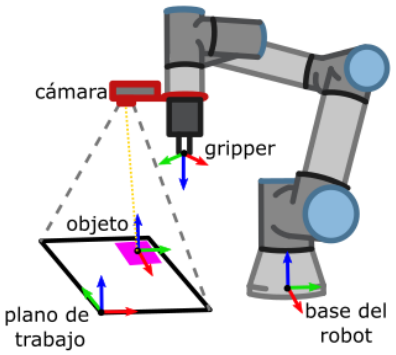


Figura 5.1. Sistemas de referencia usados en el cálculo de posiciones

5.1. Obtención de la posición de un objeto relativa al robot

La obtención de las coordenadas del objeto respecto a la base del robot se realiza en dos pasos. Primero se obtienen las coordenadas del sistema de referencia del plano de trabajo respecto al sistema de referencia de la base del robot ("work plane to base", w2b). En segundo lugar se obtienen las coordenadas del sistema de referencia del propio objeto respecto al sistema de referencia del plano de trabajo ("object to work plane", o2w). Una vez obtenidas ambas coordenadas se pueden convertir en matrices homogéneas (Mw2b y Mo2w) para multiplicarse entre sí y conseguir las coordenadas del objeto respecto a la base del robot (ver anexo A).

Las coordenadas del plano de trabajo respecto a la base del robot ya se obtuvieron durante la calibración geométrica (apartado 4.2.2.) en forma de los vectores de traslación y rotación (Tw2b y

Rw2b). Por tanto solo queda obtener las coordenadas del objeto sobre el plano de trabajo, las cuales tienen una parte de traslación y otra de rotación que se obtienen por separado.

Para obtener la traslación del objeto respecto al plano de trabajo se vuelve a usar un parámetro obtenido durante la calibración geométrica: la matriz de homografía (apartado 4.2.2.). Para poder usarla primero hay que calcular el centroide del contorno del objeto en la imagen usando sus momentos (tal como se explicó en el apartado 4.3.3.). Este centroide sirve como las coordenadas del objeto en el plano de la imagen. Multiplicando la matriz de homografía por estas coordenadas se obtienen los valores traslación del objeto sobre los ejes X e Y del plano de trabajo. Solo faltaría el valor para el eje Z, pero dado que el objeto se encuentra sobre el plano se puede considerar este valor igual a 0. Así ya se tienen las tres componentes del vector de traslación To2w (objeto respecto al plano de trabajo).

La obtención de la rotación del objeto es la parte más compleja de todo el proceso. Durante el proceso se utilizarán los ángulos "Roll, Pitch, Yaw" (rotación sobre los ejes X, Y y Z respectivamente) en grados, ya que resulta más intuitivo trabajar con ellos en vez de con el vector de rotación en radianes.

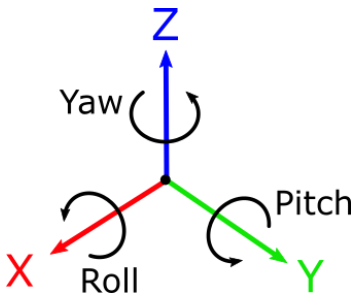


Figura 5.2. Ángulos "Roll, Pitch, Yaw"

Se parte de la base de que el sistema de referencia del objeto se encuentra sobre el plano de trabajo y por ello se considera que su eje Z está alineado con el del sistema de referencia del plano (quedando los ejes X e Y de ambos sobre el mismo plano, como se muestra en la figura 5.1). Esto implica que los ángulos de rotación en los ejes X e Y són nulos (con valor de 0^o), quedando por determinar únicamente el ángulo en el eje Z ("Yaw").

Para calcular este ángulo se empieza usando la función "cv.fitEllipse()" [10] mencionada en el apartado 4.3.3. Esta proporciona el ángulo respecto la vertical del eje mayor de la elipse que mejor se ajusta al contorno del objeto. El problema de este método es que el ángulo viene dado en un rango de 180^o, de manera que se tiene la recta sobre la que se inscribe el objeto pero no la

dirección en la que apunta. Si se usara solamente este método la dirección apuntaría a lados contrarios del objeto dependiendo de su orientación, tal como muestra la figura 5.3.

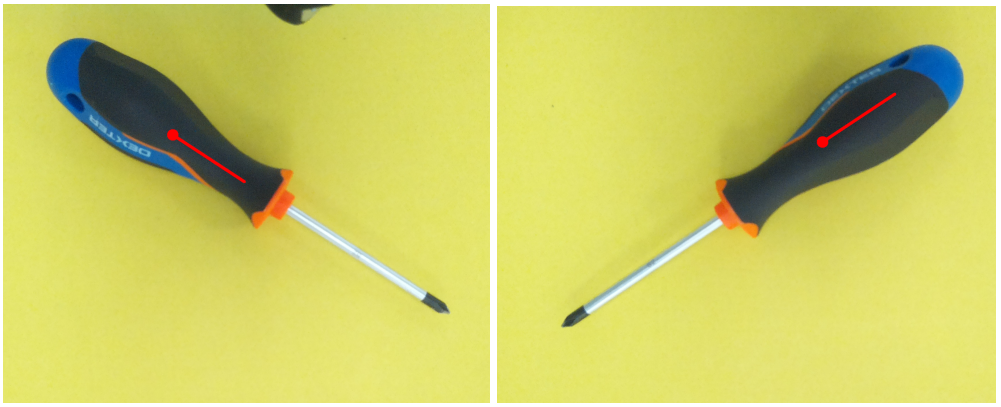


Figura 5.3. Rotación usando solamente el ángulo del eje mayor de la elipse

Para solucionar este problema se hace uso de otra función de OpenCV, “cv.minAreaRect()" [10], que devuelve el rectángulo de mínima área que envuelve el contorno del objeto. Más concretamente se utiliza el centro del rectángulo, ya que este no coincide con el centroide del contorno (figura 5.4). Esta discrepancia se puede utilizar para determinar la dirección en la que apunta el objeto.

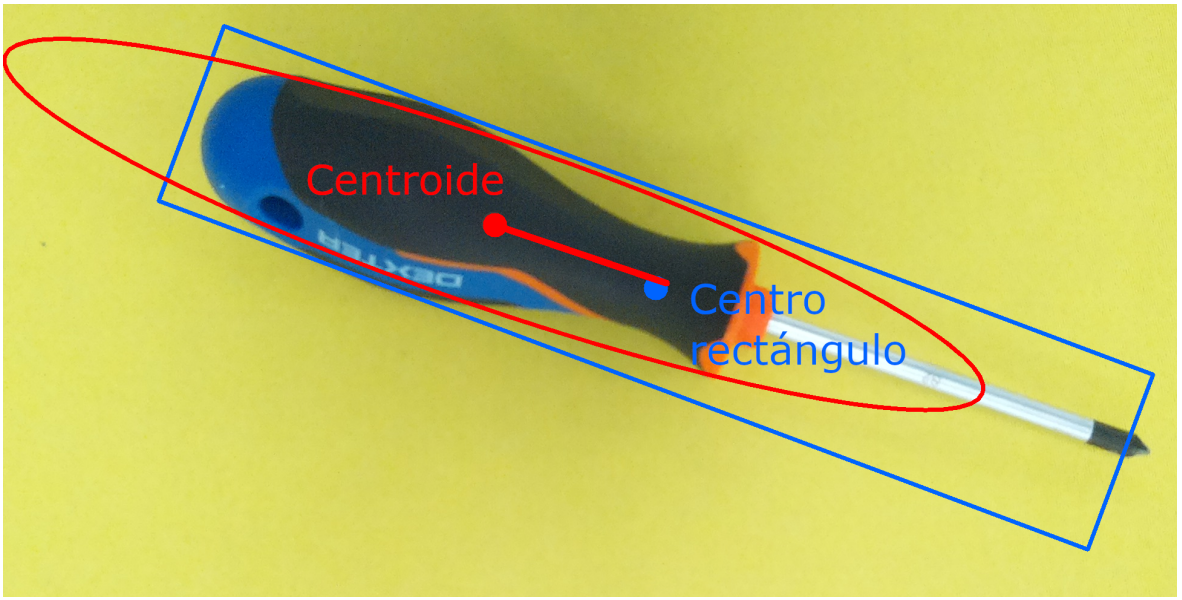


Figura 5.4. Discrepancia entre el centroide y el centro del rectángulo

Para ello primero se calcula la distancia entre el centroide y el centro del rectángulo mediante la ecuación 6.1, siendo (Xc, Yc) y (Xr, Yr) sus respectivas coordenadas en la imagen (en píxeles).

$$d = \sqrt{(Xr - Xc)^2 + (Yr - Yc)^2} \quad \textbf{(Eq. 5.1)}$$

A continuación se calculan dos puntos (punto 1 y punto 2) sobre el eje mayor de la elipse, estando estos a una distancia del centroide igual a la calculada anteriormente y en direcciones opuestas tal como muestra la figura 5.5.

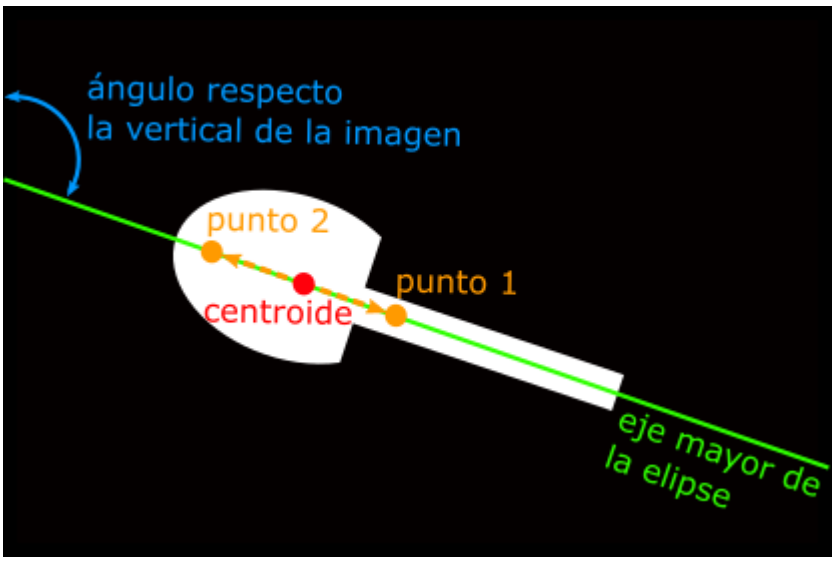


Figura 5.5. Posición de los dos puntos usados para determinar la dirección del objeto

Las coordenadas X e Y de estos puntos se calculan según las ecuaciones 6.2 y 6.3 respectivamente, siendo α un ángulo distinto para cada punto. Para el punto 1, α toma el valor del ángulo del eje mayor de la elipse respecto la vertical de la imagen (obtenido mediante la función "cv.fitEllipse()" [10]). Para el punto 2, α toma ese mismo valor restándole 180º, posicionándose así en dirección opuesta.

$$X_p = Xc + d \cdot sin(\alpha) \quad \textbf{(Ec. 5.2)}$$

$$Y_p = Yc - d \cdot cos(\alpha) \quad \textbf{(Ec. 5.3)}$$

Obtenidos ambos puntos se calcula la distancia de cada uno al centro del rectángulo usando otra vez la ecuación 6.1. Comparando ambos resultados se escoge el punto con la menor distancia como el que apunta en la dirección correcta.

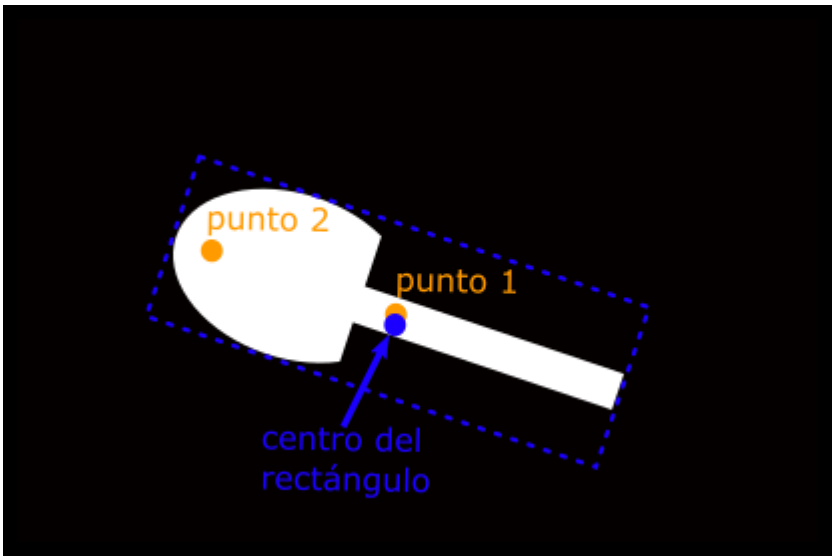


Figura 5.6. La distancia al centro del rectángulo es menor para el punto 1 que para el punto 2

Si el punto escogido es el punto 1, se usará el ángulo obtenido mediante “cv.fitEllipse()" [10] como el ángulo de rotación en el eje Z del sistema de referencia del plano de trabajo. En caso de escoger el punto 2 simplemente se restan 180º a ese mismo ángulo. De esta manera se obtiene siempre la dirección correcta, independientemente de la orientación del objeto. La figura 5.7 compara la dirección calculada únicamente con el ángulo de la elipse con la dirección calculada con este nuevo método.

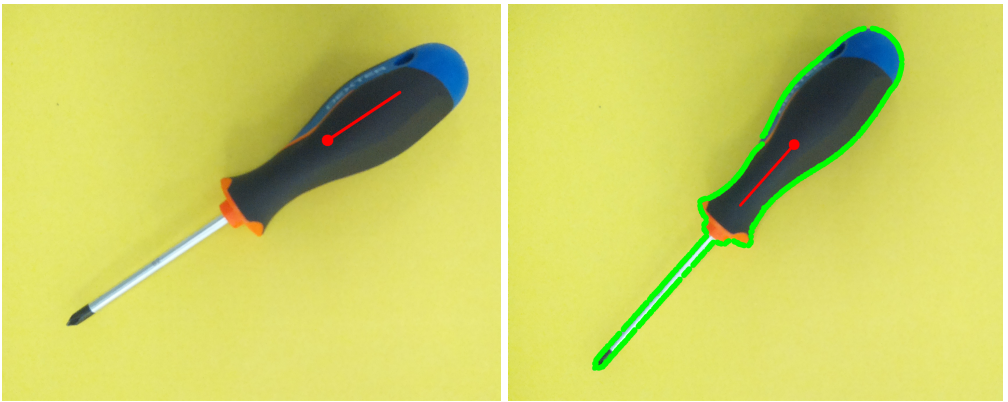


Figura 5.7. Dirección incorrecta calculada con elipse (izquierda) y dirección correcta calculada con nuevo método (derecha)

Este nuevo método, sin embargo, cuenta con una vulnerabilidad. Si un objeto es simétrico y tiene una relación de aspecto próxima a la unidad (no se alarga en ninguna dirección), por ejemplo un cuadrado, la elipse adscrita será casi circular y por tanto su orientación será inconsistente; como se muestra en la figura 5.8. Además, la distancia de su centroide al centro del rectángulo que lo envuelve será prácticamente 0, lo que también impide su uso para determinar la dirección.

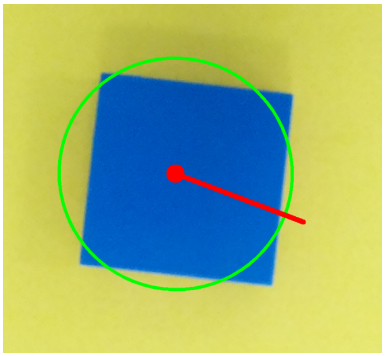


Figura 5.8. Cálculo erróneo de la orientación de un cuadrado usando la elipse

Para solucionar este problema primero hay que identificar los objetos que lo provocan, calculando su relación de aspecto y su simetría. La relación de aspecto del objeto se puede calcular dividiendo la anchura entre la altura del rectángulo que lo envuelve, tal como se explica en el apartado 4.3.3 de este documento. Para determinar la simetría se puede usar la distancia entre centroide y centro del rectángulo calculada anteriormente, si esta distancia es próxima a 0 significa que el objeto es simétrico.

Una vez detectado el objeto problemático se ha de aplicar otra técnica para calcular su rotación. En vez de usar el ángulo de la elipse adscrita, que ya se ha visto que no funciona, se usará el ángulo del lado mayor del rectángulo que envuelve el objeto. La función “cv.minAreaRect()" [10] proporciona dicho ángulo respecto la vertical de la imagen en un rango de 180^{o}. Este rango limitado, como se ha visto anteriormente, sólo proporciona la orientación del objeto pero no la dirección en la que apunta. Esto resulta un problema para otros objetos, pero en este caso al tratarse de un objeto simétrico no apunta en ninguna dirección concreta y con saber su orientación ya es suficiente.

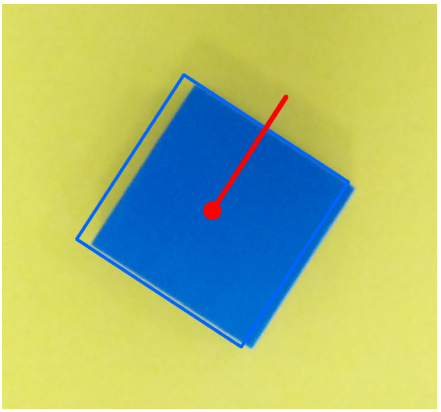


Figura 5.9. Cálculo correcto de la orientación de un cuadrado usando el rectángulo.

Calculado el ángulo con alguno de los dos métodos descritos ya se tiene el vector de rotación del objeto respecto el sistema de referencia del plano de trabajo (Ro2w), que es el último parámetro que se necesitaba. Junto a su vector de traslación (To2w) se puede obtener la matriz homogénea Mo2w (usando el método del anexo X), que transforma del sistema de referencia del objeto al sistema de referencia del plano de trabajo. Del mismo modo, usando esta vez los vectores Tw2b y Rw2b obtenidos anteriormente se obtiene la matriz homogénea Mw2b, que transforma del sistema de referencia del plano de trabajo al sistema de referencia de la base del robot.

Mo2w y Mw2b se multiplican matricialmente para obtener la matriz homogénea Mo2b, de la cual se puede extraer por fin las coordenadas del objeto respecto al sistema de referencia de la base del robot. Estas coordenadas ya se pueden enviar directamente al robot para que alcance esa posición, lo que resulta útil si solo se quisiera coger el objeto desde su centroide.

5.2. Obtención de una posición relativa al objeto

Los objetos a coger pueden tener infinidad de formas diferentes. Si bien el robot ya puede alcanzar la posición del centroide del objeto, hay muchos casos en que la geometría del objeto requiere que sea cogido desde un punto concreto distinto a su centroide. Un ejemplo es del destornillador de la figura 5.10, el cual se desea coger por la parte más estrecha del mango.

Figura 5.10. Punto de cogida de un objeto distinto a su centroide

Para coger el objeto desde el punto deseado es necesario conocer su posición relativa al sistema de referencia del propio objeto. De esta manera, si se sabe la posición del objeto respecto al robot se le puede sumar esa posición relativa para obtener la posición de cogida ("pick") respecto al robot.

El proceso de obtención de esta posición relativa consta de dos pasos (figura 5.11). Primero hay que encontrar la posición del objeto respecto a la base del robot (en forma de matriz homogénea: Mo2b) usando el método explicado en el apartado anterior (5.1). Esta matriz se ha de invertir para obtener la base del robot respecto al objeto (Mb2o).

Una vez conocida la posición del objeto el segundo paso consiste en mover manualmente el gripper del robot hasta la posición de "pick" deseada (la parte estrecha del mango en el ejemplo del destornillador). Una vez se tiene el gripper en la posición deseada, se guardan las coordenadas del TCP ("Tool Center Point") respecto la base del robot, las cuales se pueden extraer directamente de la controladora del robot mediante comunicación RTDE (se explica en el capítulo 6). Estas coordenadas se habrán de convertir también en una matriz homogénea (Mg2b) que se ha de invertir para obtener la matriz homogénea de la base al gripper.

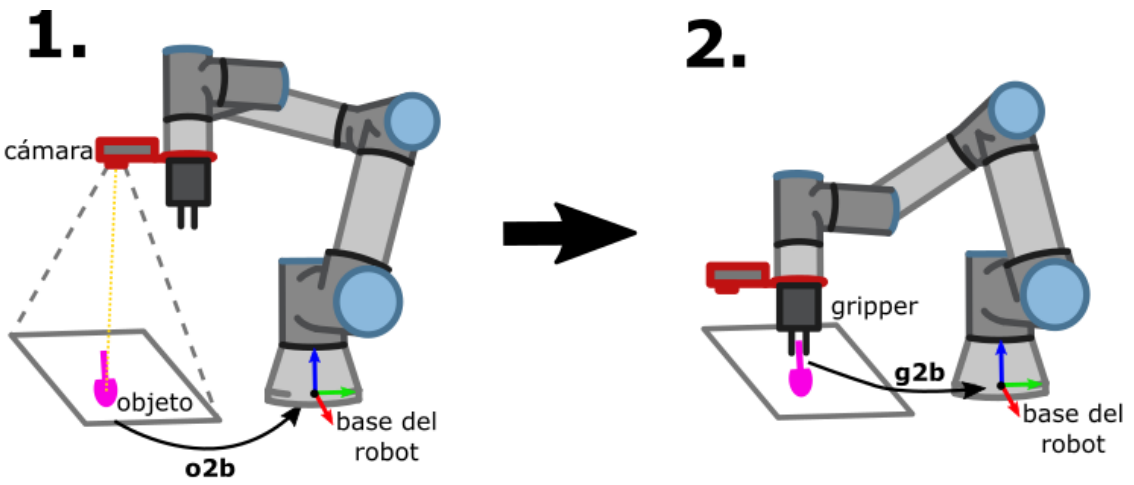


Figura 5.11. Pasos para la obtención de la posición relativa del gripper al objeto

Entre el primer paso y el segundo se ha de tener cuidado de no mover el objeto, ya que entonces su posición real no coincidirá con la calculada en la imagen.

Una vez se tienen las matrices homogéneas Mg2b y Mb2o estas se multiplican matricialmente para obtener la matriz homogénea Mg2o (ver anexo A), que representa la posición de "pick" relativa al objeto y la cual se guarda para su posterior uso.

Si en un programa se quiere que el robot alcance la posición de "pick" enseñada, simplemente se ha de multiplicar esta matriz (Mg2o) con la matriz homogénea de la posición del objeto respecto al robot (Mo2b) obtenida de la imagen. Esta multiplicación da como resultado la matriz homogénea del gripper en posición "pick" respecto al robot (Mg2b), de la cual se pueden extraer directamente las coordenadas a enviar al robot.

6. Sistema de comunicaciones

Para que el sistema funcione debe tener un sistema de comunicaciones que le permita intercambiar datos entre la raspberry y el robot, así como permitir al usuario visualizar y controlar la interfaz desde un ordenador. En este capítulo se explica dicho sistema de comunicaciones, separando la comunicación raspberry-robot de la comunicación raspberry-PC y detallando los protocolos usados en cada caso. Cabe mencionar que es necesario que todos los dispositivos (robot, Raspberry y PC) han de estar conectados a la misma red para que el sistema de comunicaciones funcione.

