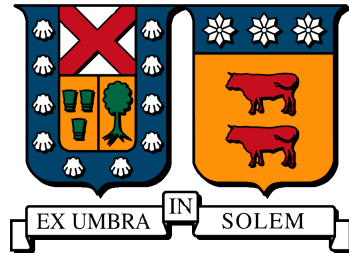


UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INGENIERÍA MECÁNICA
SANTIAGO-CHILE



**DESARROLLO DE ALGORITMO DE
CONTROL BASADO EN ESTEREOVISIÓN
PARA SIMULACIÓN DE AGARRE DE UN
OBJETO MEDIANTE UN BRAZO
ROBÓTICO**

ÁLVARO JAVIER ARAVENA PACHECO

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERÍA
CIVIL MECÁNICA**

PROFESOR GUÍA : DR. ING. DANILO ESTAY BARRIENTOS
PROFESOR CORREFERENTE : ING. JORGE SALAS GORDÓNIZ

DICIEMBRE 2019

Agradecimientos

Quiero agradecer en primer lugar a mis padres, por el incondicional apoyo tanto económico como emocional durante toda mi estadía en la universidad. Sin ellos, no estaría donde estoy hoy.

Quiero agradecer a mis amigos de la universidad que han *apañado* en todas desde que nos conocemos. Estoy muy agradecido de haber sido parte de este grupo en el que la ayuda y apoyo entre todos era algo natural.

Doy las gracias, también, a Roberto (*pélot*), quien fue mi compañero de trabajos durante casi toda la universidad, que me *apañó* en innumerables ocasiones y que me salvó ayudándome a encontrar una de mis prácticas.

Finalmente, agradezco profundamente la ayuda y apoyo de parte del Profesor Danilo y de Jorge durante todo el proceso de la memoria.

Resumen

Este trabajo consiste en el desarrollo de un algoritmo capaz de generar las instrucciones de movimiento para un brazo robótico de cinco grados de libertad para simular el agarre de un objeto mediante visión artificial, específicamente, a través de estereovisión, es decir, de la recolección de información a partir del análisis de pares de imágenes con el uso de dos cámaras. El desarrollo del algoritmo se divide en dos grandes bloques: el primero consiste en la aplicación del método de Denavit-Hartenberg (cinemática directa), resolución del problema de cinemática inversa y comunicación con servomotores para la generación del movimiento y el segundo, en la calibración de los sistemas coordinados de las cámaras y del brazo robótico y la detección del punto de agarre del objeto que debe alcanzar el brazo para realizar la simulación.

Palabras claves: brazo robótico, agarre, visión artificial, estereovisión, cinemática.

Abstract

This work consists in the development of an algorithm able to generate movement instructions for a five degree of freedom robotic arm to simulate the grabbing of an object through computer vision, specifically, through stereovision, that is, collecting information from the analysis of image pairs using two cameras. The algorithm development is divided in two main blocks: the first one consists in the application of the Denavit-Hartenberg method (forward kinematics), inverse kinematics problem solving and establishing communication with the servomotors to generate movement and the second one consists in the cameras and robotic arm coordinate systems calibration and the detection of the object grabbing point to be reached by the arm in order to accomplish the simulation.

Keywords: robotic arm, grabbing, computer vision, stereovision, kinematics.

Índice general

Agradecimientos	I
Resumen	II
Abstract	III
Introducción	X
Objetivos	XII
1. Marco teórico	1
1.1. El método de Denavit-Hartenberg	1
1.1.1. Algoritmo de Denavit-Hartenberg	3
1.2. Visión artificial	5
1.2.1. Modelo de la cámara estenopeica	5
1.2.2. Estereovisión	6
1.2.3. Algoritmos de OpenCV	8
2. Control del movimiento del brazo robótico	13
2.1. Aplicación del método de Denavit-Hartenberg	13
2.2. Algoritmo de control de movimiento basado en la aplicación del método de Denavit-Hartenberg	18
2.2.1. Datos del brazo robótico	19
2.2.2. Cinemática	21
2.2.3. Control de los servomotores	25
3. Detección de la posición de objetos a través de estereovisión	26
3.1. Módulos de calibración	28

3.1.1. Calibración de parámetros intrínsecos	28
3.1.2. Calibración de parámetros estéreo de las cámaras	31
3.1.3. Calibración de parámetros extrínsecos	33
3.2. Algoritmo de detección de posición de objetos	37
3.2.1. Descripción del código	38
Conclusiones	47
Trabajos futuros	49
Bibliografía	50
A. Códigos del algoritmo de control de movimiento del brazo robótico	56
A.1. Módulo de Python con funciones de parámetros de Denavit-Hartenberg y de servomotores	56
A.2. Módulo de Python con funciones de cinemática inversa y cinemática directa para generación del movimiento	60
A.3. Script de Arduino para control del movimiento de los servomotores . .	67
B. Códigos de algoritmo de detección de posición de objetos	69
B.1. Módulo de Python con código para calibración de parámetros intrínse- cos de las cámaras	69
B.2. Módulo de Python con código para calibración de parámetros estéreo de las cámaras	78
B.3. Módulo de Python con código para calibración de parámetros extrínsecos	87
B.4. Módulo de Python con código para detectar la posición del objeto . . .	92

Índice de figuras

1.1. Sistema de referencia y variables relevantes del modelo de cámara estenopeica. [4]	6
1.2. Cámara estéreo, sistemas de referencia y parámetros relevantes. [5] . . .	7
1.3. Tablero de ajedrez como patrón de calibración. [7]	8
1.4. Tipos de distorsión radial. [8]	9
1.5. Distorsión tangencial. [9]	9
1.6. Planos de imagen de las cámaras luego de la rectificación estéreo. [15] .	11
1.7. Dados los puntos clave (encerrados en los círculos) del objeto (a), se logran identificar cierta cantidad en la imagen (b) y a través de la restricción a la que están sometidos, se estima la pose del objeto relativa a la cámara (c). [20]	12
2.1. Imagen del modelo 3D de Inventor del brazo robótico. Elaboración propia.	13
2.2. Imagen del brazo robótico. Elaboración propia.	14
2.3. Versión simplificada del brazo robótico. Cada eslabón se encuentra destacado con un color distintivo. Elaboración propia.	14
2.4. Cadena cinemática del brazo robótico según la metodología de Denavit-Hartenberg. Elaboración propia.	15
2.5. Posición alcanzada antes de la corrección del parámetro d_5 . Elaboración propia.	17
2.6. Posición alcanzada después de la corrección del parámetro d_5 . Elaboración propia.	17
2.7. Diagrama de flujo de algoritmo de movimiento. Elaboración propia. . .	18
2.8. Trayectoria y movimiento del brazo generado por la función $ik()$. Elaboración propia.	23

3.1.	Diagrama de flujo del algoritmo completo (calibraciones, detección de posición de objetos y movimiento del brazo). Para las calibraciones “mono” ambas cámaras son iguales, consideración explicada más adelante. Elaboración propia.	27
3.2.	Tablero de ajedrez como patrón de calibración y sus parámetros relevantes. En este caso particular, el tamaño de los cuadros <code>size</code> tiene un valor de 22 milímetros y las intersecciones interiores <code>nx</code> y <code>ny</code> corresponden a 9 y 6 respectivamente. [29]	29
3.3.	Identificación de las intersecciones interiores del patrón de calibración (puntos de colores) y asociación de los puntos de objeto en milímetros (color blanco) con los puntos de imagen (color rojo y verde) (u_{yx}, v_{yx}) en píxeles (<code>[px]</code>). Elaboración propia.	30
3.4.	Marcador ArUco de 6x6 (cuadros internos). [29]	33
3.5.	Sistemas de referencia y disposición física del brazo robótico y del marcador ArUco para la calibración. Las variables x , y y z en color naranja representan la posición, en las respectivas direcciones, del sistema O_C respecto de O_Y . Elaboración propia.	34
3.6.	Estimación de pose de marcador ArUco 6x6. Los segmentos que se encuentran en el centro del marcador representan a sus ejes coordenados (rojo para el eje X , verde para el eje Y y azul para el eje Z) y el borde verde que encierra al marcador representa su reconocimiento en la imagen. Elaboración propia.	36
3.7.	(a) Mapa de disparidad sin filtrar. Los elementos más cercanos a la cámara están representado por un color claro y los más lejanos por un color oscuro. (b) Corrección del mapa de disparidad realizada por el Filtro de disparidad WLS. [41]	38
3.8.	Sección del panel de deslizadores para configuración de parámetros del módulo. Elaboración propia.	39
3.9.	(a) Imagen izquierda rectificada. (b) Representación gráfica del mapa de disparidad. Las zonas más claras representan a los elementos más cercanos a las cámaras, mientras que las regiones más oscuras representan a los más lejanos. Elaboración propia.	40
3.10.	(a) Imagen izquierda rectificada a color con objeto de interés (esfera color naranja). (b) Imagen binaria con figura del objeto luego de aplicar la función <code>inRange()</code> . Elaboración propia.	41

3.11. (a) Imagen binaria con figura del objeto de interés (círculo blanco) y regiones no deseadas. (b) Imagen con curvas encontradas. La curva de color verde corresponde a la seleccionada por el usuario, mientras que las rojas representan al resto de las curvas encontradas. Elaboración propia.	43
3.12. Determinación del centro del objeto esférico con respecto del sistema coordinado de la cámara izquierda. Elaboración propia.	44
3.13. Determinación del punto de agarre del objeto esférico respecto del sistema de referencia del brazo robótico. Elaboración propia.	45

Índice de tablas

2.1. Correspondencia de variables l_i con parámetros de Denavit-Hartenberg.	16
2.2. Valores de los parámetros de Denavit-Hartenberg para el i -ésimo GDL.	16
2.3. Valores de los parámetros de los servomotores para cada grado de libertad i	21

Introducción

Desde el Departamento de Mecánica de la Universidad Técnica Federico Santa María se desea comenzar la investigación sobre prótesis de mano inteligentes cuyo funcionamiento provenga del análisis de información mediante visión artificial. Es por esto que se motivó la realización del presente trabajo, el cual representa un primer paso en esta dirección.