6.1. Raspberry-Robot

Para las comunicaciones raspberry-robot se han distinguido dos situaciones: comunicación durante la ejecución del programa de robot y comunicación fuera del programa de robot. Cada una de estas situaciones requiere un protocolo de comunicación distinto.

Para la comunicación durante la ejecución del programa de robot solamente se requiere el envío de posiciones (vector de 6 valores) desde la Raspberry para que el robot las lea y se mueva acorde a ellas. Para este caso se ha escogido el protocolo TCP/IP socket por su sencillez de uso

En el caso de la comunicación fuera de programa, se usa para la calibración de la cámara y para enseñar la posición de pick respecto la pieza. Dado que para estas tareas se ha de mover el brazo robot de forma precisa resulta más cómodo hacerlo fuera de programa desde la interfaz de la pestaña “Mover” de Poliscope (Figura 6.1). Además, fuera de programa es posible usar el modo *freedrive* propio de los robots UR, lo que facilita aún más estas tareas. En este caso se utiliza el protocolo RTDE de Universal Robots, ya que no necesita configurarse en el programa de robot y por tanto se puede utilizar sin ningun programa en ejecución.

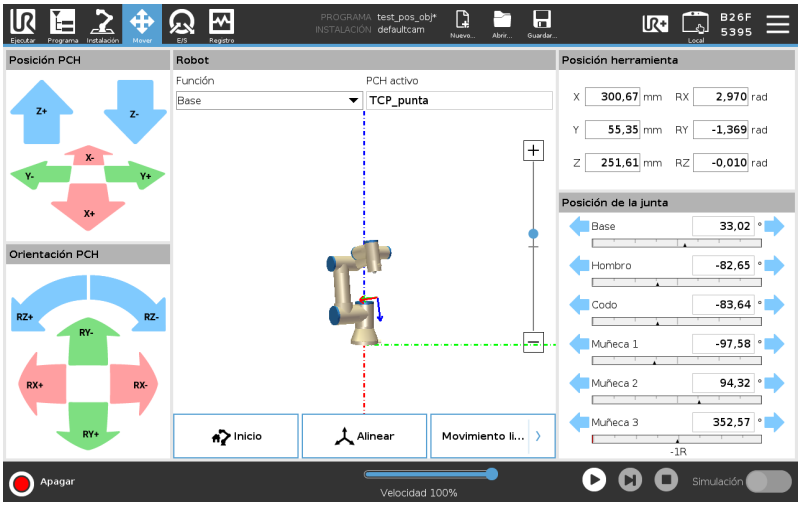


Figura 6.1. Pestaña "Mover" de Poliscope

6.1.1. Comunicación socket TCP/IP

La comunicación socket TCP/IP consta de dos partes: un servidor y un cliente. En este caso el servidor es la propia Raspberry Pi y el cliente es el robot.

Por parte de la Raspberry Pi ha sido necesario implementar el servidor en Python 3 usando la librería socket. Para usar dicha librería simplemente hay que importarla al inicio del programa mediante la línea de código: import socket. No hace falta instalarla, ya que viene incluida con el propio Python. Una vez importada la librería hay que crear el socket y asignarlo a una dirección y puerto, según muestra el siguiente fragmento de código:

```
# Crear socket TCP/IP:

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Enlazar el socket al puerto:

sHostIpAddress = "10.52.17.116" # dirección IP del servidor (la raspberry)

server_address = (sHostIpAddress, 50000) # se usa el puerto 50000

print('starting up on {} port {}'.format(*server_address))

sock.bind(server_address) # se enlaza el socket a la dirección y puerto asignados
```

Una vez inicializado el servidor ya se puede establecer conexión con el cliente (el robot en este caso) mediante el siguiente fragmento de código:

```
sock.listen(1) # esperar a un cliente

print('waiting for a connection')

connection, client_address = sock.accept() # aceptar el cliente (el robot)

print('connection from', client_address) # visualizamos la dirección del cliente
```

Con la conexión establecida ya se pueden enviar posiciones al robot. Para ello antes hay que transformar el vector de posición (formato numpy array) a una variable string que se pueda enviar. Dicha variable ha de tener el siguiente formato concreto: "(x, y, z, rx, ry, rz)", donde los tres primeros valores indican la posición y los tres últimos la orientación, la razón de este formato concreto se explicará más adelante. Para realizar esta transformación se ha implementado el siguiente código:

```
obj_coord = "(" # creamos el string a enviar

for ele in range(len(objP)): # para cada valor del vector posición

        obj_coord += str(objP[ele]) # añadimos el valor al string

        if ele != 5 : # si no estamos en el último valor añadimos una coma

             obj_coord += ", "

obj_coord += ")" # cerramos el paréntesis del string

connection.sendall(bytes(obj_coord+"\n", 'utf8')) # enviamos la posición al robot
```

Con este mismo método también es posible enviar un valor individual usando el formato "(v)". Esto es útil para comunicar al robot que objeto es que ha encontrado la cámara. Por ejemplo, si se tienen dos objetos se envía un 1 para el primero y un 2 para el segundo.

```
obj= "(1)" # trabajamos con el objeto 1

connection.sendall(bytes(objeto+"\n", 'utf8')) # enviamos la información al robot
```

También es posible recibir datos del robot en formato string. Para ello se ha creado la función "receive_from_robot()" que se muestra a continuación:

```
def receive_from_robot():
```

```
    data = connection.recv(1024) # Recibir datos del robot
    response = str(data, 'utf8') # Convertirlos a string
    print('Response: ' + response)
    return response
```

En este caso se usa para esperar a que el robot acabe el ciclo. Si la respuesta recibida es “closing socket” se corta la conexión y se para el programa, en caso contrario el programa continúa con un nuevo ciclo.

```
resp=""
while len(resp)==0: # Esperar al robot, continuar en bucle hasta recibir respuesta
        resp=receive_from_robot()
if resp=="closing socket": # Si se recibe este mensaje cerramos la conexión
        connection.close()
        break
```

Por la parte de robot se usa el lenguaje de programación URscript, propio de los robots UR. Este lenguaje incluye funciones específicas para la comunicación por sockets. Antes de iniciar el programa se hace uso de la función socket_open(dirección, puerto) para inicializar la comunicación socket. Dentro de dicha función se ha de incluir la dirección ip de la Raspberry y el puerto a utilizar, en este caso el 50000. Esta función se incluye dentro de un bucle, de modo que el programa no comienza hasta asegurarse de haber establecido la comunicación.

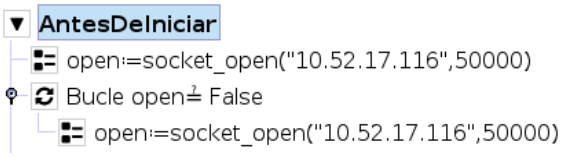


Figura 6.2. Sección del programa de robot para inicializar la comunicación socket.

Una vez dentro del programa se usa la función socket_read_ascii_float(6) para recibir posiciones de la Raspberry. Esta función requiere recibir un string con el formato “(x, y, z, rx, ry, rz)” para funcionar correctamente, esta es la razón por la que es necesario realizar una conversión de formato en el código de la Raspberry. Una vez recibida la información, la función convierte los

datos a un vector con los seis valores en el mismo orden. Este vector es entonces convertido a una posición que el robot puede utilizar para realizar un movimiento.

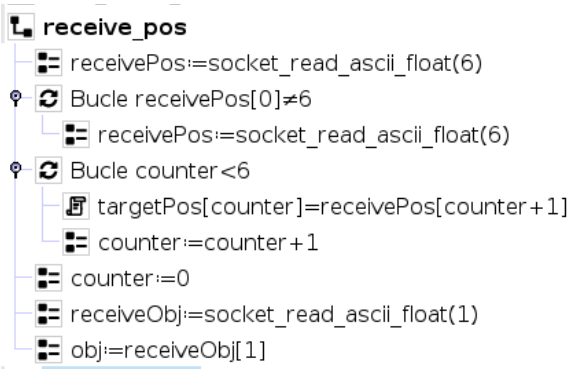


Figura 6.3. Sección del programa de robot para recibir posición y tipo de objeto de la Raspberry

Para enviar datos en formato string desde el robot a la Raspberry se usa la función socket_send_string(mensaje). En este caso se usa para indicar el final de ciclo del robot enviando el string “finciclo”. En caso de que el operario desee parar la ejecución del programa también se ha implementado la opción de enviar el mensaje "closing socket" para cerrar las comunicaciones.

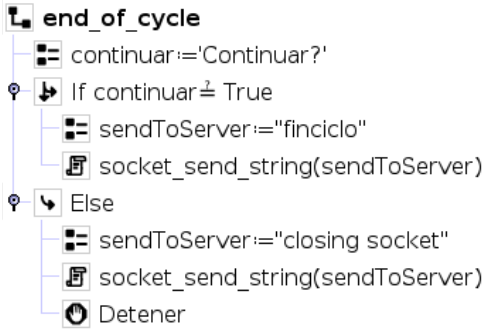


Figura 6.4. Sección del programa de robot para enviar strings a la Raspberry.

6.1.2. Comunicación RTDE

El protocolo de comunicación RTDE (“Real-Time Data Exchange”) se basa en una conexión TCP/IP y proporciona acceso en tiempo real a una gran variedad de datos procedentes de la controladora del robot: posición, velocidad, fuerza, corriente, estado de entradas/salidas, etc. El único dato de interés para este caso particular es la posición del robot, la cual se utilizará para la calibración de la cámara y para enseñar posiciones relativas a la pieza de trabajo. Es necesario aclarar que la controladora del robot actúa como servidor, permitiendo que un cliente externo (la Raspberry) se conecte y acceda a los datos en tiempo real. También es importante tener en cuenta que la

interfaz de comunicación RTDE está activada por defecto siempre que la controladora del robot se encuentre en funcionamiento, lo que significa que no es necesario que haya un programa ejecutándose para poder utilizarla.

Para poder usar esta comunicación no hay que modificar nada por parte del robot, solamente asegurarse de que no esté deshabilitada en el menú de ajustes de servicio (figura 6.5).

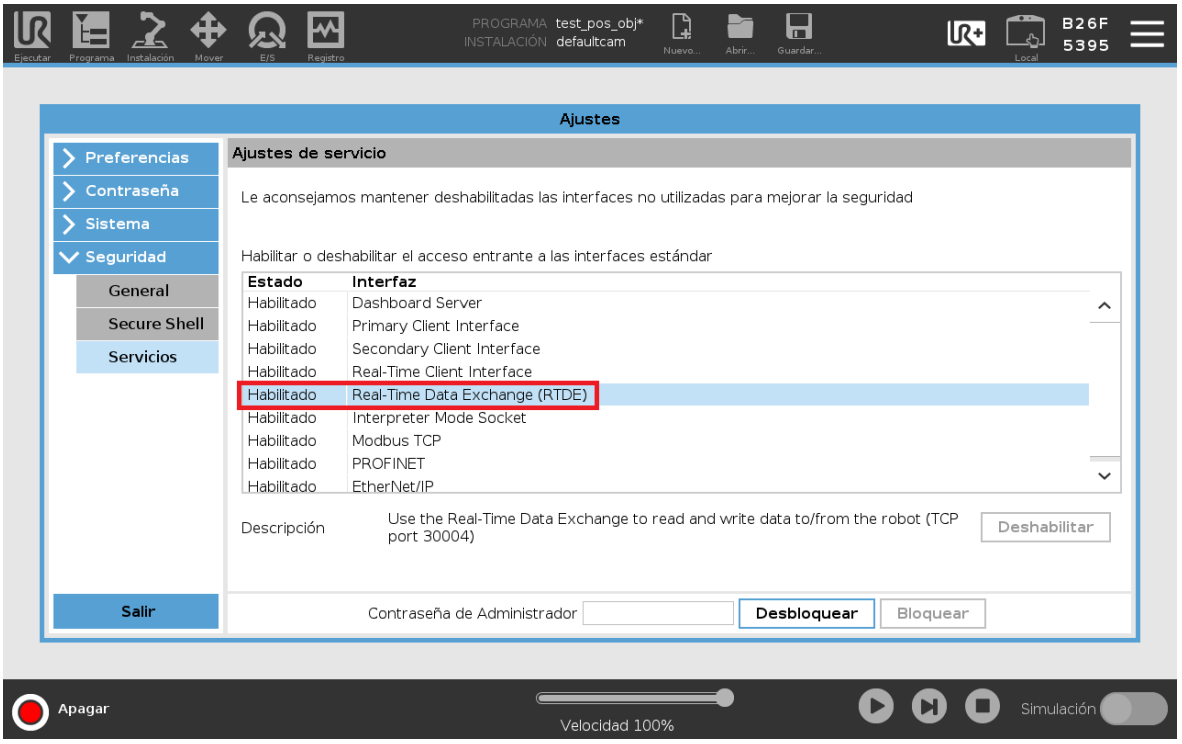


Figura 6.5. Menú de ajustes de servicio de Polyscope

Por parte de la Raspberry es necesario implementar este protocolo de comunicaciones en el código Python. Para facilitar esta tarea, Universal Robots proporciona una librería de cliente RTDE implementada en Python [11]. Dicha librería incluye funciones para la configuración, conexión y uso de la comunicación RTDE con robots UR. Para poder usarla se ha descargado una copia (en forma de carpeta con el nombre "rtde") y se ha incluido en el mismo directorio donde se encuentra el archivo Python que la va a utilizar. Dentro del programa se puede importar con las siguientes líneas de código:

```
import rtde.rtde as rtde

import rtde.rtde_config as rtde_config
```

El uso de esta librería requiere crear un archivo de configuración .xml que incluya todos los datos que se vayan a querer recibir de la controladora. En este caso solamente se necesita la posición actual del robot, por lo que el archivo .xml queda de la siguiente manera:

```
<?xml version="1.0"?>

<rtde_config>

        <recipe key="state">

            <field name="actual_TCP_pose" type="VECTOR6D"/>

        </recipe>

</rtde_config>
```

Una vez creado el archivo, se puede acceder a él desde el programa Python para configurar la comunicación y conectarnos al robot. Utilizando las funciones proporcionadas por la librería, el código de configuración y conexión es el que se muestra a continuación:

```
ROBOT_HOST = "10.52.17.250" # dirección IP del robot

ROBOT_PORT = 30004 # puerto de la interfaz RTDE

config_filename = "rtde_configuration.xml" # nombre del archivo de configuración

conf = rtde_config.ConfigFile(config_filename) #cargamos el archivo de configuración

state_names, state_types = conf.get_recipe("state") # datos que el robot enviará

# Nos conectamos al robot:

con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT) # creamos la conexión

print("connecting to robot...")

con.connect() # establecemos conexión

print("connected")

con.send_output_setup(state_names, state_types) #indicamos al robot que datos enviar
```

Con la conexión ya establecida es posible pedirle en cualquier momento al robot que empiece a enviar datos para poder recibir la posición actual del robot. Una vez recibida la información se

avisa al robot que puede dejar de enviar datos. El código que implementa este proceso es el que se muestra a continuación:

```
if not con.send_start(): #pedimos al robot que empiece a enviar datos

    sys.exit()

print("getting TCP pose...")

state = con.receive() #recibimos la podición actual del robot

act_TCP = state.actual_TCP_pose #guardamos la pose en una variable

con.send_pause() #avisamos al robot que deje de enviar datos
```

Una vez acabado el proceso de calibración o ya enseñadas las posiciones necesarias se utiliza la función “con.disconnect()” para cerrar la conexión RTDE.

6.2. Raspberry-ordenador: VNC

Como se ha decidido que tanto la interfaz como el sistema de visión se ejecute íntegramente en la Raspberry Pi es necesario el acceso remoto a esta para poder visualizar y controlar el sistema. Para ello se ha utilizado una conexión VNC.

Virtual Network Computing (VNC) es una tecnología de software libre basada en una estructura cliente-servidor que se utiliza para la observación y control remoto de un ordenador servidor desde otro ordenador cliente. En este caso concreto el ordenador servidor es la propia Raspberry Pi y el ordenador cliente es el PC desde el que se va a controlar el sistema.

VNC es independiente de la plataforma, es decir, cualquier sistema operativo que admita VNC puede compartir la pantalla de una máquina con cualquier otro sistema operativo que disponga de un cliente VNC (en este caso se utiliza VNC Viewer). Esto significa que el sistema no está limitado a ser controlado desde otro dispositivo con un sistema operativo basado en Linux. Mientras tenga instalado VNC Viewer se puede utilizar cualquier dispositivo tanto Windows como Mac. Incluso es posible hacer uso de dispositivos móviles Android o iOS, aunque esto no se recomienda ya que la pantalla táctil resulta muy cómoda para controlar el sistema.

Para poder hacer uso de la conexión VNC antes de nada ambos dispositivos (Raspberry Pi y PC) han de estar conectados a la misma red, ya sea Wi-Fi o Ethernet. Por parte de la Raspberry Pi solo hay que habilitar el servidor VNC desde la configuración de interfaces (figura 6.6).

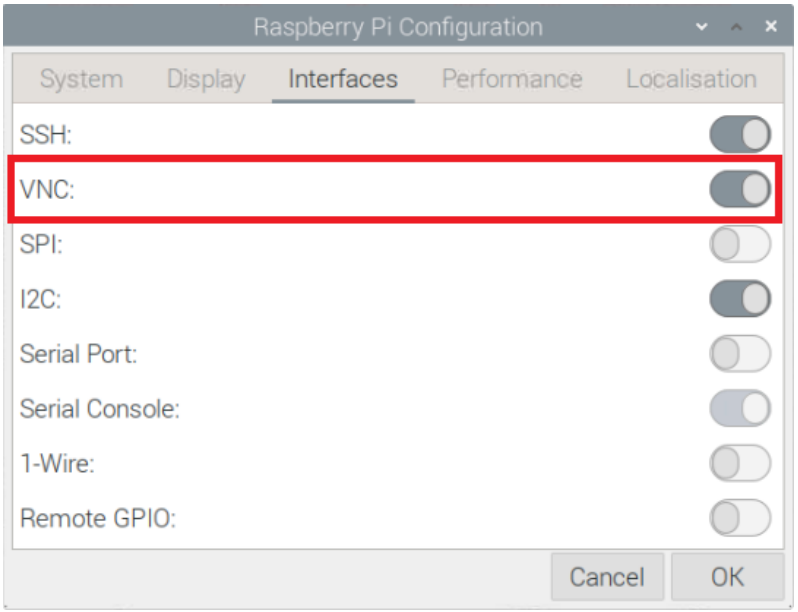


Figura 6.6. Configuración de interfaces de Raspberry Pi

Para poder acceder a la Raspberry Pi desde un PC (o cualquier otro dispositivo) hay que instalar VNC Viewer [12]. Desde VNC Viewer se ha de crear una nueva conexión utilizando el hostname de la Raspberry Pi (por defecto es “raspberrypi.local”) o su IP (la cual depende de la red).

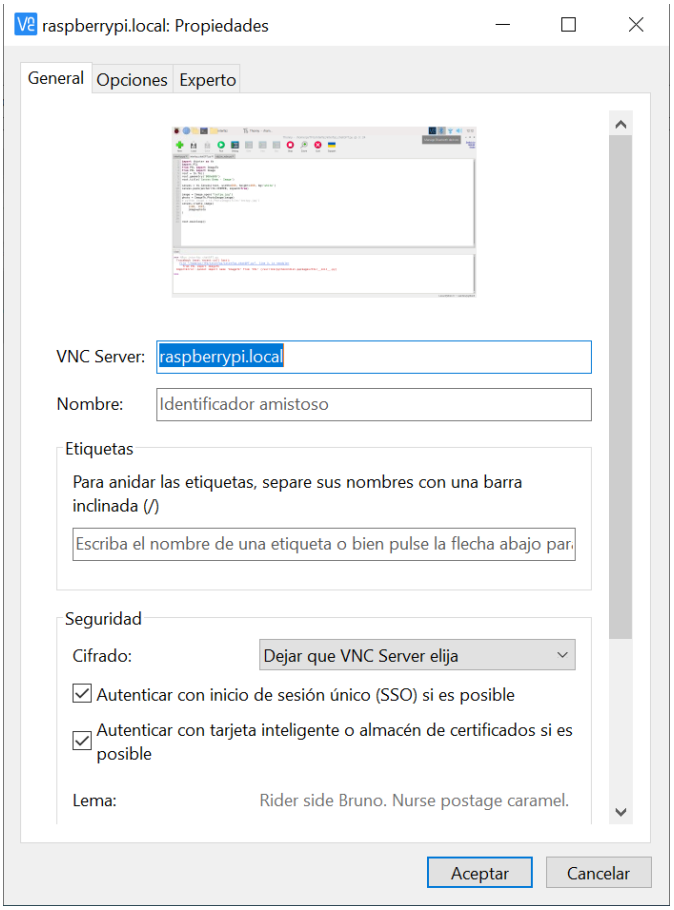


Figura 6.7. Configuración de una nueva conexión desde VNC Viewer

Una vez creada la conexión, la primera vez que se intenta acceder hay que introducir el nombre de usuario y la contraseña de la Raspberry Pi. Tras estos pasos ya se tiene acceso completo al sistema operativo de la Raspberry Pi, el cual aparece como una ventana en el PC. Desde esta ventana es posible utilizar el teclado y ratón del PC para controlar la Raspberry (figura 6.8).

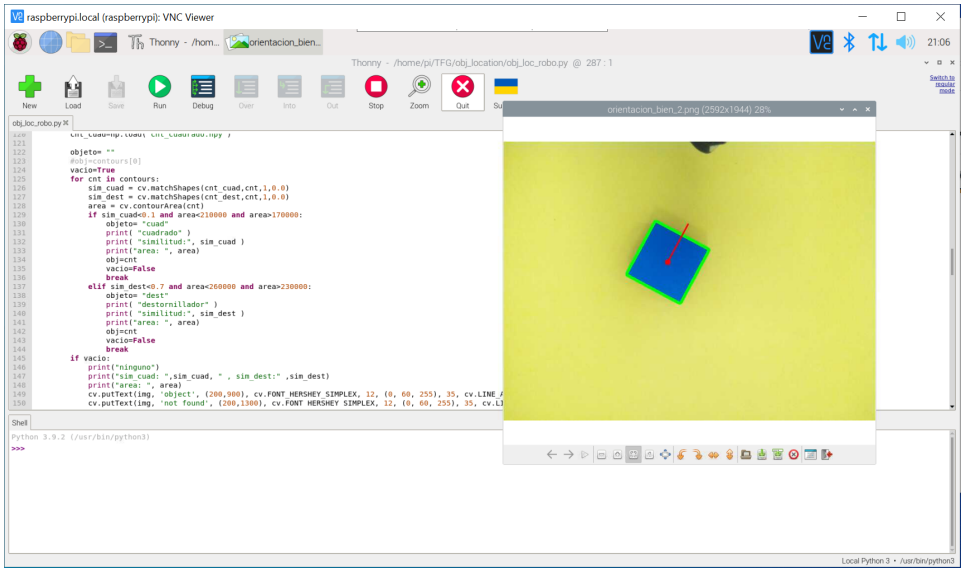


Figura 6.8. Visualización de Raspberry OS desde una ventana de Windows

7. Ejemplo de aplicación

Una vez implementado todo lo visto en los capítulos anteriores se tiene un sistema de cámara “eye-on-hand” que se puede usar sobre robots UR. Para comprobar su funcionalidad se ha creado una sencilla aplicación de ejemplo que hace un uso conjunto de todas las partes del sistema.