La idea de desarrollar una mano prostética inteligente basada en estereovisión está fundada en poder ofrecer otra alternativa a los tipos como lo son las prótesis de mano pasivas (sin partes móviles), las que son accionadas por movimientos de partes del cuerpo y las que funcionan bajo control mioeléctrico. Se apunta a que esta nueva alternativa pueda presentar ventajas tanto en el nivel de experiencia del usuario, como en los costos de construcción.

Se decide dar comienzo a esta investigación a través del desarrollo de un algoritmo que sea capaz de controlar un brazo robótico a través de la estereovisión (uso de dos cámaras) para simular el agarre de un objeto. Debido a esto último, aparece la necesidad de indagar en los modelos y métodos escogidos para este trabajo relacionados, por una parte, con el movimiento del brazo robótico y, por la otra, con la estereovisión para construir un sistema que integre todos estos componentes y que dé solución al objetivo de este primer avance.

A través del método de cinemática, la librería de algoritmos de cinemática inversa y el control de los servomotres, se desarrolla un primer sistema de módulos capaz de dar movimiento al brazo robótico al interpretar los parámetros que definen la posición que este debe alcanzar. El segundo conjunto de módulos se desarrolla en base al modelo matemático de las cámaras, al de estereovisión y a la librería de visión artificial y per-

mite obtener de forma automática, a partir de las imágenes, la posición del objeto que se desea alcanzar. La implementación conjunta de estos grupos de módulos conforman un algoritmo capaz de identificar la posición de un objeto particular y simular su agarre a través del movimiento del brazo robótico.

Objetivos

Objetivo general

Desarrollar un algoritmo de control basado en estereovisión para simulación de agarre de un objeto mediante un brazo robótico.

Objetivos específicos

- Definir la estructura cinemática de los eslabones del brazo robótico mediante la metodología de Denavit-Hartenberg.
- Desarrollar un algoritmo para generar el movimiento del brazo robótico en base a lo obtenido con la metodología de Denavit-Hartenberg.
- Elaborar algoritmo que permita obtener la posición de un objeto a través de estereovisión de forma automática y que trabaje en conjunto con el código de movimiento para simular agarre.

Capítulo 1

Marco teórico

1.1. El método de Denavit-Hartenberg

El modelo de Denavit-Hartenberg es un procedimiento que permite obtener el modelo cinemático directo, representándolo mediante un conjunto de transformaciones homogéneas. El modelo consiste en determinar una tabla de parámetros relacionados con la geometría de los eslabones del robot. Como convención, los parámetros se definen en referencia a un robot de cadena cinemática abierta. [1]

Las transformaciones necesarias dependen exclusivamente de las características geométricas de los eslabones del robot y consisten en una sucesión de rotaciones y traslaciones que permiten relacionar el sistema de referencia de un elemento con el anterior y son las siguientes:

1. Rotación alrededor del eje z_{i-1} un ángulo θ_i .
2. Traslación a lo largo de z_{i-1} una distancia d_i .
3. Traslación a lo largo de x_i una distancia a_i .
4. Rotación alrededor del eje x_i un ángulo α_i . [2]

Debido a que el producto entre matrices no es conmutativo, las transformaciones deben respetar el orden mostrado, resultando:

$${}^{i-1}A_i = T_1(z, \theta_i) T_2(0, 0, d_i) T_3(a_i, 0, 0) T_4(x, \alpha_i) \quad (1.1)$$

siendo cada transformación T_i , lo que sigue:

$$T_1 = \begin{bmatrix} \cos(\theta_i) & -\sin(\theta_i) & 0 & 0 \\ \sin(\theta_i) & \cos(\theta_i) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.2)$$

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.3)$$

$$T_3 = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.4)$$

$$T_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha_i) & -\sin(\alpha_i) & 0 \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.5)$$

luego, la ecuación 1.1 queda expresada como:

$${}^{i-1}A_i = \begin{bmatrix} \cos(\theta_i) & -\cos(\alpha_i)\sin(\theta_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\alpha_i)\cos(\theta_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad (1.6)$$

donde las variables θ_i , a_i , d_i , α_i son los parámetros Denavit-Hartenberg y se definen de la siguiente manera:

- θ_i : ángulo que forman los ejes x_{i-1} y x_i medido en un plano perpendicular al eje z_{i-1} , utilizando la regla de la mano derecha. Es un parámetro variable en articulaciones rotatorias.

- d_i : distancia a lo largo del eje z_{i-1} desde el origen del sistema de coordenadas $(i-1)$ -ésimo hasta la intersección del eje z_{i-1} con el eje x_i . Es variable en articulaciones prismáticas.
- a_i : para el caso de articulaciones rotatorias, es distancia a lo largo del eje x_i que va desde la intersección del eje z_{i-1} con el eje x_i hasta el origen del sistema i -ésimo. Para el caso de articulaciones prismáticas, se obtiene como la distancia más pequeña entre los ejes z_{i-1} y z_i .
- α_i : ángulo existente entre el eje z_{i-1} y el eje z_i , medido en un plano perpendicular al eje x_i , utilizando la regla de la mano derecha. [2]

1.1.1. Algoritmo de Denavit-Hartenberg

El procedimiento del método queda definido por los siguientes pasos:

1. Numerar los eslabones móviles desde 1 a n y la base como el eslabón 0.
2. Numerar cada articulación desde 1 a n .
3. Localizar el eje de cada articulación. En caso de ser rotativa, el eje será su propio eje de giro. Si es prismática, será el eje a lo largo del cual se produce el desplazamiento.
4. Para i desde 0 a $n-1$, situar el eje z_i sobre el eje de la articulación $i+1$.
5. Situar el origen del sistema de la base S_0 en cualquier punto del eje z_0 . Los ejes x_0 e y_0 se situarán de modo que formen un sistema dextrógiro con z_0 .
6. Para i desde 1 a $n-1$, situar el sistema S_i (solidario al eslabón i) en la intersección del eje z_i con la línea normal común a z_{i-1} y z_i . Si ambos ejes se intersectan, S_i se debe situar en el punto de intersección. Si son paralelos, S_i se debe situar en la articulación $i+1$.
7. Ubicar x_i en la línea normal común a z_{i-1} y z_i .
8. Situar y_i de modo que forme un sistema dextrógiro con x_i y z_i .
9. Ubicar el sistema S_n en el extremo del robot de modo que z_n coincida con la dirección de z_{n-1} y x_n sea normal a z_{n-1} y z_n .

10. Obtener θ_i como el ángulo que hay que rotar en torno a z_{i-1} para que x_{i-1} y x_i sean paralelos.
11. Obtener d_i como la distancia, medida a lo largo de z_{i-1} , que se debe desplazar S_{i-1} para que x_i y x_{i-1} estén alineados.
12. Obtener a_i como la distancia medida a lo largo de x_i (considerando a x_i y x_{i-1} alineados) que se debe desplazar S_{i-1} para que coincida con S_i .
13. Obtener α_i como el ángulo que se debe rotar en torno a x_i (considerando que x_i y x_{i-1} coinciden), para que S_{i-1} coincida totalmente con S_i .
14. Obtener las matrices de transformación ${}^{i-1}A_i$ (Ecuación 1.6).
15. Obtener la matriz de transformación que relacione el sistema de la base con el del extremo del robot:

$$T = {}^0A_1 {}^1A_2 \cdots {}^{n-1}A_n \quad (1.7)$$

16. La matriz T define la orientación (submatriz de rotación) y posición (submatriz de traslación) del extremo referido a la base en función de las n coordenadas articulares. [2]

1.2. Visión artificial

1.2.1. Modelo de la cámara estenopeica

El modelo de la cámara estenopeica es uno de los modelos más simples, pero razonablemente realistas, en el cual el lente es tratado simplemente como un agujero muy pequeño. Todos los rayos de luz que llegan al plano de la imagen deben pasar a través de este agujero que se ubica en frente del plano de imagen fotosensible. [3]

La transformación de un punto del espacio (x, y, z) al plano de la imagen (u, v) en este modelo se puede describir de la siguiente manera:

$$\begin{bmatrix} s \cdot u \\ s \cdot v \\ s \end{bmatrix} = \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad (1.8)$$

donde s puede ser tratado como un factor de escala en las coordenadas homogéneas utilizadas para describir los puntos de la imagen, f_u y f_v corresponden a la combinación del largo focal de la cámara y el tamaño de los pixeles en las direcciones u y v , y c_u y c_v son las coordenadas en las cuales el eje óptico intersecta el plano de la imagen (también denominado como centro óptico). El eje óptico es la recta perpendicular que se extiende desde el plano de la imagen y pasa a través del agujero de la cámara. [3]

En la Figura 1.1 se observa el sistema coordenado de la cámara y las variables relevantes. Las relaciones que se desprenden de esta configuración son las siguientes:

$$X = \frac{u_p - c_u}{f} \cdot Z \quad (1.9)$$

$$Y = \frac{v_p - c_v}{f} \cdot Z \quad (1.10)$$

donde f corresponde al largo focal de la cámara y u_p y v_p son la posición del punto p en la imagen en las direcciones u y v respectivamente. [4]