En este capítulo se hace una descripción de dicha aplicación, incluyendo la escena a tratar y los objetivos fijados. A continuación se detalla el procedimiento a seguir para configurar el sistema: conexión, calibración, procesado de imagen para la distinción de objetos y obtención de posición “pick” relativa al objeto. Por último se explica el ciclo de ejecución, tratando tanto el programa de la raspberry como el del robot.

Cabe mencionar que no se entrará en detalle técnico de cada uno de los pasos, ya que se usan técnicas y métodos ya explicados en otros capítulos del documento.

7.1. Descripción de la aplicación

En esta aplicación de ejemplo se tiene la escena mostrada en la figura 7.1.

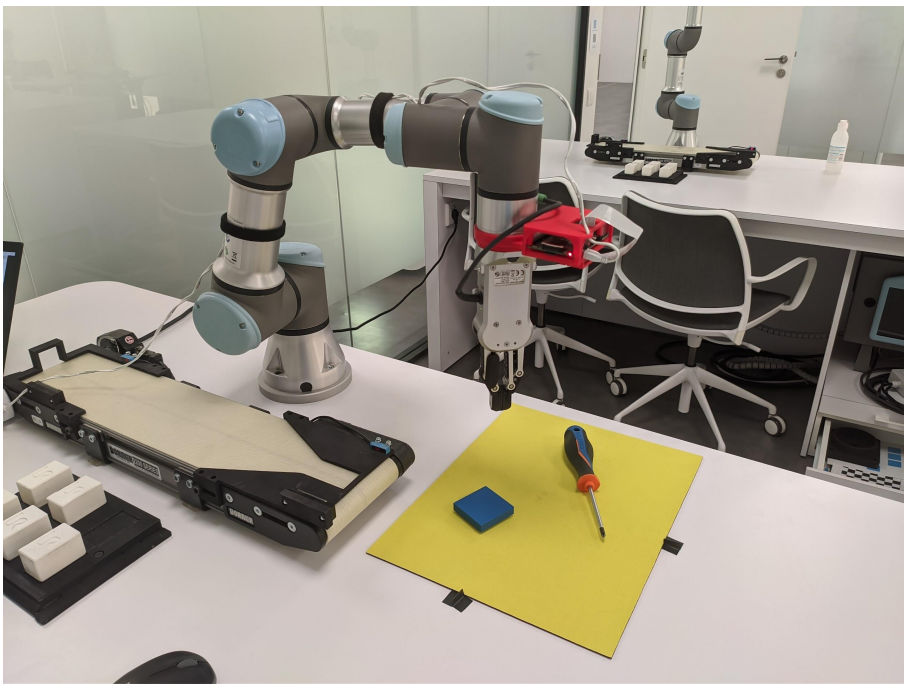


Figura 7.1. Escena de la aplicación de ejemplo

Se puede ver como sobre un robot UR3e se tiene montada la cámara “eye-on-hand” y una pinza de la marca OnRobot (concretamente un modelo antiguo de la pinza RG2 de OnRobot). Esta pinza,

a diferencia de su versión más nueva y de la mayoría de las pinzas modernas, tiene la peculiaridad de estar montada a 45° respecto al eje X del TCP de la muñeca. Esto provoca que la punta de la pinza se vea ligeramente en la imagen (figura 7.2) y se tengan que capturar las imágenes con la pinza cerrada.

Junto al robot también se tiene una cinta transportadora que obliga a desplazar el plano de trabajo a un lateral. Dicho plano de trabajo viene representado por una alfombrilla amarilla que servirá como el color de fondo de la imagen. Sobre la alfombrilla se pueden distinguir los dos objetos que se utilizarán en esta aplicación: un prisma de base cuadrada (se referirá a él simplemente como cuadrado) y un destornillador.

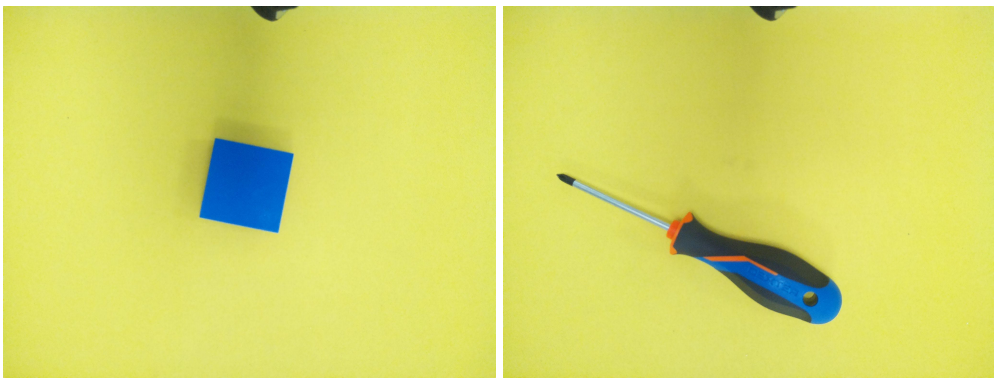


Figura 7.2. Objetos usados en la aplicación: cuadrado (izquierda) y destornillador (derecha)

Esta escena se encuentra dentro de la sala de training de las oficinas de Universal Robots Spain S.L. (figura 7.3). Dicha sala dispone de una iluminación de techo que no se puede controlar, por suerte ilumina bien la escena y no genera sombras fuertes que molesten en el procesado de imagen.

Figura 7.3. Sala de training de las oficinas de Universal Robots Spain S.L.

El objetivo de la aplicación es identificar cada uno de los dos objetos, asegurándose que no se confunda uno con el otro. Una vez identificado el objeto se tendrá que calcular su posición y hacer que el robot vaya a cogerlo desde un punto concreto. Dependiendo de qué objeto se coja, el robot deberá depositarlo bien sobre la cinta transportadora en caso del cuadrado o sobre la mesa en caso del destornillador. Al hacer esto sobre ambos objetos la escena deberá quedar como se muestra en la figura 7.4, con cada objeto en su respectivo sitio.

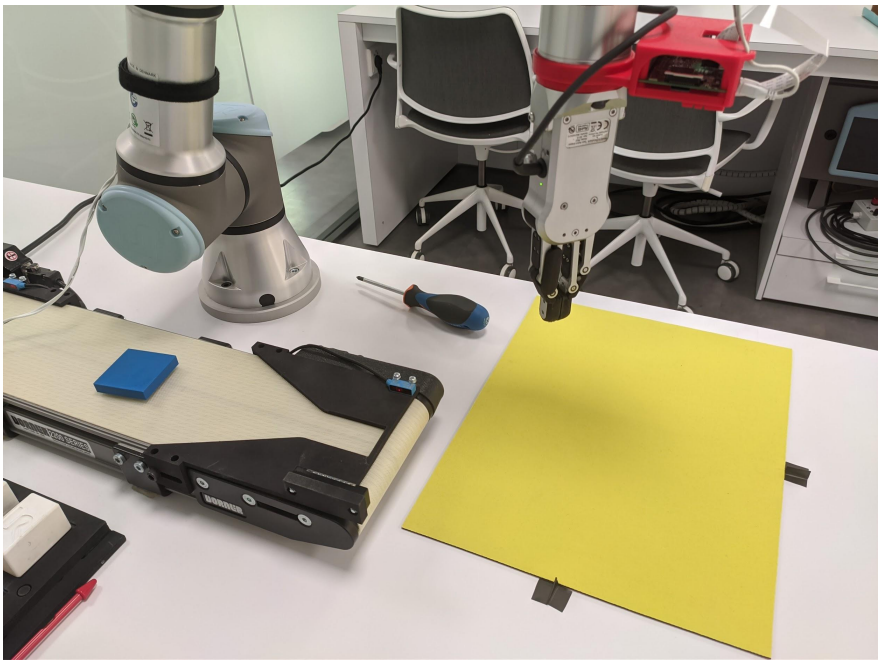


Figura 7.4. Escena objetivo de la aplicación de ejemplo

7.2. Configuración

Antes de ejecutar ningún programa debe configurarse el sistema a la aplicación concreta. El primer paso de todos es montar la cámara al robot siguiendo los pasos descritos en el apartado 3.4 de este documento. Hecho esto debe configurarse las comunicaciones, calibrar el sistema óptica y geométricamente y aprender cada uno de ambos objetos junto a su punto de cogida ("pick") deseado.

7.2.1. Comunicación

Para que el robot, la Raspberry y el PC puedan comunicarse primero hay que asegurarse de que estén los tres conectados a la misma red. En este caso se han conectado el PC y la Raspberry a una red wifi de la sala de training a la que el robot ya se encontraba conectado.

Para poder usar los protocolos de comunicación socket TCP/IP y RTDE detallados en el capítulo 6 primero se han de hallar las direcciones IP del robot y la Raspberry. En el caso de la Raspberry se puede ejecutar el comando "ifconfig" en una terminal, proporcionando la dirección IP entre otros parámetros de comunicación (figura 7.5) .

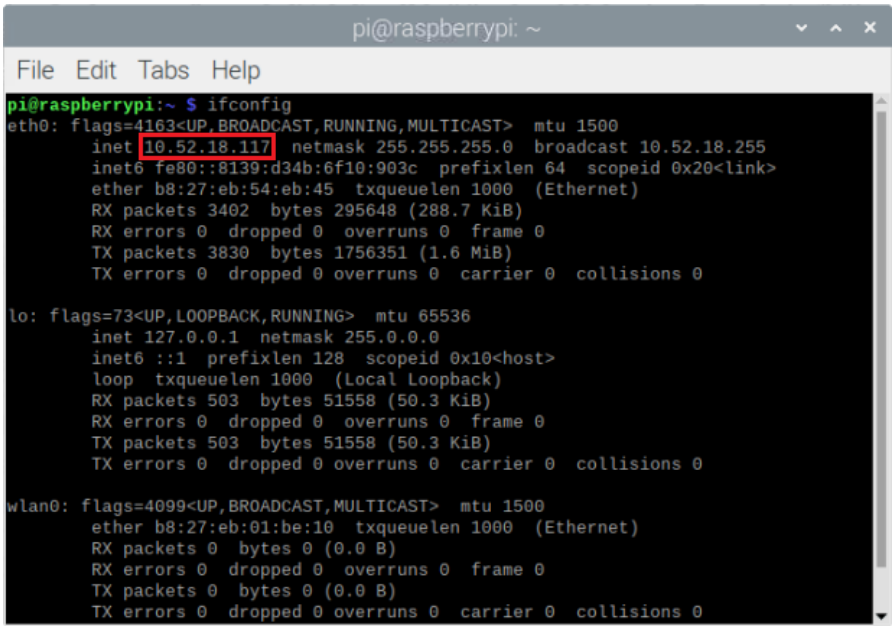


Figura 7.5. Obtención de la IP de la Raspberry mediante el comando "ifconfig"

En el caso del robot la dirección IP se puede encontrar en los ajustes del sistema, en la pestaña "Red" (figura 7.6). Por motivos de seguridad para UR, en este proyecto no se proporcionará la dirección del robot usado.

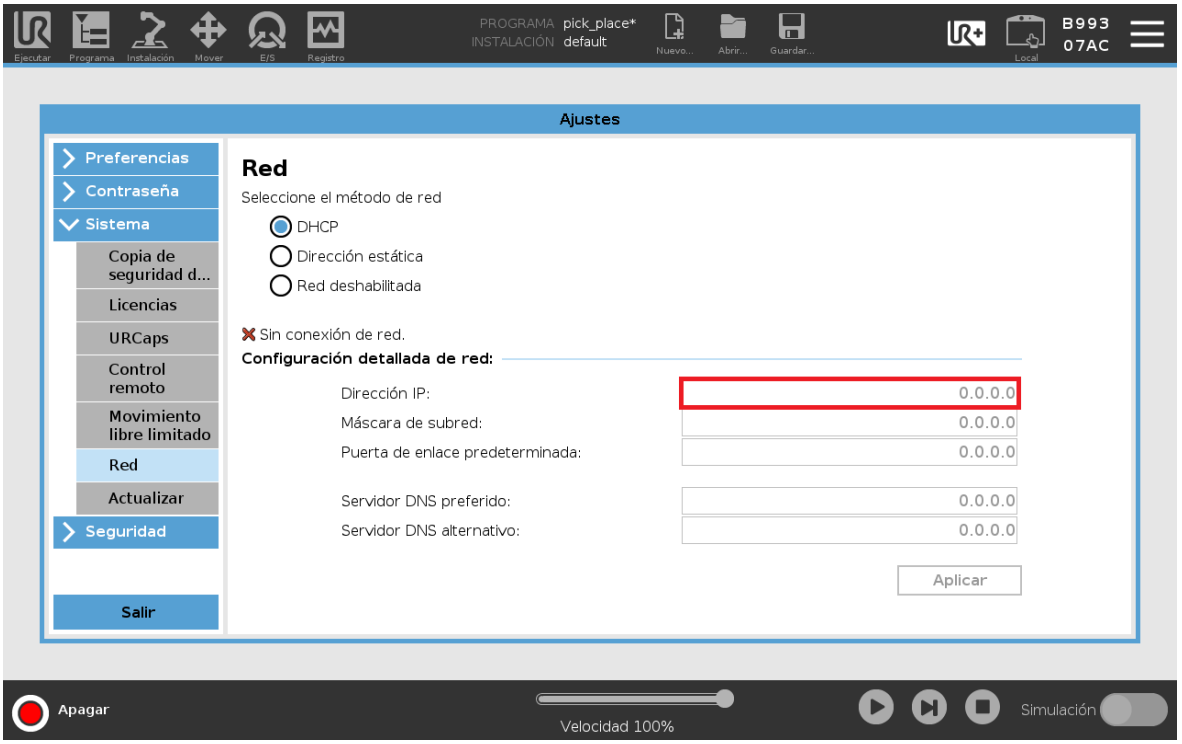


Figura 7.6. Obtención de la IP del robot

7.2.2. Calibración

Una vez montada la cámara y configuradas las comunicaciones se procede a realizar la calibración del sistema, tanto óptica como geométrica (explicadas en el apartado 4.2 de este documento). Para ambos tipos de calibración se requieren de al menos diez imágenes del patrón de calibrado (tablero de ajedrez). Dado que se pueden usar las mismas imágenes para ambas calibraciones estas se ejecutarán conjuntamente usando el código Python del anexo C.1.

Al ejecutar el código, este procederá a capturar una imagen y mostrarla al usuario. Es importante que esta primera imagen se tome desde la posición de captura deseada en la aplicación, ya que es la imagen que se usará para calcular la proyección de los píxeles al plano de trabajo. En esta posición de captura se debe visualizar el área de trabajo deseada de forma perpendicular. Para facilitar la alineación de la cámara con el eje vertical se usa el botón Alinear de la pestaña Mover de Polyscope. Para guardar esta posición en Polyscope se enseña como una función de punto llamada “captura” (figura 7.8)

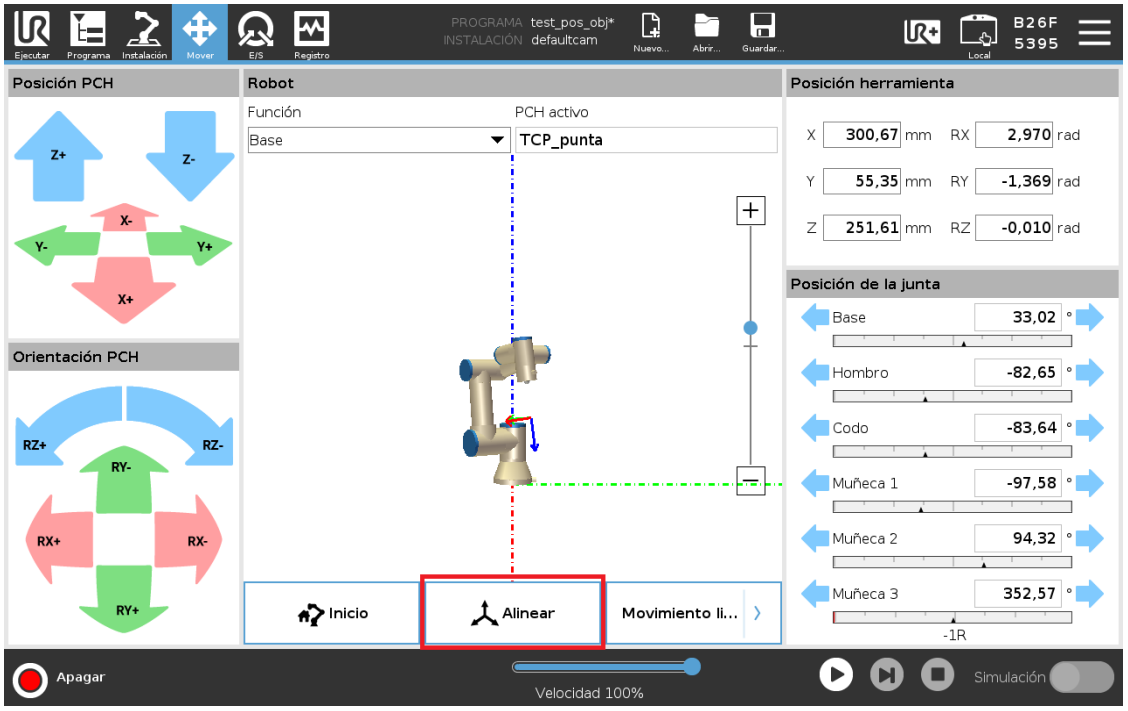


Figura 7.7. Pestaña Mover de Polyscope donde se destaca el botón Alinear

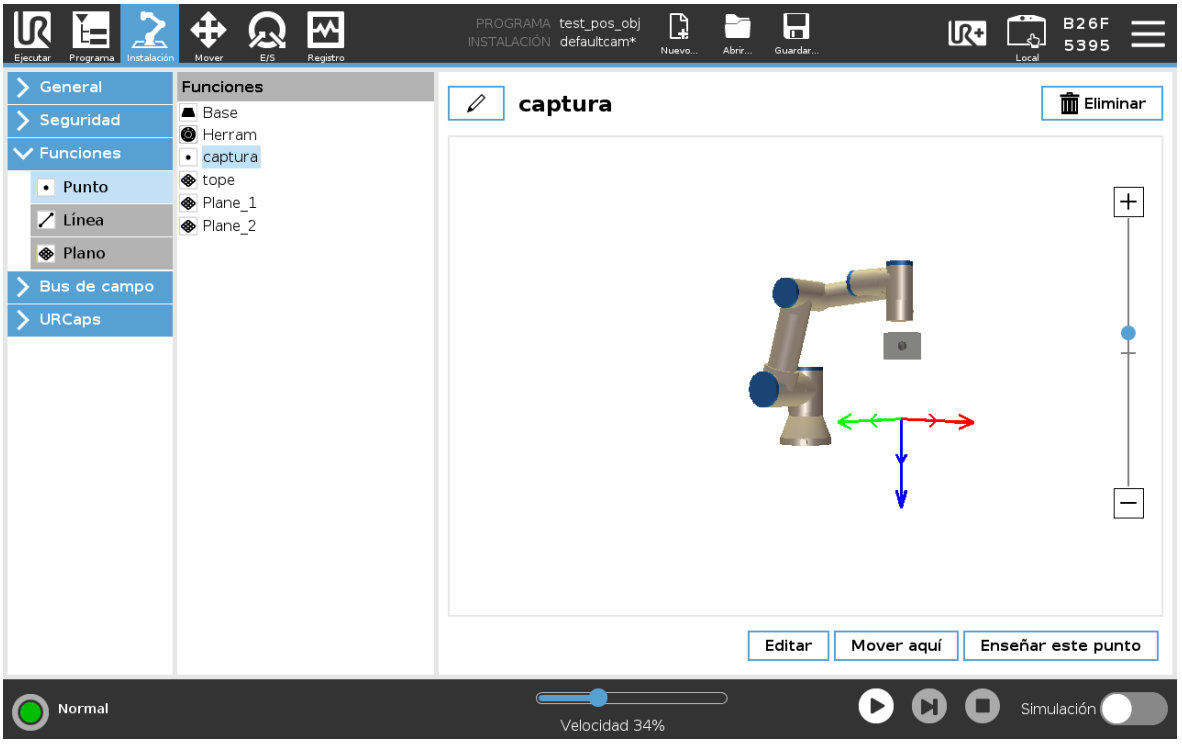


Figura 7.8. Guardado de la posición de captura como función de punto en Polyscope

Si no encuentra el patrón de ajedrez la imagen se mostrará en escala de grises y con el texto “chessboard not found” (figura 7.9), indicando al usuario que debe mover la cámara a una nueva posición.

Figura 7.9. Captura correcta (izquierda) y captura incorrecta (derecha) del patrón de calibración

En caso de sí encontrar el patrón, la imagen se muestra a color y el código espera a que el usuario confirme que la imagen es correcta pulsando la tecla enter, si se pulsa cualquier otra tecla el código la descartará y capturará una nueva. Si la imagen es aceptada se guarda junto a la posición de robot desde la que se tomó (para su posterior uso en la calibración) y se procede a capturar la siguiente imagen, así hasta tener 10 capturas de calibración (figura 7.10). Entre cada captura se debe mover la cámara junto al robot a una nueva posición de manera que se vea el patrón desde distintos ángulos.

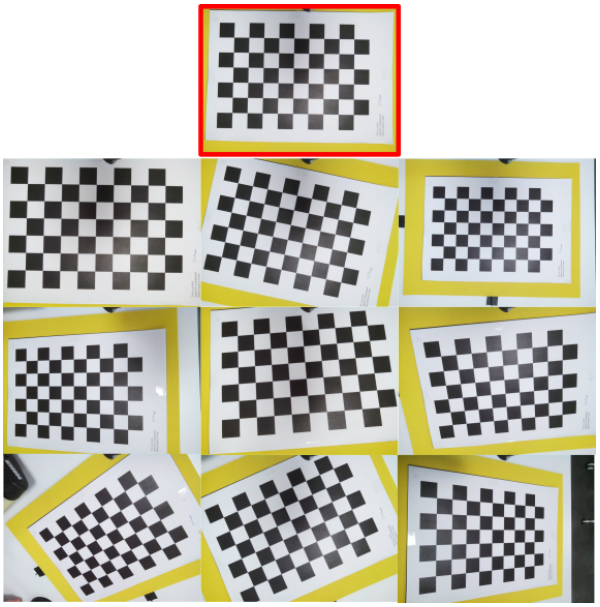


Figura 7.10. Capturas del patrón de calibración, siendo la primera desde la posición de captura de la aplicación

Una vez obtenidas todas las capturas del patrón, el código procede a calcular con ellas los parámetros de la calibración óptica. A continuación usa estos parámetros de calibración óptica junto a las mismas capturas y sus respectivas posiciones de robot para calcular los parámetros de calibración geométrica. Todos estos parámetros se explican más en detalle en el apartado 4.2 de este documento. Por último el código guarda dichos parámetros de calibración para su posterior uso en otros programas.

7.2.3. Procesado de imagen

Para poder distinguir los dos objetos de la aplicación (cuadrado y destornillador) primero se ha de procesar la imagen siguiendo los pasos indicados en el apartado 4.3 de este documento: binarizado, aplicación de operaciones morfológicas y obtención de contornos y sus propiedades.

Entre el binarizado clásico y el binarizado por color se ha escogido el segundo (de ahí el fondo amarillo). La razón ha sido que el binarizado clásico no detectaba el vástago del destornillador, separando la punta del mango (figura 7.11). En cambio el binarizado por color consigue detectarlo entero de forma robusta.

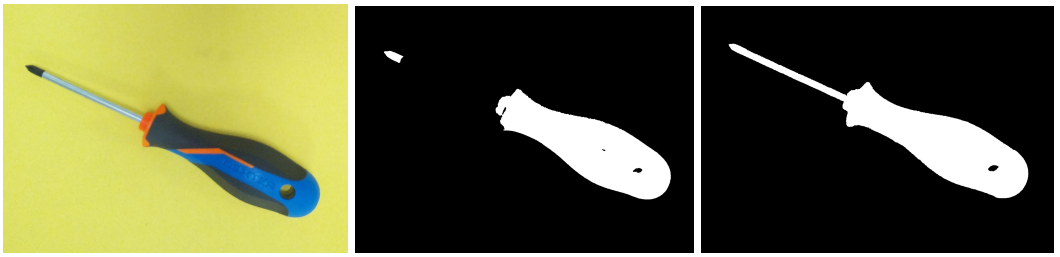


Figura 7.12. Destornillador: imagen original (izquierda), binarizado clásico (centro) y binarizado por color (derecha)

Las operaciones morfológicas en esta aplicación no son muy relevantes, ya que el binarizado no genera mucho ruido y los objetos no se fragmentan. Aun así se han aplicado las operaciones de apertura y cierre para eliminar cualquier ruido de sal y pimienta que pudiera surgir en alguna captura.

Con la imagen binarizada solo queda encontrar los contornos con la función “cv.findContours()" [10] y distinguir ambos objetos. Para distinguir cada objeto primero se hará una captura donde se vea claramente, de esta se obtendrá su contorno y se guardará como contorno de referencia. Para

detectar el mismo objeto en el futuro se usará la función “cv.matchShapes()” [10] para calcular su similitud con el contorno de referencia. Para hacer la detección más robusta también se calcula el área del contorno con la función “cv.contourArea()” [10] y se creará un rango dentro del cual debe estar el área de cualquier contorno que se obtenga de dicho objeto. El cálculo del área también se utiliza para descartar de primeras el contorno de la punta de la pinza que aparece en la imagen (figura 7.13).

Figura 7.13. Punta de la pinza en la imagen

El siguiente fragmento de código muestra un ejemplo reducido de esta distinción de objetos. En los anexos C.2 y C.3 se pueden encontrar los códigos que implementan el procesado de imagen completo.

```
#Obtenemos contornos de la imagen:

contours, hierarchy = cv.findContours(mask, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

#Cargamos contornos de referencia:

cnt_dest=np.load("cnt_destornillador.npy")

cnt_cuad=np.load("cnt_cuadrado.npy")

#Encontramos el objeto entre los contornos:

for cnt in contours: # para cada contorno...

        #Calculamos propiedades del contorno:

        sim_cuad = cv.matchShapes(cnt_cuad,cnt,1,0.0) #similitud con cuadrado
```

```
sim_dest = cv.matchShapes(cnt_dest,cnt,1,0.0) #similitud con destornillador
area = cv.contourArea(cnt) #Área del contorno
#Condiciones para ser cuadrado (similitud y rango de área):
if sim_cuad<0.1 and area<210000 and area>170000:
      objeto= "cuad" #indicamos que se ha encontrado un cuadrado
      obj=cnt #guardamos el contorno
      break #salimos del bucle for, ya hemos encontrado objeto
#Condiciones para ser rectángulo (similitud y rango de área):
elif sim_dest<0.7 and area<260000 and area>230000:
      objeto= "dest" #indicamos que se ha encontrado un destornillador
      obj=cnt #guardamos el contorno
      break #salimos del bucle for, ya hemos encontrado objeto
```

7.2.4. Enseñar posición de “pick” relativa al objeto

En esta aplicación se desea coger los objetos desde unas posiciones de pick concretas. En el caso del cuadrado simplemente se ha de coger desde los centros de cualquiera de su par de lados paralelos, como se muestra en la figura 7.14.

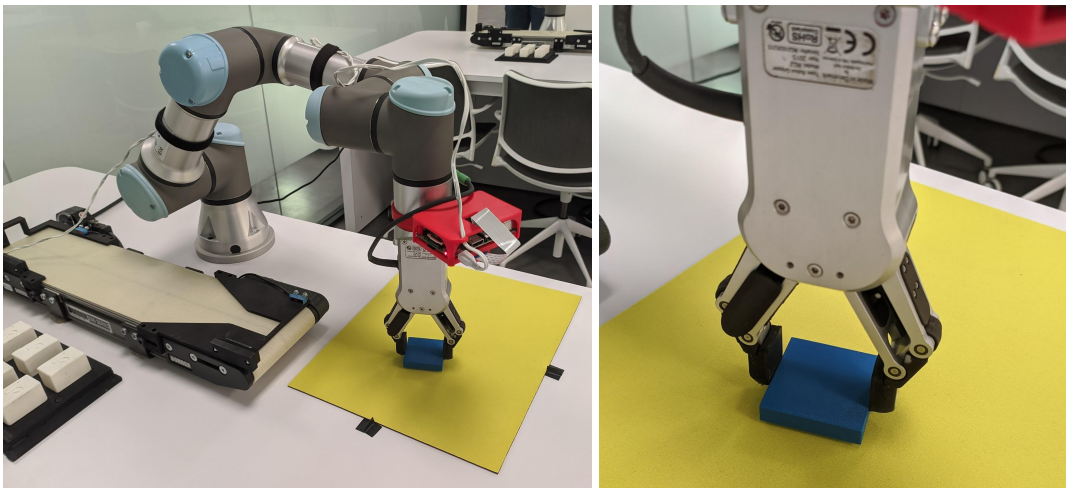


Figura 7.14. Posición de “pick” deseada para el cuadrado

Para el destornillador se requiere un punto de recogida más concreto: por la parte más estrecha de su mango (figura 7.15).

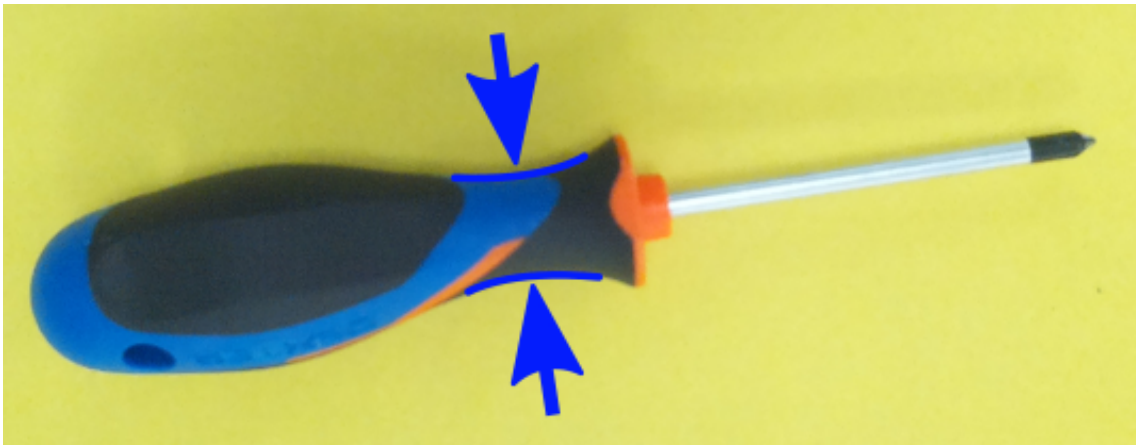


Figura 7.15. Posición de “pick” deseada para el destornillador

Para que el robot pueda coger los objetos desde estas posiciones relativas al objeto primero habrá que aprenderlas con los métodos expuestos en el capítulo 6 de este documento. El código del anexo C.2 implementa este proceso de aprendizaje de posiciones relativas. Primero calcula la posición del objeto respecto al robot a partir de una captura de imagen. A continuación espera a que el operario mueva la pinza hasta la posición de pick deseada, durante este proceso hay que tener sumo cuidado de no mover el objeto. Cuando el operario esté satisfecho con la posición de “pick” deberá pulsar la tecla enter para que el código calcule dicha posición relativa al objeto y la guarde.

Durante este proceso, el código del anexo C.2 también aprovecha para guardar los contornos de referencia del objeto mencionados en el apartado anterior.

7.3. Ejecución

Ahora que el sistema está configurado ya se puede diseñar un programa que se ejecute de forma cíclica. En esta aplicación consistirá en capturar una imagen de la escena, reconocer un objeto y que el robot lo coga y lo mueva a su respectiva posición.

Este programa consiste de dos partes: la que se ejecuta en la raspberry y la que se ejecuta en el robot.

7.3.1. Programa de la Raspberry

El programa que se ejecuta en la Raspberry incluye la parte de visión de la aplicación, que detecta los objetos y calcula su posición. Para simplificar la explicación en este apartado no se entrará en los detalles de la programación, para ello se encuentra el código Python completo y comentado en el anexo C.3. En su lugar se resumirán las partes que lo componen y se explicarán sus funciones dentro del programa.

La primera parte del programa se dedica a incluir las líneas de código que solo se ejecutan una vez al iniciarse. Entre estas líneas se incluye la importación de las librerías necesarias, la definición de funciones y la carga de los parámetros de calibración. En esta sección también se configura y realiza la conexión de la comunicación socket TCP/IP tal como se explica en el apartado 6.1.1, para ello se utilizan las direcciones IP obtenidas en el apartado 7.2.1.

A continuación ya empieza el bucle que se ejecutará cíclicamente, en paralelo al programa del robot. El resto del código viene incluido en este bucle.

Al principio del bucle se captura la imagen y acto seguido se binariza para obtener sus contornos y buscar los objetos. Los criterios para identificar al cuadrado y al destornillador son los mismos descritos en el apartado 7.2.3. En el caso de que no se encuentre ninguno de los dos objetos en la imagen esta se muestra en pantalla con el texto “object not found” (figura 7.16). A continuación el código espera a que el usuario coloque un objeto en la zona de trabajo y presione cualquier tecla para volver a capturar la escena.

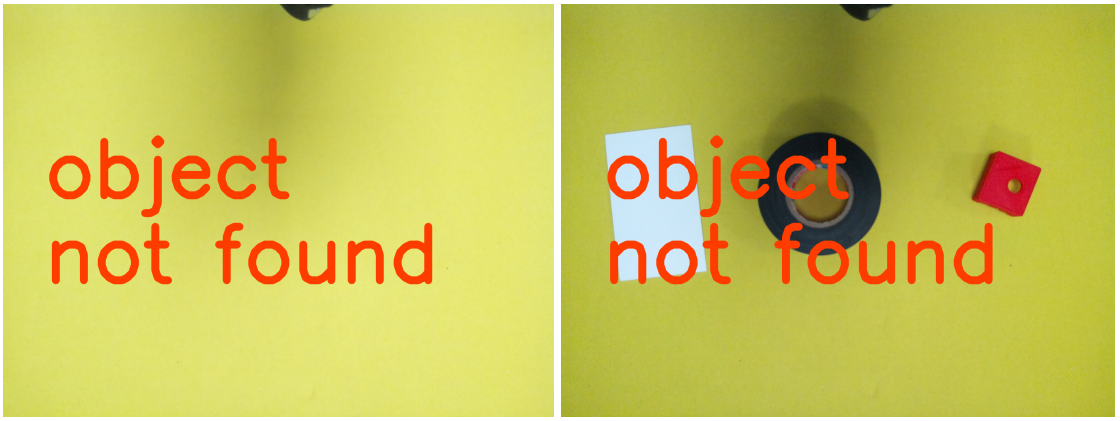


Figura 7.16. Imagen que se muestra si no se encuentra alguno de los dos objetos enseñados

En el caso de que sí se encuentre alguno de los dos objetos el programa procede a calcular su centroide y orientación en la imagen con el método explicado en el apartado 5.1. Acto seguido muestra en pantalla la imagen original con el contorno del objeto representado en verde, su centroide marcado con un punto rojo y la dirección en la que apunta con una línea del mismo color (figura 7.17). En este punto el código se queda esperando a que el usuario confirme que se ha detectado bien el objeto pulsando la tecla enter, en caso de pulsarse cualquier otra tecla capturaría una nueva imagen.

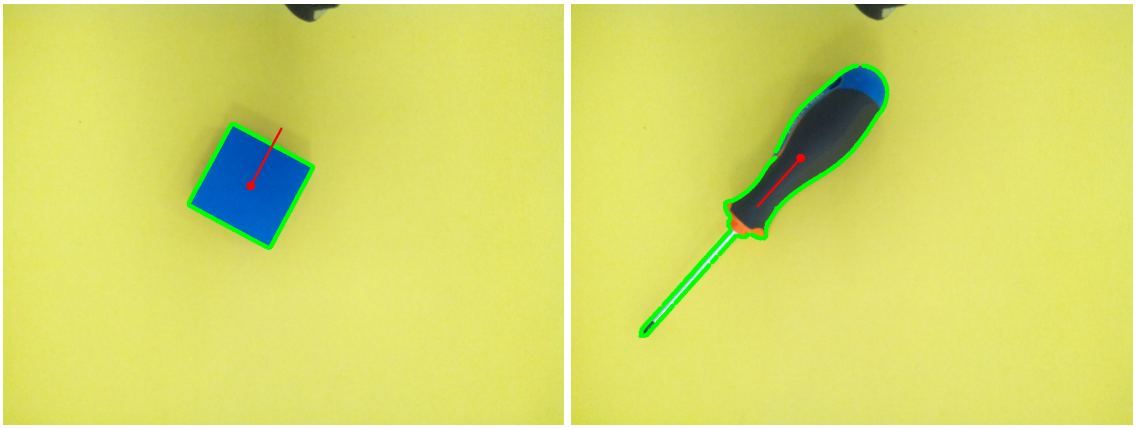


Figura 7.17. Representación de los objetos durante la ejecución del programa

Una vez se ha confirmado la correcta detección del objeto el programa procede a calcular la posición que ha de enviar al robot. Para ello primero ha de calcular la posición del objeto respecto al robot siguiendo el método explicado en el apartado 5.1. A continuación carga la posición de "pick" relativa al objeto obtenida durante la configuración (apartado 7.2.4) y se la añade a la posición del objeto para obtener las coordenadas que ha de alcanzar el robot.

Por último le envía dichas coordenadas al robot usando el método expuesto en el apartado 6.1.1. Junto a las coordenadas también envía un número que indica que objeto se ha encontrado: 1 para el cuadrado y 2 para el destornillador.

Hecho esto, el programa se queda esperando a que el robot acabe sus movimientos y vuelva a la posición de captura, momento en el que se volverá al inicio del bucle para repetir el ciclo.

El tiempo que tarda en computarse un ciclo de programa (desde que se recibe la señal de capturar imagen hasta que se envían las coordenadas al robot) ronda los 4 segundos. Este tiempo puede variar ligeramente según el posicionamiento de los objetos. Dado que no se ha dedicado un esfuerzo específico a optimizar el programa este tiempo es susceptible a ser reducido. No obstante, al tratarse de una aplicación educativa el tiempo de ejecución reducido no es demasiado relevante ya que no hay ninguna tasa de producción que se vea afectada, como sí ocurre en una aplicación industrial.

7.3.2. Programa del robot

El programa que ejecuta la controladora del robot se limita a recibir la información de la Raspberry y controlar los movimientos del robot y la pinza. El programa completo organizado en carpetas y subprogramas se muestra en la figura 7.18.

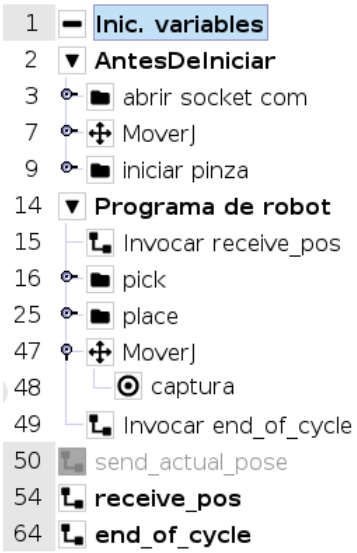


Figura 7.18. Programa del robot

En este programa, al igual que el de la Raspberry, la primera sección solo se ejecuta una vez al iniciar el programa. Esta sección incluye un nodo que inicializa todas las variables usadas durante

el programa a un valor predeterminado. También presenta una secuencia “AntesDeIniciar” que como su nombre indica se ejecuta antes de iniciar el programa principal (figura 7.19). En esta secuencia se inicia la comunicación socket usando la IP de la Raspberry obtenida en el apartado 7.2.1 y el puerto que se desee utilizar (en este caso el 50000), si no se conecta a la primera entra en un bucle en el que sigue intentándolo hasta conseguirlo. Acto seguido el robot se mueve a la posición de captura con un comando “MoverJ”. Por último se inicializa la pinza para que suelte cualquier posible objeto que tenga agarrado y luego se cierre del todo para no interferir en la imagen.

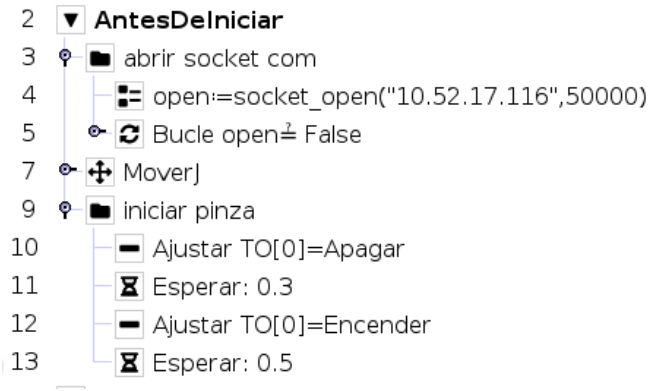


Figura 7.19. Secuencia “AntesDeIniciar” del programa del robot

Al iniciar el programa se invoca al subprograma “receive_pos” (figura 7.20). Este subprograma provoca que el robot se quede en la posición de captura esperando a recibir la posición objetivo (“targetPos”) por parte de la Raspberry vía comunicación socket, tal como se explica en el apartado 6.1.1. Una vez recibida la posición se recibe también el número que indica que objeto se ha encontrado (1 para cuadrado y 2 para robot).

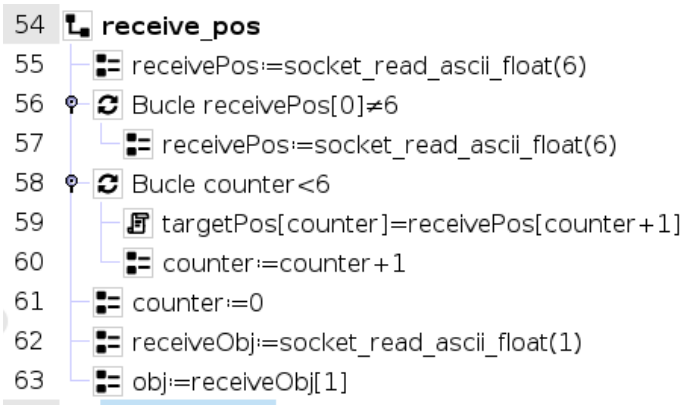


Figura 7.20. Subprograma “receive_pos” del programa de robot

Una vez recibida la posición y el tipo de objeto el robot pasa a ejecutar los comandos de la carpeta "pick" (figura 7.21). Dentro de esta carpeta lo primero que se hace es guardar la posición del objeto como una función de punto, de esta manera se puede definir una posición "p_aprox_pick" relativa a esa función. Esta posición se encontrará siempre a unos 40 mm por encima de la posición de "pick". El robot abre la pinza para acto seguido ir a la posición "p_aprox_pick". A continuación realiza un movimiento lineal hacia la posición "pick" relativa al objeto que se haya definido durante la configuración (apartado 7.2.4). Alcanzado ese punto el robot cierra la pinza para coger el objeto.

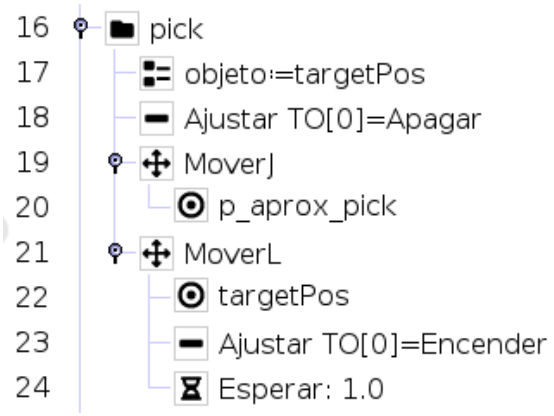


Figura 7.21. Comandos de la carpeta "pick" del programa de robot

Con el objeto agarrado el robot pasa a ejecutar los comandos de la carpeta "place" (figura 7.22). En esta sección dependiendo del objeto encontrado el robot lo llevará a una posición o a otra, en cualquiera de los dos casos la secuencia a ejecutar es la misma. Primero se pasa por un punto de paso alejado del plano de trabajo y de la cinta para asegurarse de no chocar con nada. A continuación se mueve a una posición de aproximación que se halla a unos 50 mm sobre la posición "place". Acto seguido se realiza un movimiento lineal hasta la posición "place", donde se abre la pinza para soltar el objeto. Por último se vuelve a la posición de aproximación, donde se cierra la pinza en preparación para volver a la posición de captura.

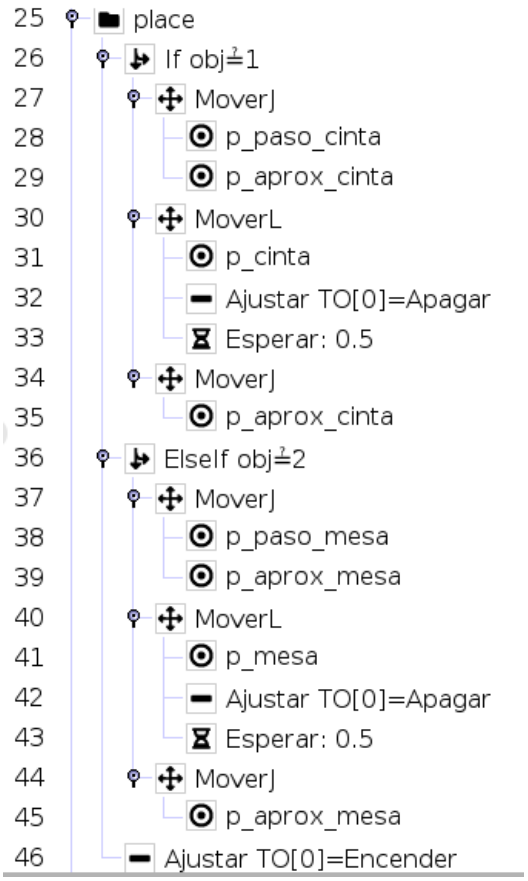


Figura 7.22. Comandos de la carpeta “place” del programa de robot

Para el caso del cuadrado la posición “place” se encuentra sobre la cinta transportadora y se le llama “p_cinta”. En el caso del destornillador la posición se encuentra sobre la mesa y se le llama “p_mesa”. Ambas posiciones se muestran en la figura 7.23.