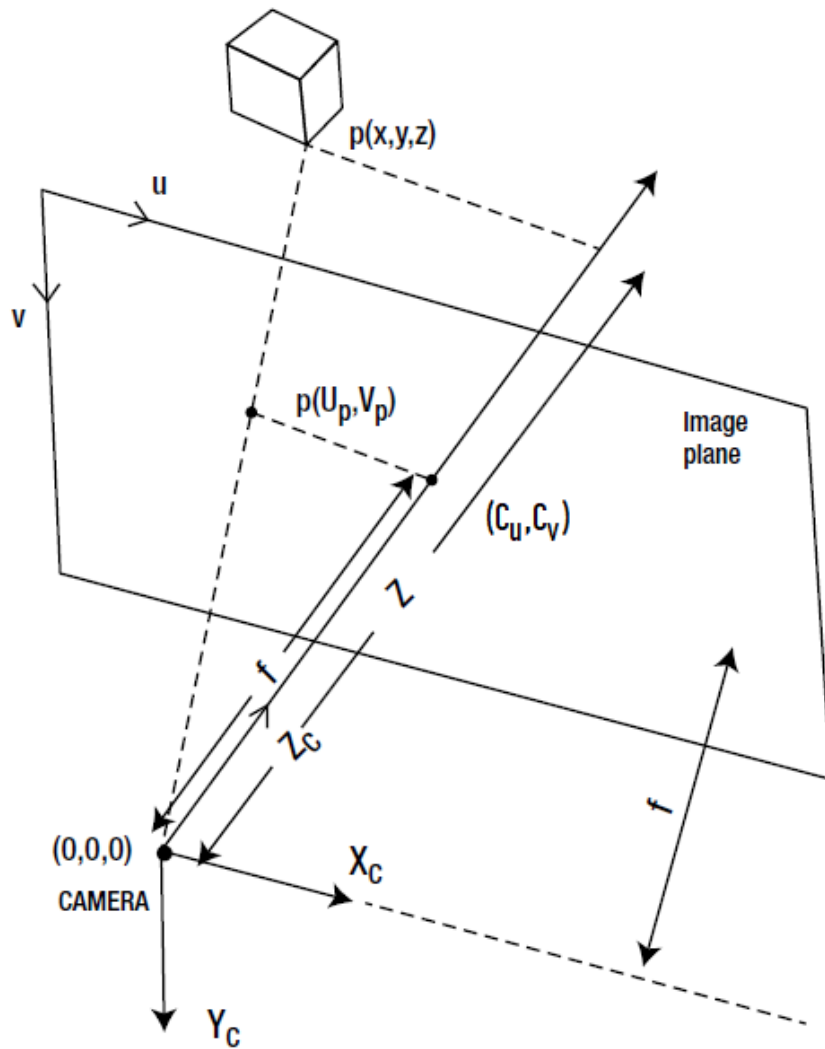


Figura 1.1: Sistema de referencia y variables relevantes del modelo de cámara estenopeica. [4]

1.2.2. Estereovisión

La estereovisión corresponde a la captura de información de la escena a través de dos cámaras separadas por una distancia fija llamada línea base (“*baseline*” en inglés), como se observa en la Figura 1.2. Una configuración de cámaras estéreo permite calcular la profundidad física de un punto de imagen utilizando el concepto de disparidad. Este último corresponde a la magnitud del movimiento horizontal aparente (desde una imagen respecto a la otra) del punto en cuestión. [5]

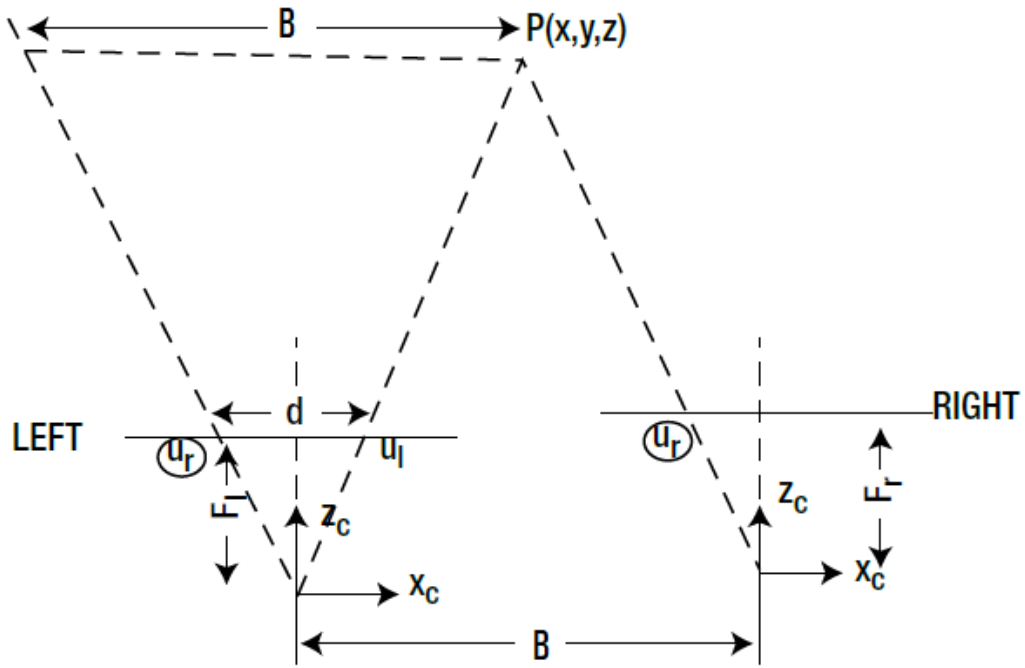


Figura 1.2: Cámara estéreo, sistemas de referencia y parámetros relevantes. [5]

De la configuración del modelo mostrado en la Figura 1.2 se tienen las siguientes relaciones:

$$d = u_l - u_r \quad (1.11)$$

$$Z = B \cdot \frac{f}{d} \quad (1.12)$$

donde d corresponde a la disparidad, u_l y u_r corresponden a las posiciones del punto en la dirección u de la imagen izquierda y derecha respectivamente, f es el largo focal y B corresponde a la distancia de la línea base. [5]

1.2.3. Algoritmos de OpenCV

OpenCV¹ es una librería de visión artificial de código abierto y es la que se utilizará para el desarrollo del algoritmo de detección de la posición de objetos. En las siguientes secciones se tratarán los algoritmos de esta librería que presentan una mayor relevancia.