Figura 7.23. Posición “place” del cuadrado (izquierda) y del destornillador (derecha)

El último paso del programa es invocar el subprograma “end_of_cycle” (figura 7.24). Este subprograma crea un “popup” que pregunta al usuario si quiere continuar con la ejecución del programa. Si la respuesta es afirmativa, envía a la Raspberry el string “finciclo” vía socket, indicándole que ha acabado su ciclo y que puede volver a capturar una nueva imagen y reiniciar el ciclo de programa. Si la respuesta es que no, envía el string “closing socket” para cerrar la comunicación socket y a continuación detiene el programa.

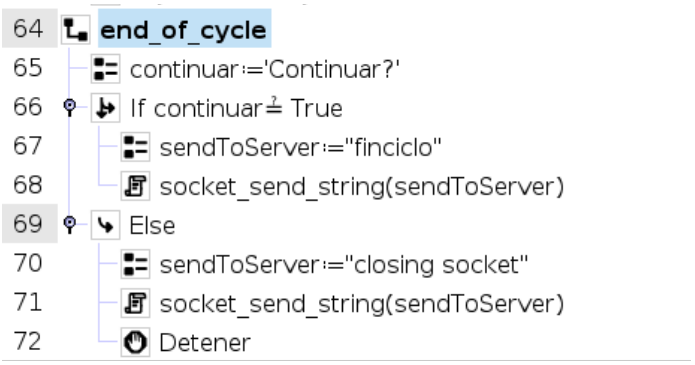


Figura 7.24. Subprograma end_of_cycle

Si se quiere que el programa se ejecute cíclicamente de forma automática sin intervención del usuario solo hay que cambiar el primer nodo de manera que quede: “continuar:=True”. De esta manera no saltará ningún popup y se enviará siempre el string “finciclo

Conclusiones

Acabado el proyecto es necesario reflexionar si se ha cumplido con los objetivos propuestos y qué dificultades se han encontrado en el camino. También se han de pensar en posibles mejoras a desarrollar a partir del trabajo realizado.

El objetivo principal del proyecto era crear una cámara “eye_on_hand” de bajo coste para robots de UR que se pudiera usar en un ámbito educativo. En este sentido sí que se ha cumplido el objetivo, ya que se ha conseguido un prototipo funcional capaz de usarse en una sencilla aplicación de visión. En esta aplicación se ha comprobado el cumplimiento de los objetivos secundarios: tener un sistema de sujeción funcional; establecer distintos tipos de comunicación entre la Raspberry, el robot y el PC; distinguir distintos objetos en la imagen y calcular su posición en el mundo real. En cuanto al objetivo de bajo coste, al haberse basado en la plataforma Raspberry y haber evitado el uso de más componentes externos el precio se ha conseguido mantener reducido.

Dicho esto, la filosofía de este proyecto de usarse en educación requería la mayor facilidad de uso posible para usuarios que no tengan experiencia en el campo de la visión. En este sentido el proyecto no ha cumplido con este criterio, ya que para crear una nueva aplicación se requiere tener conocimientos de visión y programación en Python. Para solucionar esto se podría haber creado el código en forma de una librería Python que implementa las distintas funciones, simplificando la programación a futuros usuarios.

De hecho, en la concepción inicial del proyecto también se planteó la creación de una interfaz gráfica de usuario completa. Esta interfaz dispondría de distintas pestañas para cada parte del proceso de creación de una aplicación de visión: configuración, calibración, procesado de imagen, enseñar objetos y posiciones relativas, visualizar las capturas y otra información durante la ejecución del programa, etc. De esta manera un usuario podría crear una aplicación de visión sin programar ni una sola línea de código. Sin embargo durante el desarrollo del proyecto se llegó a la conclusión de que era una tarea mucho más compleja y laboriosa de lo que se había pensado en un principio y que no daría tiempo a implementarla. Es por eso que la obvia continuación de este proyecto sería el desarrollo de dicha interfaz, que podría dar pie a otro trabajo de fin de estudios.

Otras posibles mejoras a introducir al sistema serían la implementación de más aplicaciones de visión como el control de calidad o el alineamiento con código QR. Otra opción podría ser el desarrollo de una URCap que integre el control y la visualización de imágenes en Polyscope. También se planteó la idea de incorporar Matlab-Simulink a la hora de programar el sistema de visión. Dichas mejoras también pueden dar pie a futuros trabajos de fin de estudios.

Análisis Económica

En este apartado, se realizará un desglose detallado de los costes asociados a la realización del presente trabajo, en el que se incluirán todas las siguientes categorías: costes de recursos humanos, costes de materiales, costes de software.

Para el coste de recursos humanos se tendrá en cuenta el total de horas dedicadas a la realización de este trabajo. Dado que el proyecto se realizó estando en un convenio de prácticas entre la EEBE y Universal Robots Spain S.L. el precio por hora que se usará será el establecido en dicho convenio: 8€/h.

Tabla 1. Costes de recursos humanos

Concepto	Horas	Precio/hora (€/h)	Precio total (€)
Investigación	75	8	600
Configuración de la Raspberry	30	8	240
Diseño CAD	110	8	880
Programación del sistema de visión	125	8	1000
Programación del cálculo de posiciones	75	8	600
Programación del sistema de comunicaciones	45	8	360
Prueba de prototipos	30	8	240
Implementación de aplicación de ejemplo	25	8	200
Elaboración de la documentación	120	8	960
Total horas	**635**	-	-
		Subtotal	**5080 €**

A continuación se presentan los costes asociados a los materiales utilizados en el proyecto. Cabe destacar que no se tiene en cuenta el coste del robot UR3e puesto que se tiene en cuenta que ya se dispone de él. El proyecto se ha planteado como una herramienta que se instala sobre cualquier robot de UR, por lo que los costes se limitan a los materiales de la propia herramienta. Del mismo modo tampoco se ha considerado el coste de la pinza RG2 usada para la aplicación de ejemplo, ya que la cámara se diseñó para encajar con cualquier elemento terminal.

En la tabla 2 se muestran los materiales utilizados para la creación del sistema de visión (robot y elemento terminal no incluidos). Para cada elemento se indica la cantidad usada y su precio de mercado (IVA incluido), así como el proveedor del que se ha obtenido dicho valor.

Tabla 2. Costes de materiales

Concepto	Proveedor	Cantidad	Precio unitario	Precio total (€)
Raspberry Pi 3	RS	1 ud.	48,41 €	48,41
Raspberry Camera Module v1	SATKIT	1 ud.	16,94€	16,94
Router Inalámbrico TL-WR802N	Amazon	1 ud.	24,49 €	24,49
Raise3D Premium PLA Filament	Raise3D	0,15 kg	31,90 €/kg	4,79
			Subtotal	**94,63 €**

Estos 94,63€ representan el coste aproximado que implicaría reproducir el sistema en otro robot. Si se decidiera mejorar la Raspberry y la cámara a sus versiones más modernas este precio sería ligeramente superior.

En cuanto a los costes de software, como todo lo que se ha utilizado es de código abierto (Raspberry OS, VNC, OpenCV, Scipy, y la librería RTDE de UR) no se ha requerido de la compra de ninguna licencia. Por lo tanto el coste de software puede considerarse nulo.

El coste total de desarrollo del proyecto se obtiene sumando los costes de recursos humanos a los costes de materiales, tal como muestra la tabla 3.

Tabla 3. Coste total del proyecto

Concepto	Precio (€)
Costes de recursos humanos	5080
Costes de materiales	94,63
TOTAL	**5174,63 €**

Bibliografía

[1] *Universal Robots e-Series User Manual* [en línea]. Universal Robots S.L. . Disponible en: https://s3-eu-west-1.amazonaws.com/ur-support-site/181319/99403_UR3e_User_Manual_en_Global.pdf

[2] *Raspberry Pi 3 Model B* [en línea]. Raspberry Org. [Consulta: 22 noviembre 2022]. Disponible en: https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf

[3] *Raspberry Pi Documentation – Camera* [en línea]. Raspberry Org. [Consulta: 14 agosto 2022]. Disponible en: https://www.raspberrypi.com/documentation/accessories/camera.html#libcamera-and-libcamera-apps

[4] *TL-WR802N Router inalámbrico Nano N 300Mbps* [en línia]. TP-Link España. [Consulta: 27 enero 2023]. Disponible en: https://www.tp-link.com/es/home-networking/wifi-router/tl-wr802n/

[5] *Raspberry Pi Datasheets* [en línea]. Raspberry Org. [Consulta: 18 octubre 2022]. Disponible en: https://datasheets.raspberrypi.com/

[6] Banglei Guan, Yingjian Yu, Ang Su, Yang Shang, and Qifeng Yu, *Self-calibration approach to stereo cameras with radial distortion based on epipolar constraint*. Appl. Opt. 58, 8511-8521 (2019).

[7] *Camera Modeling: Exploring Distortion and Distortion Models, Part I* [en línea]. Tangram Vision. [Consulta: 21 abril 2023]. Disponible en: https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i

[8] *OpenCV - Calibration pattern* [en línea]. OpenCV. [Consulta: 15 febrero 2023]. Disponible en: https://github.com/opencv/opencv/blob/4.x/doc/pattern.png

[9] *OpenCV - Camera Calibration and 3D Reconstruction* [en línea]. OpenCV. [Consulta: 15 febrero 2023]. Disponible en: https://docs.opencv.org/4.x/d9/d0c/group__calib3d.html#ga3207604e4b1a1758aa66acb6ed5aa65d

[10] *OpenCV - Python Tutorials* [en línea]. OpenCV . [Consulta: 02 octubre 2022]. Disponible en: https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html

[11] *Real-Time Data Exchange (RTDE) Guide* [en línea] Universal Robots SL. [Consulta: 20 noviembre 2022] Disponible en:

https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/

[12] *Download VNC Viewer* [en línea]. VNC® Connect. [Consulta: 02 septiembre 2022]. Disponible en: https://www.realvnc.com/en/connect/download/viewer/

[13] Cobas, S. P. *Strategies for remote control and teleoperation of a UR robot MEMORY* [en línea]. Trabajo final de máster, UPC, Escola Tècnica Superior d'Enginyeria Industrial de Barcelona. 2020. Disponible en: https://upcommons.upc.edu/bitstream/handle/2117/333559/tfm-muei-etseib-sergiponsacobas.pdf?sequence=1&isAllowed=y

[14] Ferriol, A.G. *DISSENY I PROTOTIPAT D'UNA PINÇA ELÈCTRICA PER ROBOT COL·LABORATIU* [en línea]. Trabajo final de grado, UPC, Escola d'Enginyeria de Barcelona Est. 2022. Disponible en: https://upcommons.upc.edu/bitstream/handle/2117/374650/TFE_Antoni_Ferriol_2022.pdf?sequence=1&isAllowed=y

[15] Xavier, M.G. *SISTEMA AUTOMÀTIC DE CLASSIFICACIÓ DE PECES BASAT EN VISIÓ ARTIFICIAL I ROBÒTICA* [en línea]. Trabajo final de grado, UPC, Escola d'Enginyeria de Barcelona Est. 2021. Disponible en: https://upcommons.upc.edu/handle/2117/356299

[16] *Wrist Camera Vision System for e-Series Universal Robots Instruction Manual* [en línea]. Robotiq. [Consulta: 15 septiembre 2022]. Disponible en: https://assets.robotiq.com/website-assets/support_documents/document/Vision_System_e-Series_PDF_20190116.pdf

[17] Juho Kannala, Janne Heikkilä and Sami S. Brandt (2008). *Geometric Camera Calibration*. University of Oulu, Finlandia. Disponible en: https://users.aalto.fi/~kannalj1/publications/preprintECEWiley2008.pdf

[18] Peter I. Corke. *VISUAL CONTROL OF ROBOTS: High-Performance Visual Servoing*. CSIRO Division of Manufacturing Technology, Australia. Disponible en: https://petercorke.com/bluebook/book.pdf

[19] *TCP/IP SOCKET COMMUNICATION VIA URSCRIPT* [en línea] Universal Robots SL. [Consulta: 20 noviembre 2022] Disponible en: https://www.universal-robots.com/articles/ur/interface-communication/tcpip-socket-communication-via-urscript/

[20] *OVERVIEW OF CLIENT INTERFACES* [en línea] Universal Robots SL. [Consulta: 20 noviembre 2022] Disponible en: https://www.universal-robots.com/articles/ur/interface-communication/overview-of-client-interfaces/

[21] *scipy.spatial.transform.Rotation — SciPy v1.10.1 Manual* [en línea] SciPy Org. [Consulta: 03 marzo 2023]. Disponible en:

https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.transform.Rotation.html#scipy.spatial.transform.Rotation

[22] *Raspberry Pi Documentation - Camera software* [en línea]. Raspberry Org. [Consulta: 02 septiembre 2022]. Disponible en: https://www.raspberrypi.com/documentation/computers/camera_software.html#common-command-line-options

[23] Patricio Moracho. *Mostrar imagen escalada con OpenCV imshow()* [en línea]. Stack Overflow, 18 agosto 2021. [Consulta: 02 septiembre 2022]. Disponible en: https://es.stackoverflow.com/questions/477800/mostrar-imagen-escalada-con-opencv-imshow

[24] Avram Plitch. *How to Set Up a Headless Raspberry Pi, No Monitor Needed* [en línea]. Tom's Hardware, 17 septiembre 2022. [Consulta: 12 agosto 2022]. Disponible en: https://www.tomshardware.com/reviews/raspberry-pi-headless-setup-how-to,6028.html

[25] Nilesh Vijayrania. *Camera Image Perspective Transformation to different plane using OpenCV* [en línea]. Medium, 20 diciembre 2020. [Consulta: 17 febrero 2023]. Disponible en: https://nilesh0109.medium.com/camera-image-perspective-transformation-to-different-plane-using-opencv-5e389dd56527

Anexo A: Matrices de transformación homogéneas

Una matriz homogénea representa la posición y orientación de un punto en un sistema de coordenadas tridimensional. Consta de dos partes: una matriz de rotación de 3x3 y un vector de traslación (figura A.1).

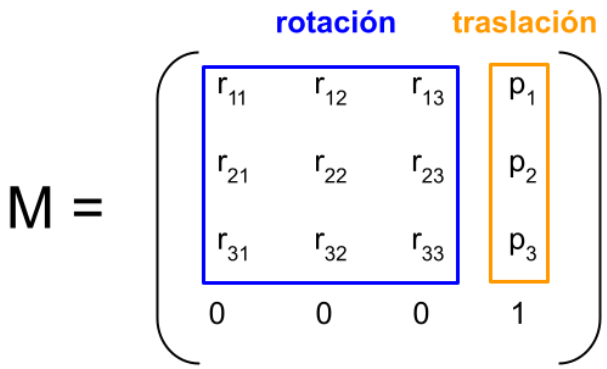


Figura A.1. Matriz homogénea

Para construir la matriz homogénea de un objeto en Python primero se crea una matriz identidad de 4x4 mediante la función “np.eye(4)”, a la cual se añade la matriz de rotación y el vector de traslación en sus respectivas posiciones. Es común que la parte de rotación se tenga en forma de vector, en ese caso se habrá de convertir primero a su forma matricial mediante el uso de la función “scipy.spatial.transform.Rotation.from_rotvec().as_matrix()” (se ha de tener instalada la librería scipy).

Su principal uso consiste en transformar las coordenadas de un objeto desde un sistema de referencia a otro. Por ejemplo, se tienen tres sistemas de referencia: A, B y C. Se conocen las coordenadas del sistema A respecto B (matriz homogénea: MA2B) y del sistema B respecto C (matriz homogénea: MB2C). Se pueden conocer las coordenadas de A respecto C (matriz homogénea: MA2C) multiplicando matricialmente las matrices MA2B y MB2C, tal como muestra la figura A.2.

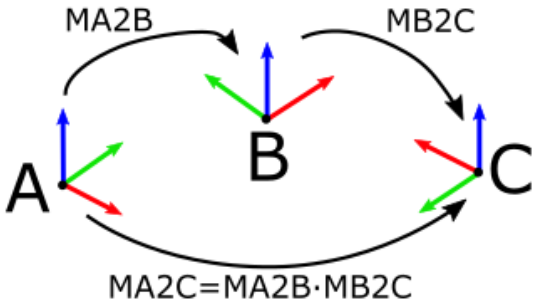


Figura A.2. Transformación de un sistema de referencia a otro

En varios puntos de este proyecto es necesario invertir una transformación entre dos sistemas de referencia. Por ejemplo, se tiene la transformación A2B que transforma un punto expresado en el sistema de referencia A al sistema de referencia B y se necesita la transformación inversa B2A que transforma un punto expresado en el sistema de referencia B al sistema de referencia A.

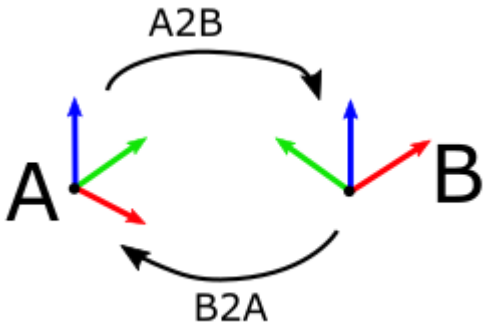


Figura A.3. Transformaciones inversas entre dos sistemas de referencia.

Para realizar esta conversión se necesita trabajar con la matriz homogénea de la transformación A2B (MA2B). Una vez conseguida dicha matriz se utiliza la función “numpy.linalg.inv()” para calcular su inversa multiplicativa y así obtener la matriz homogénea de la transformación B2A (MB2A). De esta matriz se pueden extraer sus respectivos vector de traslación y matriz de rotación para su posterior uso. En caso de que se necesite, se puede volver a convertir la matriz de rotación a vector rotacional mediante el uso de la función “scipy.spatial.transform.Rotation.from_matrix().as_rotvec()”.

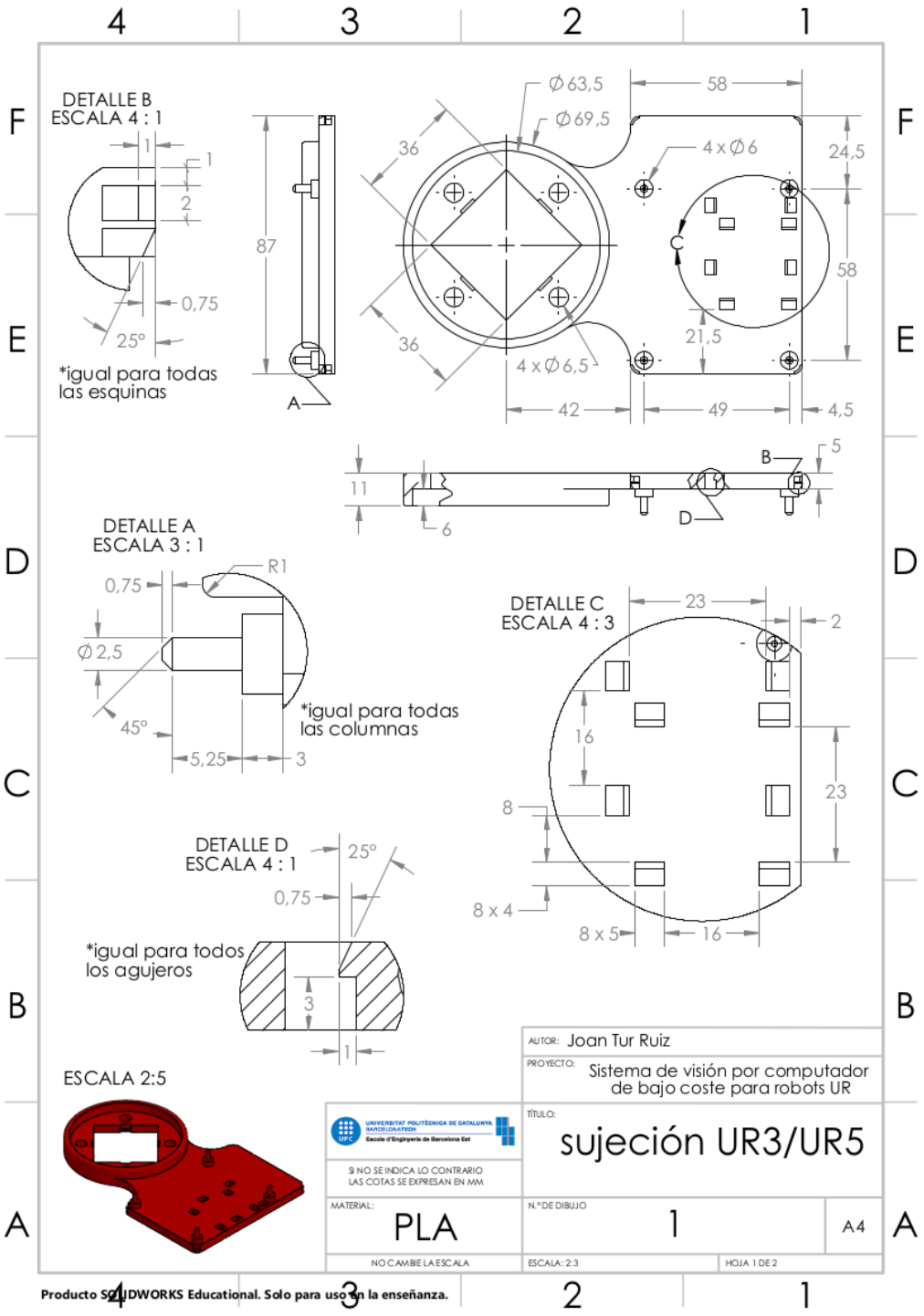
A continuación se muestra un fragmento de código que implementa este proceso de inversión:

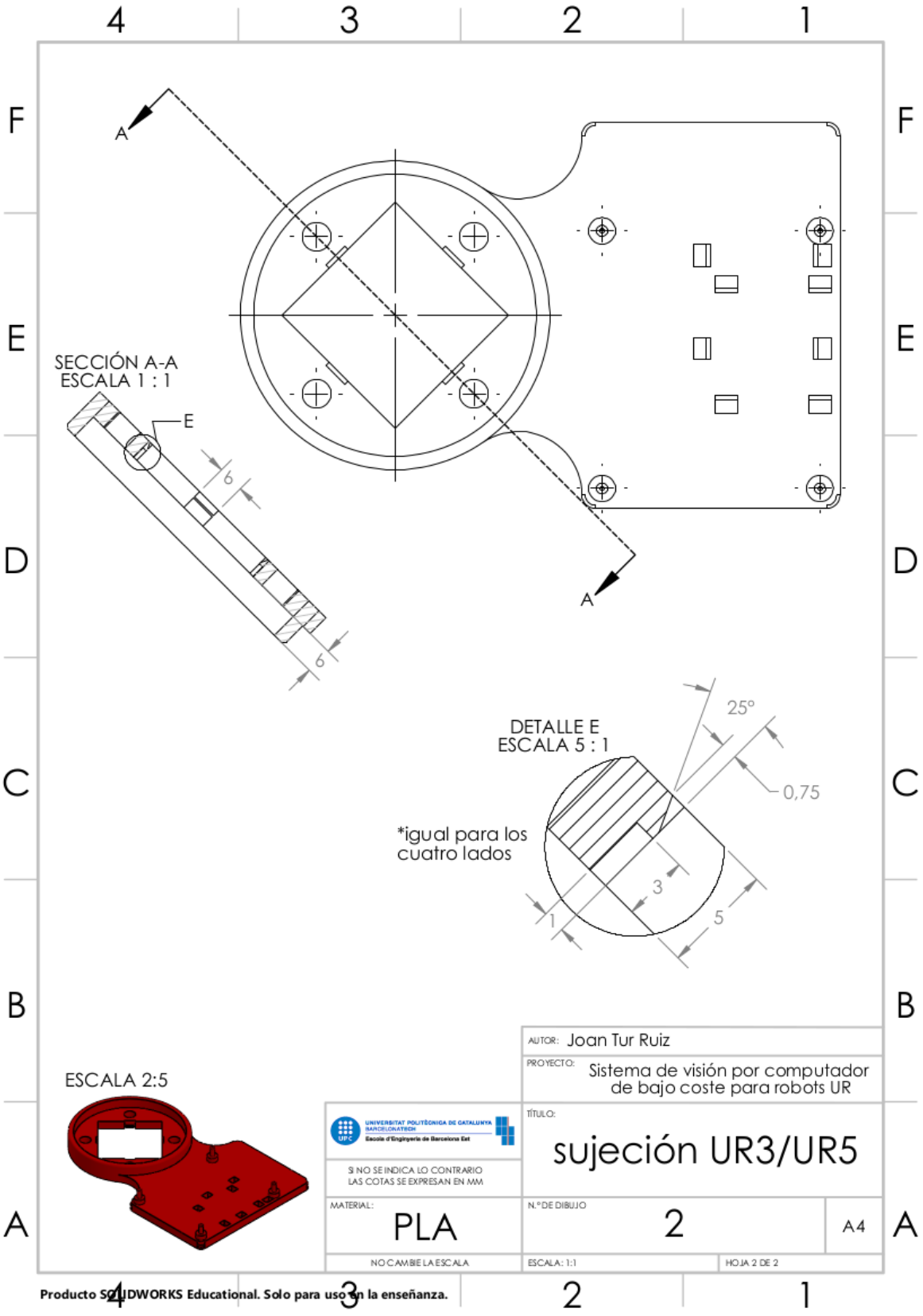
```
import numpy as np #importamos la librería numpy
```

```
from scipy.spatial.transform import Rotation as R #importamos la libreria scipy


Ma2b = np.eye(4) #creamos una matriz identidad de 4x4

rot = R.from_rotvec(Ra2b.T).as_matrix() #convertimos el vector de rotación a matriz

Ma2b[:3,:3] = rot #añadimos la  matriz de rotación

Ma2b[:3,3]= Ta2b.T #añadimos el vector de traslación, ya tenemos la matriz homogénea

Mb2a = np.linalg.inv(Ma2b) #invertimos la matriz, ya tenemos B2A

Tb2a = Mba[:3,3] #extraemos el vector de traslación

Rb2a = R.from_matrix(Mba[:3,:3]).as_rotvec() #convertimos matriz rotacional a vector
```

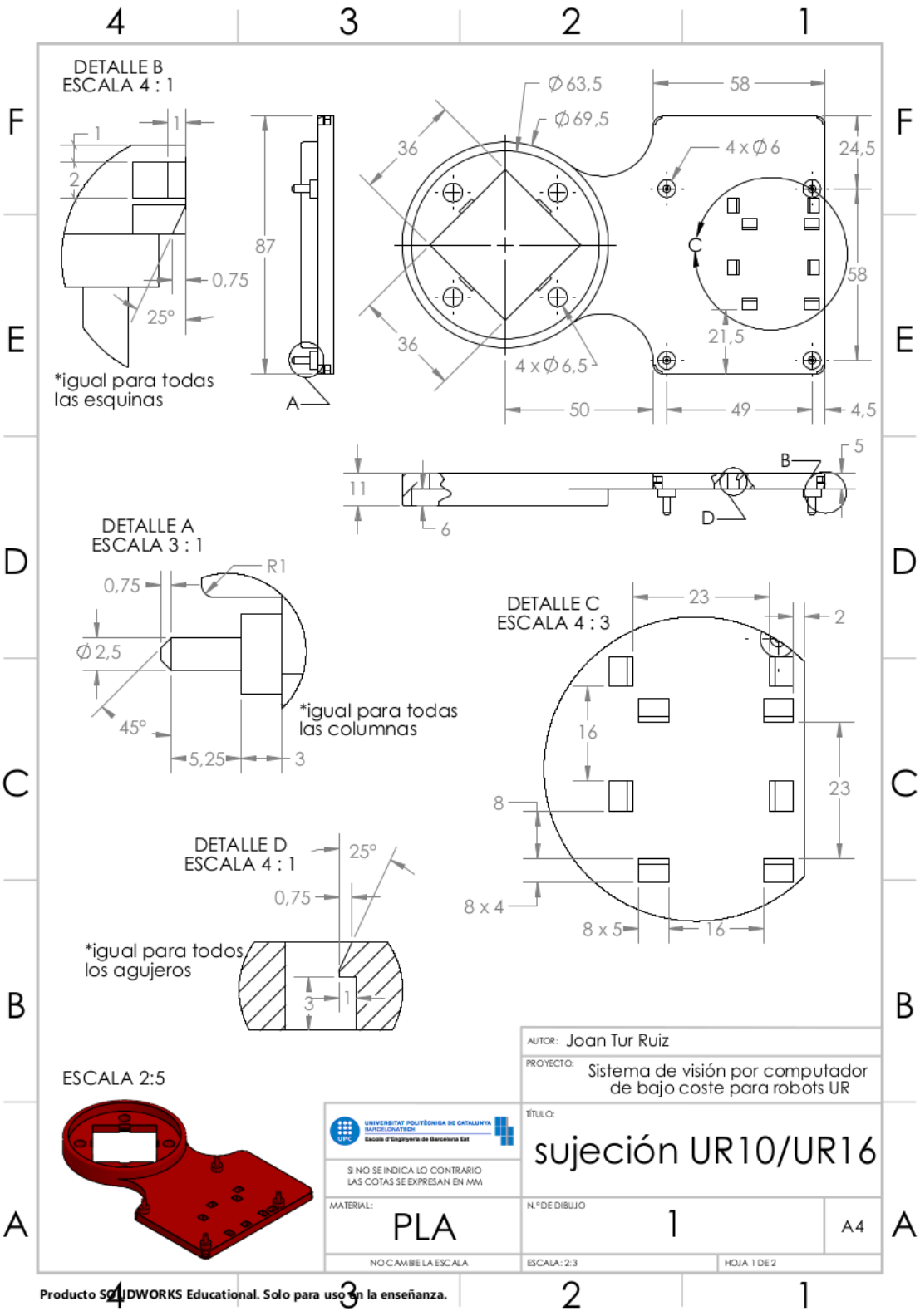
Anexo B: Planos CAD

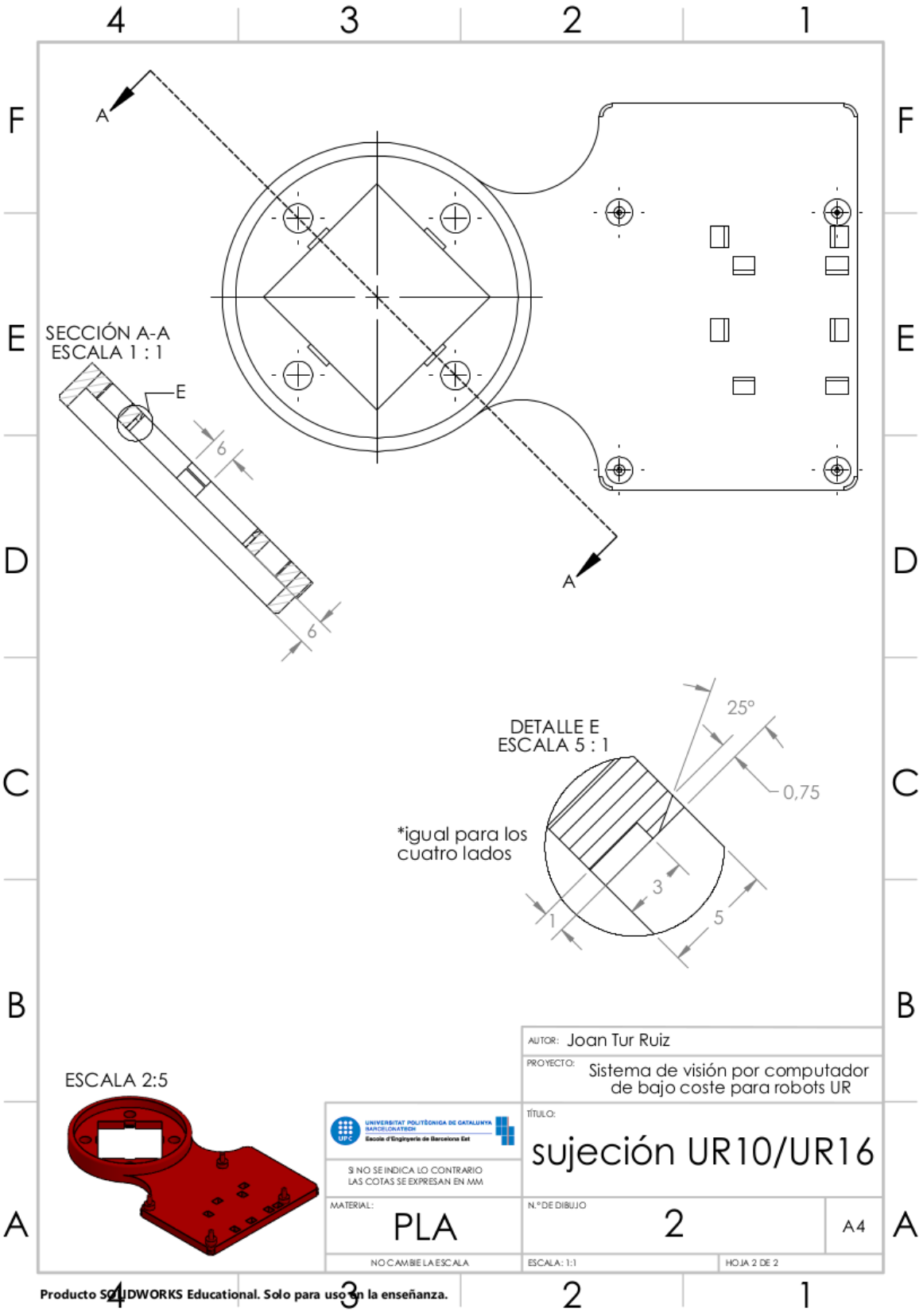
B1. Planos de la sujeción Raspberry-robot para UR3e y UR5e



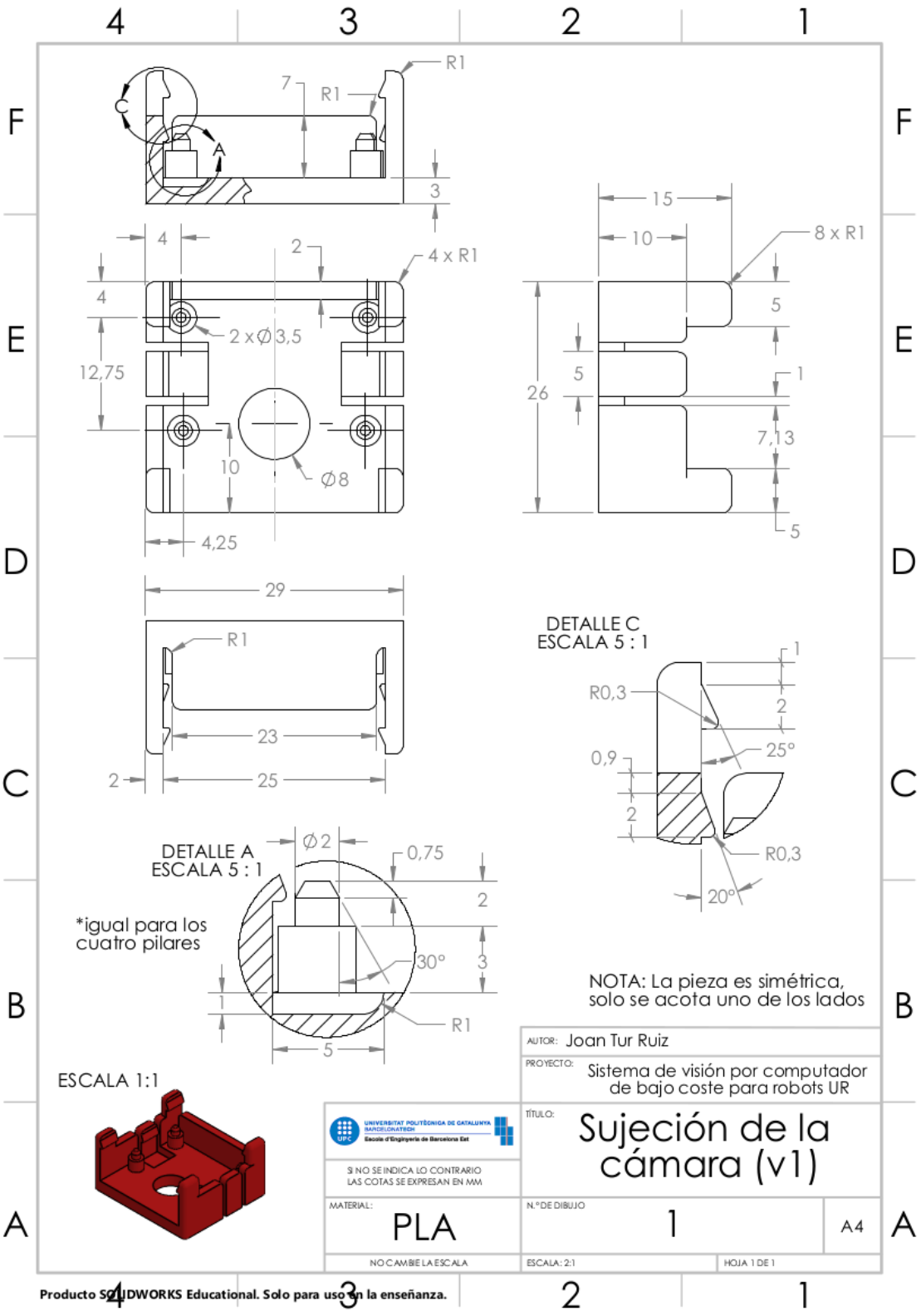

4
3
2
1
F
E
D
C
B
A
A
A
SECCIÓN A-A
ESCALA 1 : 1
E
6
6
DETALLE E
ESCALA 5 : 1
25°
0,75
*igual para los
cuatro lados
3
1
5
ESCALA 2:5
AUTOR: Joan Tur Ruiz
PROYECTO: Sistema de visión por computador de bajo coste para robots UR
TÍTULO: sujeción UR3/UR5
SI NO SE INDICA LO CONTRARIO LAS COTAS SE EXPRESAN EN MM
MATERIAL: PLA
N.º DE DIBUJO 2
A4
NO CAMBIE LA ESCALA
ESCALA: 1:1
HOJA 2 DE 2
Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.

B2. Planos de la sujeción Raspberry-robot para UR10e y UR16e

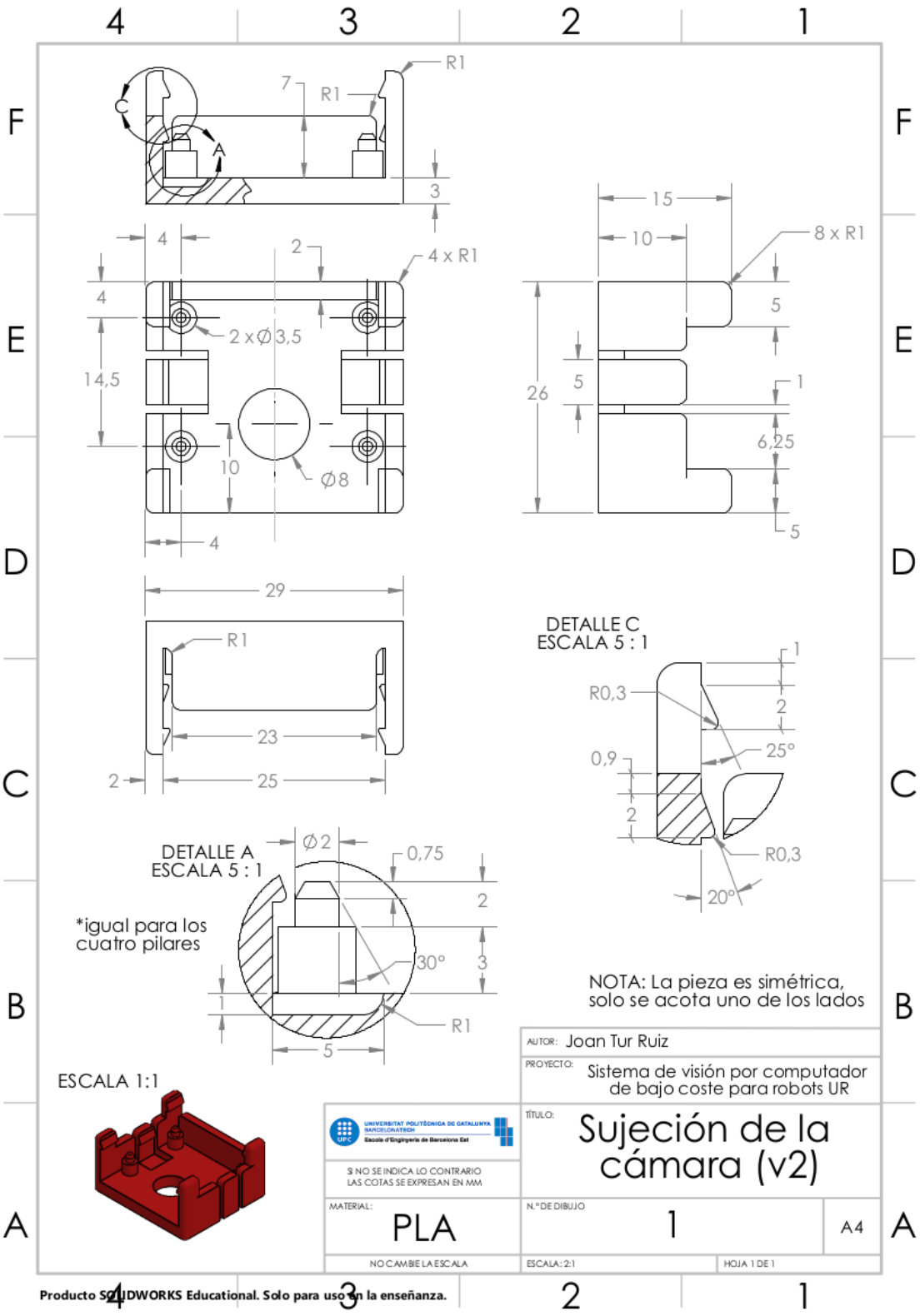



4
3
2
1
F
E
D
C
B
A
A
A
SECCIÓN A-A
ESCALA 1 : 1
E
6
6
DETALLE E
ESCALA 5 : 1
25°
0,75
*igual para los
cuatro lados
3
1
5
ESCALA 2:5
AUTOR: Joan Tur Ruiz
PROYECTO: Sistema de visión por computador de bajo coste para robots UR
TÍTULO:
sujeción UR10/UR16
SI NO SE INDICA LO CONTRARIO LAS COTAS SE EXPRESAN EN MM
MATERIAL:
PLA
N.º DE DIBUJO
2
A4
NO CAMBIE LA ESCALA
ESCALA: 1:1
HOJA 2 DE 2
Producto SOLIDWORKS Educational. Solo para uso en la enseñanza.

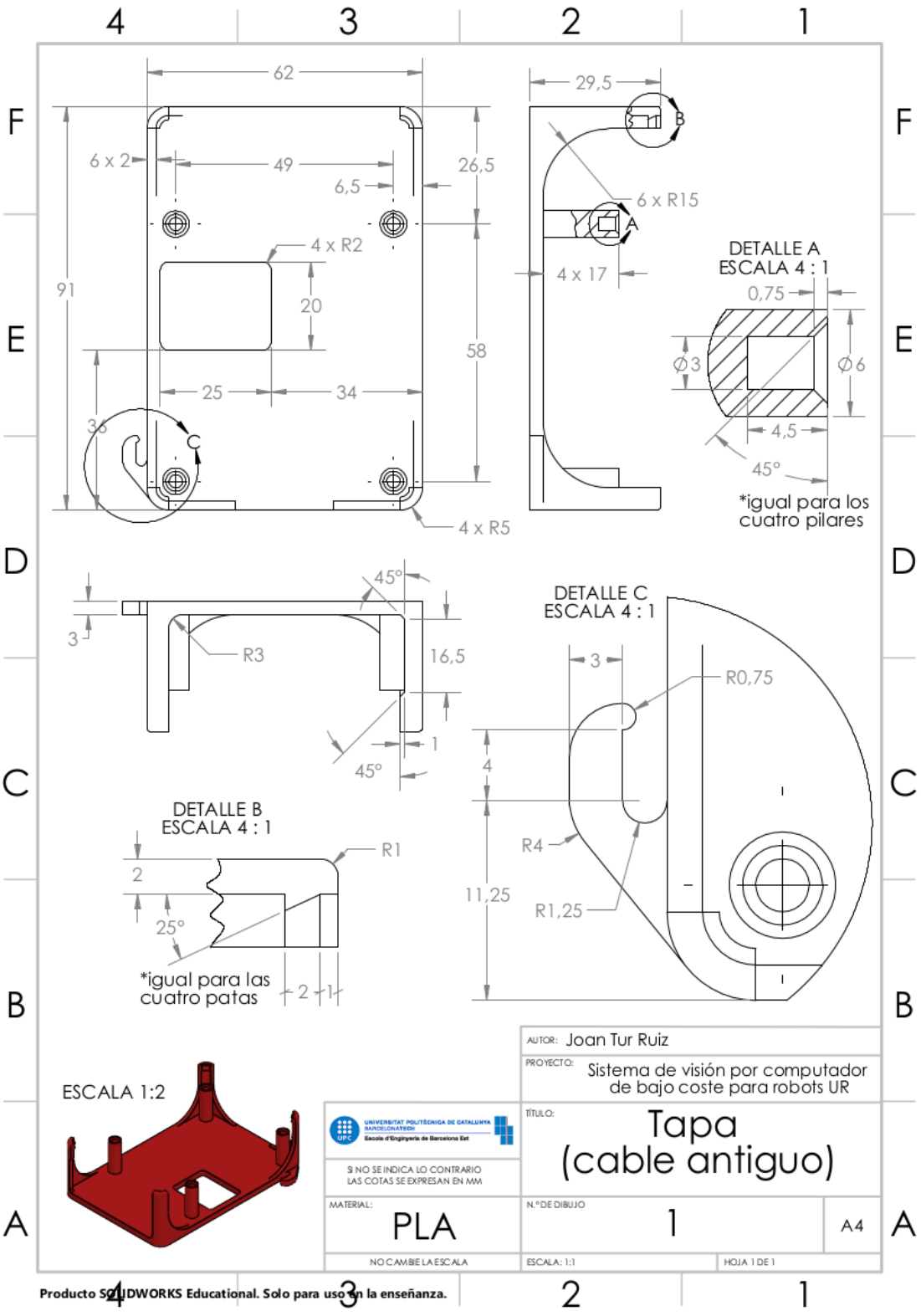
B3. Plano de la sujeción de la cámara (module v1)