I. Calibración de una cámara

La calibración de la cámara corresponde al proceso de relacionar su rango de medición con las cantidades del mundo real que mide y encontrar los parámetros de las distorsiones que afectan a las imágenes. Para esto, se deben determinar los parámetros de distorsión y los del modelo de la cámara (también denominados como *parámetros intrínsecos*, que son los mismos que conforman la matriz 3x3 del modelo de la Ecuación 1.8 de la Sección 1.2.1 [6]). Esta calibración se logra presentando un patrón de calibración conocido (ver Figura 1.3) a la cámara repetidas veces en posiciones y orientaciones ligeramente diferentes. [4] [7]

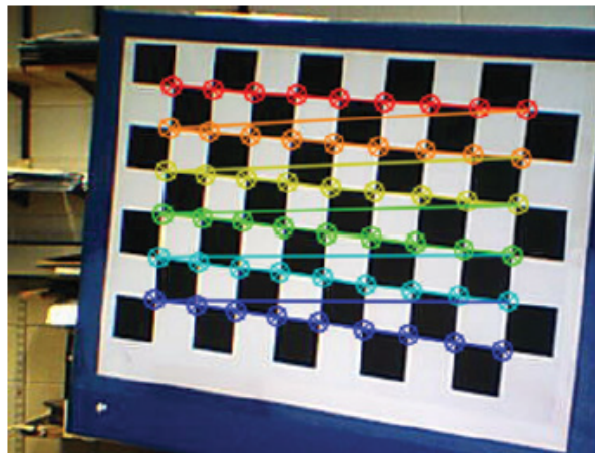


Figura 1.3: Tablero de ajedrez como patrón de calibración. [7]

¹OpenCV, página web - <https://opencv.org/>

Las dos formas comunes de distorsión que aparecen en los sistemas de cámaras son:

1. **Distorsión radial:** como se muestra en la Figura 1.4, es simétrica radialmente y el nivel de distorsión está relacionado a la distancia desde el eje óptico de la cámara (usualmente cerca del centro de la imagen). El efecto producido por esta distorsión es el escalamiento de la imagen en la medida que la zona de la imagen está más cerca o lejos del eje óptico.
2. **Distorsión tangencial:** ocurre cuando el lente no es perfectamente paralelo respecto del plano de la imagen. El efecto de esta distorsión es el escalamiento no uniforme que varía desde un lado de la imagen al otro (ver Figura 1.5). [7]

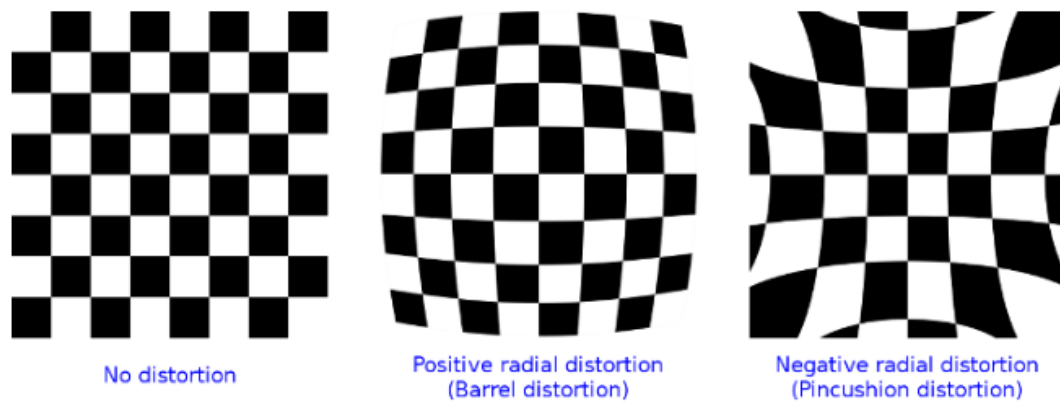


Figura 1.4: Tipos de distorsión radial. [8]

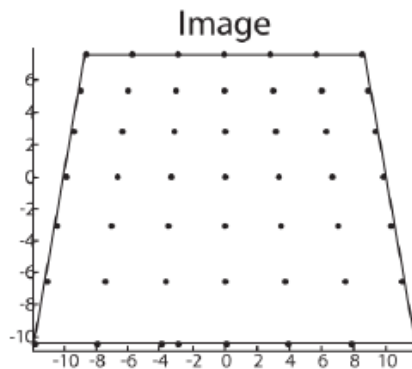


Figura 1.5: Distorsión tangencial. [9]

La función de OpenCV `calibrateCamera()` [10] es la que permite calibrar la cámara.

II. Calibración de dos cámaras

Esta calibración (calibración estéreo) corresponde el proceso de calcular la relación geométrica entre las dos cámaras en el espacio, vale decir, la matriz de rotación R (la cual permitirá que el plano de imagen de la cámara derecha quede paralelo al de la cámara izquierda) y el vector de traslación $\vec{T}$. [11]

Para llevar a cabo la calibración estéreo se sigue prácticamente el mismo procedimiento utilizado en la calibración de una cámara: presentar un patrón de calibración conocido (ver Figura 1.3) en diferentes posiciones y orientaciones pero esta vez a las dos cámaras simultáneamente. [5] [11]

La función de OpenCV que se encarga del proceso de calibración estéreo corresponde a `stereoCalibrate()` [12].