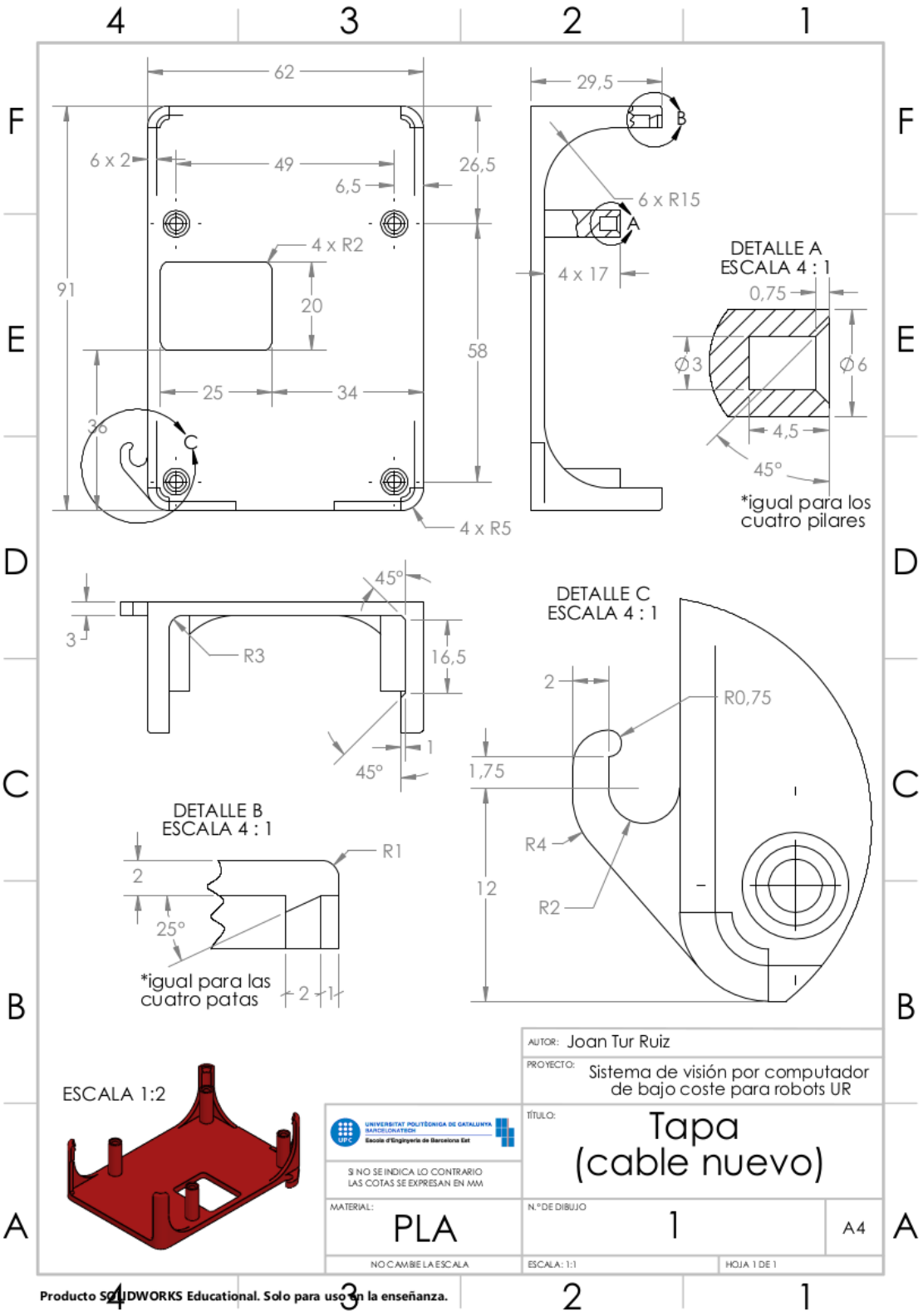
B4. Plano de la sujeción de la cámara (module v2)



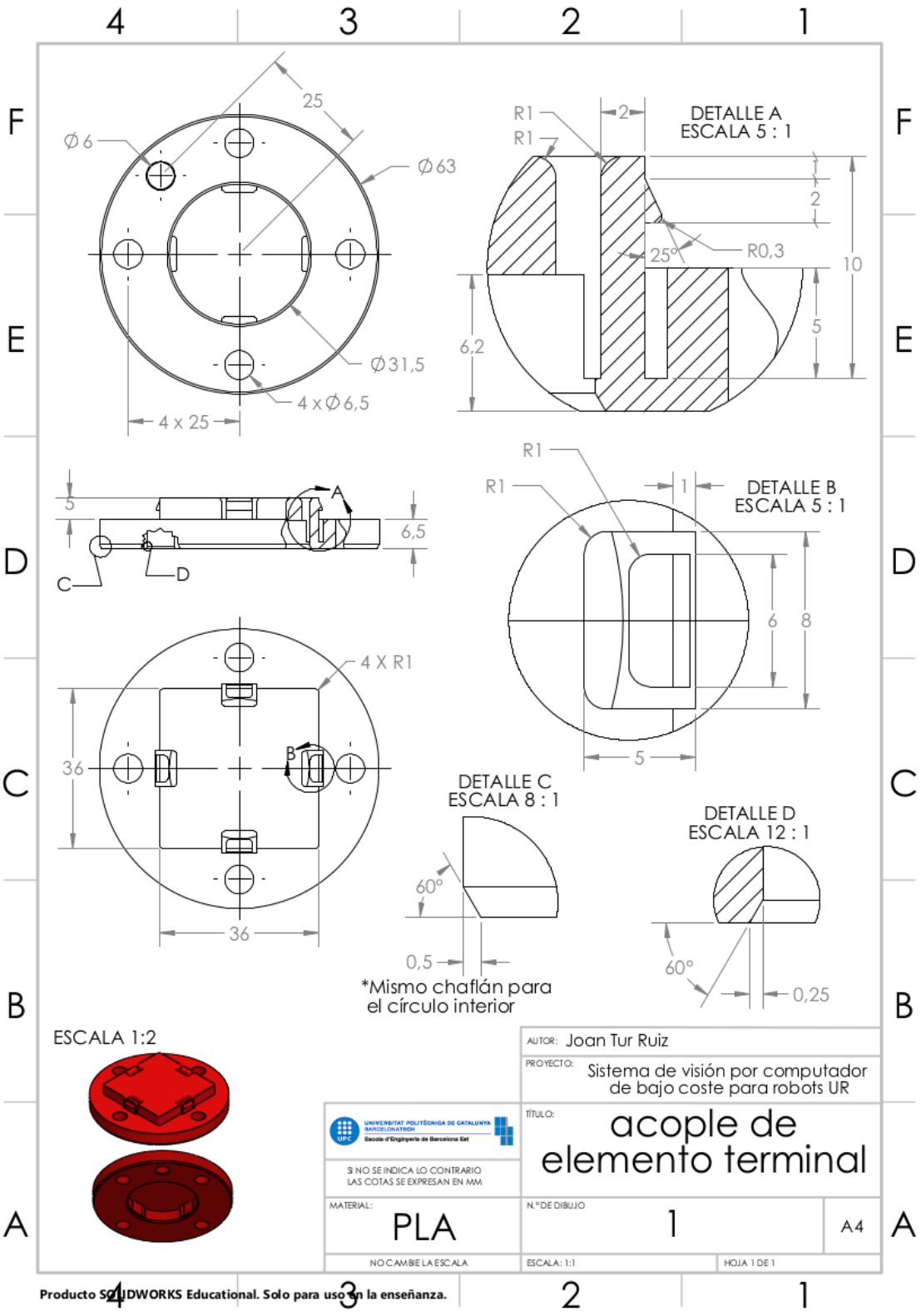
B5. Plano de la tapa protectora (cable antiguo)



B6. Plano de la tapa protectora (cable nuevo)



B7. Plano del acople para elemento terminal



Anexo C: Códigos Python

C1. Código de calibración óptica y geométrica

```
########## importamos las librerías necesarias ##########
import rtde.rtde as rtde
import rtde.rtde_config as rtde_config
import subprocess
import cv2 as cv
import numpy as np
import sys
from math import pi as Pi
from scipy.spatial.transform import Rotation as R
import socket
import time



########## definimos función para redimensionar la imagen ##########
def ResizeWithAspectRatio(image, width=None, height=None, inter=cv.INTER_AREA):
    dim = None
    (h, w) = image.shape[:2]

    if width is None and height is None:
        return image
    if width is None:
        r = height / float(h)
        dim = (int(w * r), height)
    else:
        r = width / float(w)
        dim = (width, int(h * r))

    return cv.resize(image, dim, interpolation=inter)
```

```
########## Preparamos parámetros para la calibración ##########

chessboardSize=(9,6) #Tamaño del tablero de ajedrez
frameSize=(2592,1944) #Tamaño de la imagenen en px

# Criterio de terminación
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# preparamos los puntos del tablero de ajedrez (coordenadas 3D)
objp = np.zeros((chessboardSize[0]*chessboardSize[1],3), np.float32)
objp[:,:2] = np.mgrid[0:chessboardSize[0],0:chessboardSize[1]].T.reshape(-1,2)


########## Configuramos comunicación RTDE #############
# Indicamos dirección:
ROBOT_HOST = "XX.XX.XX.XXX" # dirección ip del robot (se ha quitado por seguridad)
ROBOT_PORT = 30004 # puerto de la interfaz RTDE

#Cargamos archivo de configuración:
config_filename = "rtde_configuration.xml"
conf = rtde_config.ConfigFile(config_filename)
state_names, state_types = conf.get_recipe("state")

# Nos conectamos al robot:
con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT)
print("connecting to robot...")
con.connect()
print("connected")

# Mandamos los valores que queremos recibir
con.send_output_setup(state_names, state_types)

############## Inicializamos variables ###############
```

```
Tg2b=[] #vectores de traslación gripper-base
Rg2b=[] #vectores de rotación gripper-base
Tw2c=[] #vectores de traslación plano de trabajo-camara
Rw2c=[] #vectores de rotación plano de trabajo-camara
n=0 #contador de capturas

#Matrices para guardar los puntos de las esquinas del tablero:
objpoints = [] #puntos 3D en el mundo real
imgpoints = [] #puntos 2D en la imagen

################################################################################
###############
######### Capturar imagenes del patrón de calibración junto con las posiciones del
robot  #########
################################################################################
###############

# Bucle para capturar 10 imágenes (se empieza en la posición 0):
while n<=9:
      subprocess.call("libcamera-jpeg -t 1 -o testpy.jpg", shell=True) #Capturar
imagen
    img = cv.imread("testpy.jpg") #Cargar imagen
    gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) #convertir imagen a escala de grises

    # Buscamos las esquinas del tablero de ajedrez (método rápido):
    ret, corners = cv.findChessboardCorners(gray, chessboardSize, None,
cv.CALIB_CB_FAST_CHECK)

    # Si se encuentra el tablero:
    if ret == True:
        #Volvemos a buscar las esquinas pero con más precisión:
        ret, corners = cv.findChessboardCorners(gray, chessboardSize, None)
        #refinamos las esquinas:
```

```
corners2 = cv.cornerSubPix(gray,corners, (11,11), (-1,-1), criteria)
#Dibujamos las esquinas sobre la imagen a color:
imgdcc= img
cv.drawChessboardCorners(imgdcc, chessboardSize, corners2, ret)
#Mostramos la imagen al usuario:
resize = ResizeWithAspectRatio(imgdcc, width=1000) # Redimensionar
cv.imshow("calib " + str(n), resize) #visualizar

#Esperamos que el usuario pulse una tecla:
k = cv.waitKey(0)
cv.destroyWindow("calib " + str(n)) #cerramos la imagen

#Si la captura nos parece correcta (se ha pulsado enter) la guardamos:
if k == ord("\r") or k== ord("\n"):
    #Cargamos de nuevo la imagen (la otra tiene las esquinas dibujadas):
    img = cv.imread("testpy.jpg")
    #guardar imagen como calirobo0, calirobo1, ...:
    cv.imwrite("calirobo"+str(n)+".png", img)

    # Guardamos los puntos de las esquinas para la calibración óptica:
    objpoints.append(objp) #puntos 3D
    imgpoints.append(corners2) #puntos 2D sobre la imagen

    # Recibimos la posición del robot:

    # empezamos sincronización de datos (RTDE):
    if not con.send_start():
        sys.exit()
    print("getting TCP pose...")
    state = con.receive() #Recibimos los datos
    act_TCP = state.actual_TCP_pose #leemos la posición
    con.send_pause() #Pausamos sincronización
    pos=np.array(act_TCP) #guardamos la posición en una matriz numpy
    print("posición del robot: ",pos)
```

```
            #Guardamos el vector de traslación y de rotación de la posición del robot:
              Tg2b.append(pos[:3].T)
              Rg2b.append(pos[3:].T)

              n+=1 #sumamos al contador

    #Si no encuentra el tablero se avisa al usuario:
    else:
        #convertimos a RGB para poner texto en color:
        gray = cv.cvtColor(gray, cv.COLOR_GRAY2BGR)
        # escribimos el texto en la imagen:
        cv.putText(gray, 'chessboard', (200,900), cv.FONT_HERSHEY_SIMPLEX, 12, (0,
60, 255), 35, cv.LINE_AA)
        cv.putText(gray, 'not found', (200,1300), cv.FONT_HERSHEY_SIMPLEX, 12, (0,
60, 255), 35, cv.LINE_AA)
        # Redimensionamos y visualizamos la imagen:
        resize = ResizeWithAspectRatio(gray, width=1000) # Redimensionar
        cv.imshow("CHESSBOARD NOT FOUND " + str(n), resize) #Visualizar

        # Esperamos a que el operario pulse una tecla y cerramos la imagen:
        k = cv.waitKey(0)
        cv.destroyAllWindows()

# Guardamos los vectores de traslación y rotación de todas las posiciones de robot:
np.save("Tg2b",Tg2b)
np.save("Rg2b",Rg2b)

######################################################################
##############   Realizamos la calibración óptica  ##############
######################################################################

#Esta calibración calcula y devuelve:
```

```
# - los parámetros intrínsecos de la camara en una matriz (mtx)
# - los coeficientes de distorsión (dist)
# - vector de traslación del plano respecto la cámara para cada captura (tvecs)
# - vector de rotación del plano respecto la cámara para cada captura (rvecs)

ret, mtx, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints, frameSize,
None, None)

#guardamos los parámetros de la calibración óptica:
np.savez("calibration",mtx=mtx,dist=dist,rvecs=rvecs,tvecs=tvecs)



######################################################################
#############   Realizamos la calibración geométrica   ##############
######################################################################

print("Calculando calibración geométrica... ")

#####   Calculamos la posición del tablero respecto la cámara en cada imagen   #####

# cargamos los vectores de traslación y rotación de las posiciones de robot (gripper
a base):
Tg2b=np.load("Tg2b.npy")
Rg2b=np.load("Rg2b.npy")

# Necesitamos la transformación base a gripper,
# no gripper a base (posición de robot), hacemos la inversión:

Rb2g=[] #creamos variable para guardar vectores de rotación base a gripper
Tb2g=[] #creamos variable para guardar vectores de traslación base a gripper

for i in range(len(Tg2b)): #Para de cada una de las capturas:
    Mg2b = np.eye(4) #creamos matriz identidad
```

```
    rot = R.from_rotvec(Rg2b[i].T).as_matrix() #convertimos a matriz de rotación
    Mg2b[:3,:3] = rot #añadimos matriz de rotación
    Mg2b[:3,3] = Tg2b[i].T #añadimos vector de traslación
    Mb2g = np.linalg.inv(Mg2b) #hacemos inversión
    # Extraemos los vectores de traslación y rotación:
    Tb2g.append(Mb2g[:3,3])
    Rb2g.append(R.from_matrix(Mb2g[:3,:3]).as_rotvec())

#Cargamos los parámetros de calibración óptica de la camara
with np.load('calibration.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]

#Para de cada una de las capturas:
for i in range(len(Rb2g)):
    # Cargamos la captura:
    fname = "calirobo" + str(i) +".png"
    img = cv.imread(fname)

    gray = cv.cvtColor(img,cv.COLOR_BGR2GRAY) #convertimos a escala de grises

    #buscamos las esquinas del tablero:
ret,corners=cv.findChessboardCorners(gray,(chessboardSize[0],chessboardSize[1]),None
)

    #Si se encuentra el tablero:
    if ret == True:
        #Refinamos las esquinas
        corners2 = cv.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)

        # Encontramos los vectores de traslación y rotación
        # del plano de trabajo respecto la cámara:
        ret,rvecs, tvecs = cv.solvePnP(objp, corners2, mtx, dist)
```

```
        # multiplicamos los vectores de traslación por
        # la medida de un cuadro del tablero (24.5 mm):
        tvecs=tvecs*0.0245

        #guardamos los vectores de traslación y rotación
        Tw2c.append(tvecs)
        Rw2c.append(rvecs)

        # Si se trata de la primera imagen desde la posición de captura:
        if i==0:
            #Calculamos matriz de homografía:
            objpscaled=objp*0.0245 #puntos 3D de las esquinas del tablero en metros
            H, status = cv.findHomography(corners2, objpscaled) #matriz homografía
            np.save("H_cp",H) #la guardamos para su uso en otros programas

    #si no se encuentra el tablero:
    # *debería encontrarse, ya que las capturas son las mismas que antes, pero puede
    # haber casos excepcionales en los que no sea así
    else:
        #Eliminamos esos vectores del conjunto
        del Tb2g[i]
        del Rb2g[i]
        print("No se pudo encontrar tablero en la imagen calirobo" + str(i))


######## Calculamos la transformación del plano de trabajo respecto a la base #######

#La siguiente función devuelve los vectores de rotación y traslación de
#las transformaciones base-plano de trabajo y gripper-cámara:
Rb2w, Tb2w, Rg2c,Tg2c =cv.calibrateRobotWorldHandEye(Rw2c, Tw2c, Rb2g, Tb2g)


# Necesitamos la transformación plano de trabajo a base (w2b), hacemos la inversión:
Mb2w = np.eye(4) #creamos matriz identidad
```

```
Mb2w[:3,:3] = Rb2w #añadimos matriz de rotación
Mb2w[:3,3] = Tb2w.T #añadimos vector de traslación
Mw2b = np.linalg.inv(Mb2w) #hacemos la inversión
#extraemos los vectores de traslación y rotación:
Tw2b=Mw2b[:3,3]
Rw2b=R.from_matrix(Mw2b[:3,:3]).as_rotvec()

print("Traslación WP respecto base del robot: ")
print(Tw2b)
print("Traslación del TCP respecto la camara: ")
print(Tg2c)

# Guardamos los vectores de traslación y rotación del plano de
# trabajo respecto la base para su posterior uso en otros programas:
np.save("Tw2b",Tw2b)
np.save("Rw2b",Rw2b)
```