III. Rectificación estéreo

El proceso de rectificación estéreo corresponde a la reproyección de los planos de imagen de ambas cámaras de tal manera que estén contenidos exactamente en el mismo plano y que las filas de las imágenes estén perfectamente alineados. En la Figura 1.6 se puede apreciar el efecto de la rectificación. [13]

La correspondencia estéreo (encontrar un mismo punto en la imagen de ambas cámaras), luego de la rectificación, será mucho más confiable y manejable, por lo que resultará mucho más sencillo calcular la disparidad. [13]

La rectificación estéreo en OpenCV se realiza en tres pasos. Primero, la función `stereoRectify()` [14] toma parámetros resultantes de las dos calibraciones ya descritas y retorna:

- R_l : Matriz de rotación que se aplicará sobre la imagen de la cámara izquierda para alinearla.
- R_r : Matriz de rotación que se aplicará sobre la imagen de la cámara derecha para su alineación.
- P_l : Matriz aparente de la cámara izquierda (parámetros intrínsecos) alineada.

- P_r : Matriz aparente de la cámara derecha (parámetros intrínsecos) alineada.

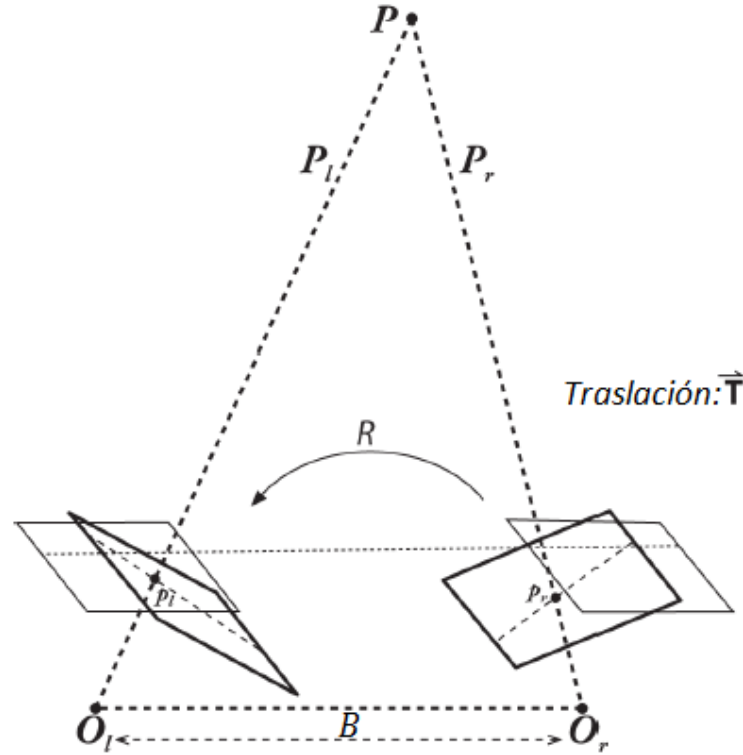


Figura 1.6: Planos de imagen de las cámaras luego de la rectificación estéreo. [15]

Luego, con esta información y con la función `initUndistortRectifyMap()` [16], se obtienen los mapas de pixeles para rectificar las imágenes. Finalmente, la función `remap()` [17] se encarga de aplicar los mapas para la rectificación de las imágenes. [18]

IV. Estimación de pose de un objeto

Para encontrar la relación entre la pose de un objeto y la cámara, se deben haber identificado puntos claves (“*keypoints*” en inglés, cuya definición corresponde a pequeñas porciones de la imagen, que por una o diversas razones, son inusualmente distintas [19]) de este y haberlos asociado a su sistema coordenado. Como restricción, cada punto encontrado debe estar contenido en una única recta que sale desde el pixel correspondiente de la imagen a través de la apertura de la cámara. Dadas suficientes de estas restricciones (es decir, suficientes puntos) y si el objeto está a una distancia

adecuada de la cámara, el cálculo de su pose respecto a esta última tendrá solo una solución. En la Figura 1.7 se puede observar lo explicado. [20]