C2. Código de obtención de posición “pick” relativa al objeto

```
########## importamos las librerías necesarias ##########
import rtde.rtde as rtde
import rtde.rtde_config as rtde_config
import numpy as np
import cv2 as cv
import glob
from math import pi as Pi
from math import atan2, cos, sin, sqrt, radians
from scipy.spatial.transform import Rotation as R
import socket
import time
import subprocess
import sys
```

```
########## definimos función para redimensionar la imagen ##########
def ResizeWithAspectRatio(image, width=None, height=None, inter=cv.INTER_AREA):
    dim = None
    (h, w) = image.shape[:2]

    if width is None and height is None:
        return image
    if width is None:
        r = height / float(h)
        dim = (int(w * r), height)
    else:
        r = width / float(w)
        dim = (width, int(h * r))

    return cv.resize(image, dim, interpolation=inter)


########## Cargamos calibración óptica de la cámara ###########
with np.load('calibration.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]


########## Configuramos comunicación RTDE ############
# Indicamos dirección:
ROBOT_HOST = "XX.XX.XX.XXX" # dirección ip del robot (se ha quitado por seguridad)
ROBOT_PORT = 30004 # puerto de la interfaz RTDE

#Cargamos archivo de configuración:
config_filename = "rtde_configuration.xml"
conf = rtde_config.ConfigFile(config_filename)
state_names, state_types = conf.get_recipe("state")
```

```
# Nos conectamos al robot:
con = rtde.RTDE(ROBOT_HOST, ROBOT_PORT)
print("connecting to robot...")
con.connect()
print("connected")

# Mandamos los valores que queremos recibir
con.send_output_setup(state_names, state_types)



#########################################################################
####################   CAPTURA Y PROCESADO DE IMAGEN  ###################
#########################################################################

#Entramos en un bucle en el que se van capturando imagenes hasta que el
#operario esté satisfecho con una y confirme que es correcta:

ok=0
while ok==0:
    #Capturar imagen, fijamos balance de blancos por tener un fondo amarillo:
       subprocess.call("libcamera-jpeg  -t  1  -o  testpy.jpg  --awbgains  1.2,1.9",
shell=True)
    #Cargar imagen:
    img = cv.imread("testpy.jpg")



    ###### BINARIZACIÓN POR COLOR ######

    #convertimos a HSV para aislar la tonalidad de la saturación y el brillo:
    hsv=cv.cvtColor(img, cv.COLOR_BGR2HSV)

    # definimos el umbral del color de fondo:
    lower_range = np.array([26,0,100])
```

```
upper_range = np.array([36,255,255])

#Creamos una máscara (imagen binaria) con este umbral:
mask = cv.bitwise_not(cv.inRange(hsv, lower_range, upper_range))
# Si el umbral se creara con el color del objeto
# se tendría que invertir la imagen binaria:
#mask = cv.bitwise_not(mask)

# Aplicamos una operación morfológica de apertura y cierre:
#elemento estructural: círculo
kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE,(10,10))
mask = cv.morphologyEx(mask, cv.MORPH_OPEN, kernel) # Morphology open
mask = cv.morphologyEx(mask, cv.MORPH_CLOSE, kernel) # Morphology close


######## CLASIFICACIÓN DE OBJETOS ##########

#Encontramos los contornos en la imagen:
contours,hierarchy=cv.findContours(mask, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

#Buscamos el contorno del objeto que nos interesa:

obj=contours[0] #inicializamos variable del contorno del objeto
for cnt in contours: #Para cada contorno...
    area=cv.contourArea(cnt) #calculamos area
    if area>8550: #descartamos la pinza
        obj=cnt #guardamos el controno del objeto
        break

#Distinguimos entre cuadrado y destornillador:

#*Si es la primera vez que se enseña el objeto esta parte debe
# comentarse y reemplazarse por uno de los siguientes comados:
# objeto="cuad" # objeto="dest"
```

```
#Cargamos los contornos del cuadrado y el destornilaldor:
cnt_dest=np.load("cnt_destornillador.npy")
cnt_cuad=np.load("cnt_cuadrado.npy")
#Calculamos la similitud con cada uno:
sim_cuad = cv.matchShapes(cnt_cuad,obj,1,0.0)
sim_dest = cv.matchShapes(cnt_dest,obj,1,0.0)

#Segun la similitud escogemos cuadrado o destornillador:
if sim_cuad<0.1:
    objeto="cuad"
    print( "cuadrado" )
    print( "similarity:", sim_cuad )
elif sim_dest<1:
    objeto="dest"
    print( "destornillador" )
    print( "similarity:", sim_dest )
else: print("ninguno")

#mostramos su area para poder crear un umbral en el programa de la aplicación:
print("area: ", cv.contourArea(obj))

#Visualizamos la imagen original con el contorno dibujado
cv.drawContours(img, obj, -1 , (0,255,0), 20) #dibujamos contorno
resize2 = ResizeWithAspectRatio(img, width=740) # Redimensionamos la imagen
cv.imshow("contorno", resize2) # Visualizamos la imagen

# Esperamos a que el operario pulse una tecla:
k = cv.waitKey(0)

# Si el operario pulsa enter es que el objeto se ha detectado correctamente:
if k == ord("\r") or k== ord("\n"):
    #cv.destroyWindow("contours")
```

```
        #Según el objeto guardamos su contorno para usarse en otros programas:
        if objeto=="cuad":
            np.save("cnt_cuadrado",obj)
        elif objeto=="dest":
            np.save("cnt_destornillador",obj)
        ok=1 #salimos del bucle

    # Si pulsa cualquier otra tecla es que no
    # se ha detectado bien y hay que volver a capturar

cv.destroyAllWindows() #cerramos la imagen


######################################################################
###################   CÁLCULO DE POSICIONES   ####################
######################################################################

#Calculamos el centroide del objeto en la imagen:
M = cv.moments(obj) # Calculamos momentos
cx = int(M['m10']/M['m00']) #Calculamos centroide.x
cy = int(M['m01']/M['m00']) #Calculamos centroide.y
img_coord=[cx, cy] #guardamos coordenadas

##### Calculamos la rotación del objeto en la imagen: #####

#obtenemos mínimo rectángulo que envuelve al contorno:
rect = cv.minAreaRect(obj)
(xr,yr)=rect[0] #extraemos el centro del rectángulo en la imagen
(w, h)=rect[1] #extraemos anchura y altura del rectángulo

#Calculamos relación de aspecto:
if h>w: aspect_ratio = float(w)/h
else: aspect_ratio = float(h)/w
#print("aspect_ratio: ", aspect_ratio)
```

```
#obtenemos elipse que mejor se ajusta al contorno:
# la función da el centro de la elipse (xe,ye), la longitud de sus ejes
# (Ma, ma) y el ángulo del eje mayor respecto la vertical (angle)
(xe,ye),(MA,ma),angle = cv.fitEllipse(obj)

# Calculamos la distancia entre el centroide y el centro del rectángulo:
dcr= sqrt((xr-cx)**2+ (yr-cy)**2)

# Si los dos centros están muy cerca el objeto es simétrico y la dirección es irrelevante,
# si la relación de aspecto es próxima a 1 el objeto no es alargado y la elipse no se calcula bien,
# En caso de que cualquiera de estas dos condiciones se cumpla usaremos el ángulo del rectángulo.
# Si no se cumple ninguna calcularemos la dirección en la que apunta el objeto:
if dcr>10 and aspect_ratio<0.8: #no es simétrico y es alargado
    # calculamos un punto en cada dirección sobre el
    # eje de la elipse, equidistanciados del centroide:
    (p1x,p1y) = (cx+int(dcr*sin(radians(angle))), cy-int(dcr*cos(radians(angle))))
(p2x,p2y)=(cx+int(dcr*sin(radians(angle-180))), cy-int(dcr*cos(radians(angle-180))))

    # Calculamos la distancia de cada punto al centro del rectángulo:
    dp1r= sqrt((xr-p1x)**2+ (yr-p1y)**2)
    dp2r= sqrt((xr-p2x)**2+ (yr-p2y)**2)

    #según que punto esté más cerca sumamos o no 180º al ángulo de la elipse:
    if dp2r<dp1r:
        angle=angle+180

else: # es simétrico o no alargado
    angle=rect[2] #usamos el ángulo del rectángulo

# Dibujamos el centroide y la dirección en la imagen
```

```
cv.circle(img,(cx,cy), 20, (0,0,255), -1)
cv.line(img,(cx,cy),(cx+int(300*sin(radians(angle))),cy-int(300*cos(radians(angle)))
,(0,0,255),10)
resize2 = ResizeWithAspectRatio(img, width=740) # Redimensionar la imagen
cv.imshow('Output Image', resize2) #Visualizamos la imagen

# Mostramos coordenadas y orientación del objeto sobre la imagen
print("U: " + str(cx) + ";  V: " + str(cy) + ";  O: " + str(angle))

# Calculamos traslación del objeto respecto al
# plano de trabajo, usando la matriz de homografía:
H=np.load("H_cp.npy") # Cargamos matriz de homografía
x, y, w = H @ np.array([[*img_coord, 1]]).T
xx=x/w
yy=y/w
To2w=np.array([xx[0], yy[0], 0]) #Traslación del objeto respecto al plano de trabajo

# creamos la orientación del objeto respecto al plano de trabajo,
# dado que está sobre el plano la rotación en X e Y será 0:
Ro2w= np.array([0,0, angle])

#Mostramos la posición del objeto sobre el plano de trabajo:
position=np.concatenate((To2w.reshape(1, -1), Ro2w.reshape(1, -1)), axis = 1)
print("posicion objeto sobre plano: ", position)

#Creamos la matriz homogénea de la transformación del objeto al plano de trabajo:
Mo2w= np.eye(4) #creamos matriz identidad
rot= R.from_euler("xyz", Ro2w, degrees=True).as_matrix() #matriz de rotación
Mo2w[:3,:3] = rot #añadimos matriz de rotación
Mo2w[:3,3] = To2w.T #añadimos vector de traslación

#Cargamos vectores de traslación y rotación del plano de trabajo respecto a la base:
Tw2b=np.load("Tw2b.npy") # Vector de traslación
Rw2b=np.load("Rw2b.npy") # Vector de rotación
```

```
#Creamos la matriz homogénea:
Mw2b = np.eye(4) #creamos matriz identidad
rot = R.from_rotvec(Rw2b).as_matrix() #Convertimos a matriz de rotación
Mw2b[:3,:3] = rot #añadimos matriz de rotación
Mw2b[:3,3] = Tw2b.T #añadimos vector de traslación

# obtenemos objeto respecto a la base multiplicando ambas matrices homogéneas:
Mo2b=Mw2b@Mo2w # Matriz homogénea del objeto respecto a la base del robot
To2b=Mo2b[:3,3] #extraemos vector de traslación
Ro2b=R.from_matrix(Mo2b[:3,:3]).as_rotvec() #extraemos vector de rotación

# Mostramos posición del objeto respecto a la base
print("posicion objeto respecto base: ", To2b, R.from_rotvec(Ro2b).as_euler('xyz',
degrees=True))



########################################################################
############ Obtención de la posición relativa al objeto ##############
########################################################################

print("Mueva el robot a la posición relativa al objeto deseada, pulse enter cuando
esté en esa posición")

# Esperamos a que el operario mueva el robot a la posición deseada (pulsará enter):
ok=0
while ok==0:
    k = cv.waitKey(0)
    if k == ord("\r") or k== ord("\n"):
        cv.destroyAllWindows()
        ok=1
```

```
# Recibimos la posición del robot:

if not con.send_start(): # empezamos sincronización de datos (RTDE)
    sys.exit()
print("getting TCP pose...")
state = con.receive() #Recibimos los datos
act_TCP = state.actual_TCP_pose #leemos la posición
con.send_pause() #Pausamos sincronización
pos=np.array(act_TCP) #guardamos la posición en una matriz numpy
print("posición del robot: ",pos)

Tg2b = pos[:3].T #extraemos vector de traslación
Rg2b = pos[3:].T #extraemos vector de rotación

# creamos matriz homogénea gripper a base:
Mg2b = np.eye(4) #creamos matriz identidad
rot = R.from_rotvec(Rg2b.T).as_matrix() #Convertimos a matriz de rotación
Mg2b[:3,:3] = rot #añadimos matriz de rotación
Mg2b[:3,3] = Tg2b.T #añadimos vector de traslación

# la invertimos para obtener matriz base a gripper:
Mb2o=np.linalg.inv(Mo2b)

# multiplicando base a objeto por gripper a base obtenemos la matriz gripper a
# objeto, esta matriz representa la posición relativa que buscamos:
Mg2o = Mb2o@Mg2b
Tg2o= Mg2o[:3,3] #extraemos vector de traslación
Rg2o=R.from_matrix(Mg2o[:3,:3]).as_rotvec() #extraemos vector de rotación

#Mostramos el resultado:
print("gripper respecto a "+objeto+":", Tg2o, R.from_rotvec(Rg2o).as_euler('xyz',
degrees=True))
```

```
#guardamos la matriz homogénea para usarla en otros programas:
np.save("relpos_"+objeto, Mg2o)

con.disconnect() #cerramos la conexión RTDE
```

C3. Programa de la aplicación de ejemplo

```
########## importamos las librerías necesarias ##########
import numpy as np
import cv2 as cv
import glob
from math import pi as Pi
from math import atan2, cos, sin, sqrt, radians
from scipy.spatial.transform import Rotation as R
import socket
import time
import subprocess
import sys

########## definimos función para redimensionar la imagen ##########
def ResizeWithAspectRatio(image, width=None, height=None, inter=cv.INTER_AREA):
    dim = None
    (h, w) = image.shape[:2]

    if width is None and height is None:
        return image
    if width is None:
        r = height / float(h)
        dim = (int(w * r), height)
    else:
        r = width / float(w)
        dim = (width, int(h * r))
```

```
    return cv.resize(image, dim, interpolation=inter)

########## definimos función para recibir valores del robot ##########
def receive_from_robot():
     # Receive data from client
    data = connection.recv(1024)
    response = str(data, 'utf8')
    print('Response: ' + response)
    return response

########## Cargamos calibración óptica de la camara ###########
with np.load('calibration.npz') as X:
    mtx, dist, _, _ = [X[i] for i in ('mtx','dist','rvecs','tvecs')]


############ Configuramos comunicación socket ##############

# Creamos socket TCP/IP:
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

# Definimos dirección de la raspberry:
sHostIpAddress = "10.52.17.116" #ip de la raspberry
server_address = (sHostIpAddress, 50000) #usamos puerto 50000

# Asociamos la dirección al socket:
print('starting up on {} port {}'.format(*server_address))
sock.bind(server_address)

#Establecemos conexión con el robot:
sock.listen(1)
print('waiting for a connection')
connection, client_address = sock.accept() # Aceptamos cliente
```

```
print('connection from', client_address)

######################################################################
################## BUCLE PRINCIPAL DEL PROGRAMA ######################
######################################################################

while True:

    ##############################################################
    #############   CAPTURA Y PROCESADO DE IMAGEN   ##############
    ##############################################################

    #Entramos en un bucle en el que se van capturando imagenes hasta que el
    #operario esté satisfecho con una y confirme que es correcta:
    ok=0
    while ok==0:
        #Capturar imagen, fijamos balance de blancos por tener un fondo amarillo:
        subprocess.call("libcamera-jpeg -t 1 -o testpy.jpg --awbgains 1.2,1.9",
shell=True)
        img = cv.imread("testpy.jpg") #Cargar imagen

        ###### BINARIZACIÓN POR COLOR ######

        #convertimos a HSV para aislar la tonalidad de la saturación y el brillo:
        hsv=cv.cvtColor(img, cv.COLOR_BGR2HSV)

        # definimos el umbral del color de fondo:
        lower_range = np.array([26,0,100])
        upper_range = np.array([36,255,255])


        #Creamos una máscara (imagen binaria) con este umbral:
        mask = cv.bitwise_not(cv.inRange(hsv, lower_range, upper_range))
```

```
        #Si el umbral se creara con el color del objeto
        # se tendría que invertir la imagen binaria:
        #mask = cv.bitwise_not(mask)

        #Aplicamos una operación morfológica de apertura y cierre:
        #elemento estructural: círculo
        kernel = cv.getStructuringElement(cv.MORPH_ELLIPSE,(10,10))
        mask = cv.morphologyEx(mask, cv.MORPH_OPEN, kernel) # Morphology open
        mask = cv.morphologyEx(mask, cv.MORPH_CLOSE, kernel) # Morphology close

        # Si se quisiera usar la binarización clásica se puede descomentar el
        # siguiente fragmento de código y quitar el de la binarización por color:

        ######## BINARIZACIÓN CLASICA #########
        #     imgg=cv.cvtColor(img, cv.COLOR_BGR2GRAY)
        #
        #     blur = cv.GaussianBlur(imgg,(5,5),0)
        #     ret1,imgob = cv.threshold(imgg,0,255,cv.THRESH_BINARY+cv.THRESH_OTSU)
        #
        #     kernel = np.ones((15,15),np.uint8)
        #     imgob = cv.morphologyEx(imgob, cv.MORPH_OPEN, kernel) #Morphology open
        #     imgob =cv.morphologyEx(imgob,cv.MORPH_CLOSE, kernel) #Morphology close
        #
        #     resize1 = ResizeWithAspectRatio(imgob, width=740) # Resize by width OR
        #     cv.imshow("otsu_bin " + fname, resize1)


        ######## CLASIFICACIÓN DE OBJETOS ##########

        #Encontramos los contornos en la imagen:
            contours, hierarchy = cv.findContours(mask, cv.RETR_TREE,
cv.CHAIN_APPROX_SIMPLE)

        #Cargamos los contornos del cuadrado y el destornillador:
```

```
cnt_dest=np.load("cnt_destornillador.npy")
cnt_cuad=np.load("cnt_cuadrado.npy")

objeto= "" #creamos variable que indica que objeto es
vacio=True #creamos variable que indica si no se ha encontrado objeto
for cnt in contours: #Para cada contorno...
    #Calculamos la similitud con cuadrado y con destornillador:
    sim_cuad = cv.matchShapes(cnt_cuad,cnt,1,0.0)
    sim_dest = cv.matchShapes(cnt_dest,cnt,1,0.0)
    area = cv.contourArea(cnt)  #Calculamos

    # Según la similitud y si se encuentra dentro del rango de area
    # el objeto se reconocerá como cuadrado o como destornillador

    if sim_cuad<0.1 and area<210000 and area>170000:
        objeto= "cuad"
        print( "cuadrado" )
        print( "similitud:", sim_cuad )
        print("area: ", area)
        obj=cnt #guardamos el contorno del objeto
        vacio=False #se ha encontrado un objeto
        break
    elif sim_dest<1 and area<260000 and area>230000:
        objeto= "dest"
        print( "destornillador" )
        print( "similitud:", sim_dest )
        print("area: ", area)
        obj=cnt #guardamos el contorno del objeto
        vacio=False #se ha encontrado un objeto
        break

if vacio: #si no se ha encontrado ningún objeto...
    print("ninguno")
```

```
            print("sim_cuad: ",sim_cuad, " , sim_dest:" ,sim_dest)
            print("area: ", area)
            #Mostramos imagen con el texto "object not found":
               cv.putText(img, 'object', (200,900), cv.FONT_HERSHEY_SIMPLEX, 12, (0,
60, 255), 35, cv.LINE_AA)
               cv.putText(img, 'not found', (200,1300), cv.FONT_HERSHEY_SIMPLEX, 12,
(0, 60, 255), 35, cv.LINE_AA)
            resize = ResizeWithAspectRatio(img, width=1000) # redimensionamos imagen
            cv.imshow("OBJECT NOT FOUND", resize) #mostramos imagen
            k = cv.waitKey(0)

        else: #si SÍ se ha encontrado algún objeto...

            ########################################################################
            ###################   CALCULO DE POSICIONES   ##########################
            ########################################################################

            #Calculamos el centroide del objeto en la imagen:
            M = cv.moments(obj) # Calculamos momentos
            cx = int(M['m10']/M['m00']) #Calculamos centroide.x
            cy = int(M['m01']/M['m00']) #Calculamos centroide.y
            img_coord=[cx, cy] #guardamos coordenadas


            #obtenemos mínimo rectangulo que envuelve al contorno:
            rect = cv.minAreaRect(obj)
            (xr,yr)=rect[0] #extraemos el centro del rectangulo en la imagen
            (w, h)=rect[1] #extraemos anchura y altura del rectangulo

            #Calculamos relación de aspecto:
            if h>w: aspect_ratio = float(w)/h
            else: aspect_ratio = float(h)/w
            #print("aspect_ratio: ", aspect_ratio)
```

```
            #obtenemos elipse que mejor se ajusta al contorno:
            # la función da el centro de la elipse (xe,ye), la longitud de sus ejes
            # (Ma, ma) y el angulo del eje mayor respecto la vertical (angle)
            (xe,ye),(MA,ma),angle = cv.fitEllipse(obj)

            # Calculamos la distancia entre el centroide y el centro del rectángulo:
            dcr= sqrt((xr-cx)**2+ (yr-cy)**2)

            # Si los dos centros están muy cerca el
            # objeto es simétrico y la dirección es irrelevante,
            # si la relación de aspecto es próxima a 1 el
            # objeto no es alargado y la elipse no se calcula bien.
            # En caso de que cualquiera de estas dos condiciones
            # se cumpla usaremos el ángulo del rectángulo.
            # Si no se cumple ninguna calcularemos la
            # dirección en la que apunta el objeto:
            if dcr>10 and aspect_ratio<0.8: #no es simétrico y es alargado

                #calculamos un punto en cada dirección sobre el
                # eje de la elipse, equidistanciados del centroide:
                  (p1x,    p1y)    =    (cx    +    int(dcr*sin(radians(angle))),
cy-int(dcr*cos(radians(angle))))
                 (p2x,    p2y)    =    (cx    +    int(dcr*sin(radians(angle-180))),
cy-int(dcr*cos(radians(angle-180))))

                # Calculamos la distancia de cada punto al centro del rectángulo:
                dp1r= sqrt((xr-p1x)**2+ (yr-p1y)**2)
                dp2r= sqrt((xr-p2x)**2+ (yr-p2y)**2)

                # según que punto esté más cerca sumamos o no
                # 180º al ángulo de la elipse:
                if dp2r<dp1r:
                    angle=angle+180
```

```
        else: # es simétrico o con relación de aspecto unitaria
            angle=rect[2] #usamos el ángulo del rectángulo

        # Dibujamos el contorno, el centroide y la dirección sobre la imagen:
        cv.drawContours(img, obj, -1 , (0,255,0), 20)
        cv.circle(img,(cx,cy), 20, (0,0,255), -1)
          cv.line(img,         (cx,cy),         (cx+int(300*sin(radians(angle))),
cy-int(300*cos(radians(angle)))), (0,0,255),10)
        resize2 = ResizeWithAspectRatio(img, width=1000) # Redimensionamos
        cv.imshow('objeto:'+objeto, resize2) #Visualizamos la imagen



        # Calculamos traslación del objeto respecto al
        # plano de trabajo, usando la matriz de homografía:
        H=np.load("H_cp.npy") # Cargamos matriz de homografía
        x, y, w = H @ np.array([[*img_coord, 1]]).T
        xx=x/w
        yy=y/w
        #Traslación del objeto respecto al plano de trabajo:
        To2w=np.array([xx[0], yy[0], 0])
        # Creamos la rotación del objeto respecto al plano de trabajo,
        # dado que está sobre el plano la rotación en X e Y será 0:
        Ro2w= np.array([0,0, angle])

        # Creamos la matriz homogénea de la transformación
        # del objeto al plano de trabajo:
        Mo2w= np.eye(4) #creamos matriz identidad
        rot= R.from_euler("xyz", Ro2w, degrees=True).as_matrix()#matriz rotación
        Mo2w[:3,:3] = rot #añadimos matriz de rotación
        Mo2w[:3,3] = To2w.T #añadimos matriz de traslación

        #Cargamos vectores de traslación y rotación
        # del plano de trabajo respecto a la base:
```

```
            Tw2b=np.load("Tw2b.npy") # Vector de traslación
            Rw2b=np.load("Rw2b.npy") # Vector de rotación

            #Creamos la matriz homogénea::
            Mw2b = np.eye(4) #creamos matriz identidad
            rot = R.from_rotvec(Rw2b).as_matrix() #Convertimos a matriz de rotación
            Mw2b[:3,:3] = rot #añadimos matriz de rotación
            Mw2b[:3,3] = Tw2b.T #añadimos matriz de traslación

            # obtenemos objeto respecto a la base multiplicando
            #ambas matrices homogeneas:
            Mo2b=Mw2b@Mo2w #Matriz homogénea del objeto respecto a la base del robot
            To2b=Mo2b[:3,3] #extraemos vector de traslación
            Ro2b=R.from_matrix(Mo2b[:3,:3]).as_rotvec()#extraemos vector de rotación

            # Mostramos posición del objeto respecto a la base:
              print("posicion objeto respecto base: ",To2b,
R.from_rotvec(Ro2b).as_euler('xyz',degrees=True))

            # Esperamos a que el operario confirme que el objeto
            # se ha detectado correctamente (pulsará enter):
            k = cv.waitKey(0)
            if k == ord("\r") or k== ord("\n"):
                ok=1 # Salimos del bucle

    # Cargamos la posición relativa al objeto que se haya encontrado:
    if objeto=="cuad":
        Mg2o=np.load("relpos_cuad.npy")
    elif objeto=="dest":
        Mg2o=np.load("relpos_dest.npy")

    # Calculamos la posición objetivo multiplicando la matriz
    # objeto-base por la matriz de la posición relativa:
```

```
    Mtarget=Mo2b@Mg2o # Posición objetivo
    Ttarget=Mtarget[:3,3] # Extraemos vector de traslación
    Rtarget=R.from_matrix(Mtarget[:3,:3]).as_rotvec() # Extraemos vector de rotación

    #Mostramos la posición objetivo:
    Ptarget= np.concatenate((Ttarget, Rtarget), axis=None)
    print("posicion objetivo: ", Ttarget, R.from_rotvec(Rtarget).as_euler('xyz',
degrees=True))


    ######################################################################
    #############   ENVIAR POSICIÓN OBJETIVO AL ROBOT   #############
    ######################################################################

    # la posición a enviar tiene que tener este formato:
    # "(0.4, 0, 0.5, 0, -3.14159, 0)"

    # realizamos la conversión:
    target_coord = "("
    for ele in range(len(Ptarget)):
        target_coord += str(Ptarget[ele])
        if ele != 5 :
            target_coord += ", "
    target_coord += ")"

    #Enviamos la posición:
    connection.sendall(bytes(target_coord+"\n", 'utf8'))
    print("posicion enviada: ", target_coord)

    # Enviamos el código del objeto encontrado (1=cuadrado; 2=destornillador):
    if objeto=="cuad":
        obj_code="(1)"
    elif objeto=="dest":
        obj_code="(2)"
```

```
connection.sendall(bytes(obj_code+"\n", 'utf8'))

########## esperamos a que el robot acabe el ciclo #######

# entramos en un bucle esperando la señal del robot
resp=""
while len(resp)==0:
    resp=receive_from_robot()
# Una vez recibida la señal de fin de ciclo se vuelve al
#principio del bucle principal y se repite la ejecución


# si el robot ha enviado el string "closing_socket"
# cerramos las comunicaciones y paramos el programa:
if resp=="closing socket":
        connection.close()
        cv.destroyAllWindows()
        break

cv.destroyAllWindows() #cerramos la imagen
```