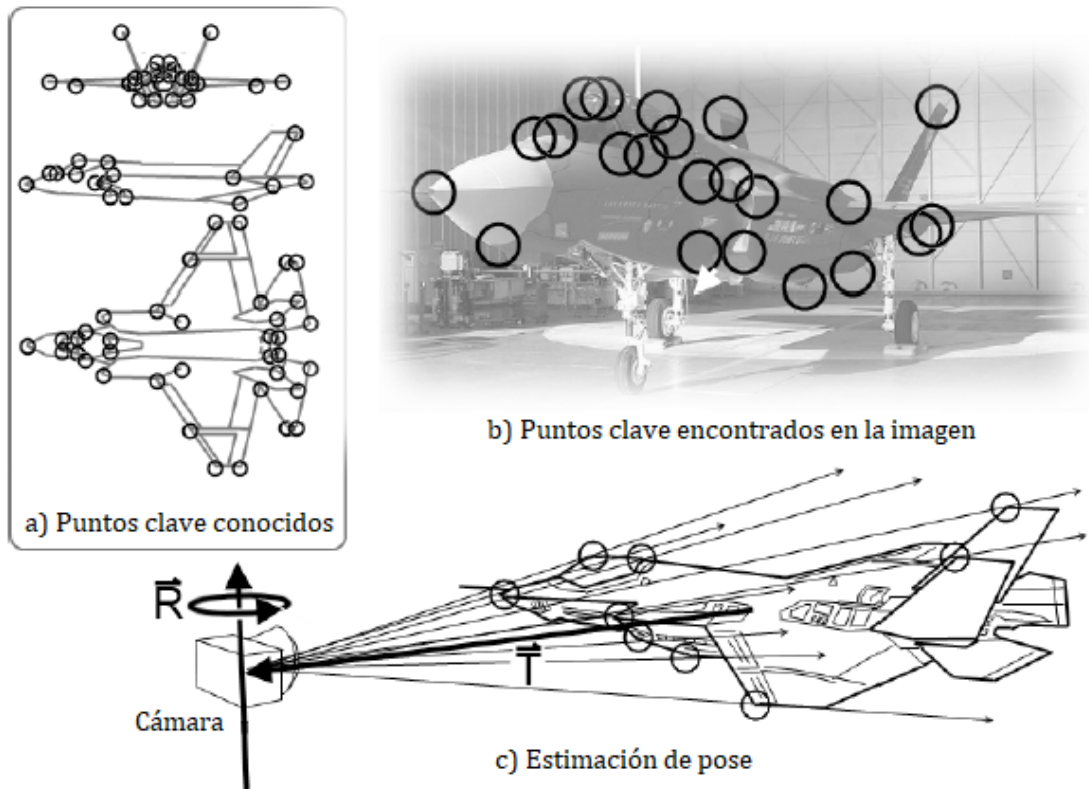


Figura 1.7: Dados los puntos clave (encerrados en los círculos) del objeto (a), se logran identificar cierta cantidad en la imagen (b) y a través de la restricción a la que están sometidos, se estima la pose del objeto relativa a la cámara (c). [20]

La función de OpenCV que se encarga de resolver el problema de la pose de un objeto corresponde a `solvePnP()` [21].

Capítulo 2

Control del movimiento del brazo robótico

2.1. Aplicación del método de Denavit-Hartenberg

El brazo robótico que se ha utilizado en este trabajo ha sido diseñado previamente y entregado para el desarrollo de este trabajo. El brazo corresponde a uno con 5 grados de libertad, todos de tipo rotativo. En las Figuras 2.1, 2.2 y 2.3 se puede apreciar la estructura del brazo. En los códigos y en este escrito aparecerá el nombre de “Y.A.R.B.I.Z.”, el cual corresponde a una forma alternativa de referirse al brazo.

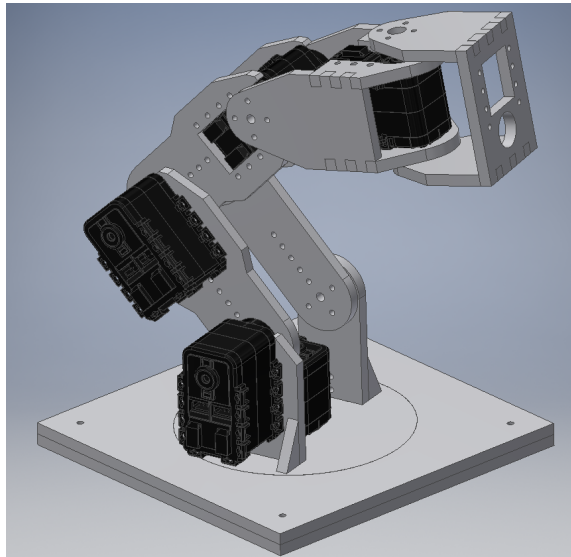


Figura 2.1: Imagen del modelo 3D de Inventor del brazo robótico. Elaboración propia.



Figura 2.2: Imagen del brazo robótico. Elaboración propia.

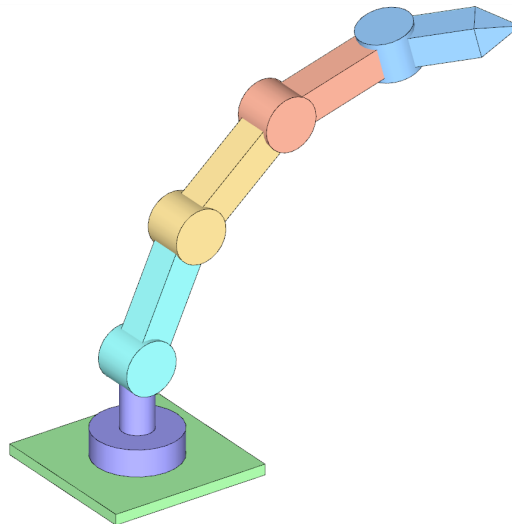


Figura 2.3: Versión simplificada del brazo robótico. Cada eslabón se encuentra destacado con un color distintivo. Elaboración propia.

En la Figura 2.4 se muestra el resultado de aplicar el algoritmo presentado en la Sección 1.1.1.

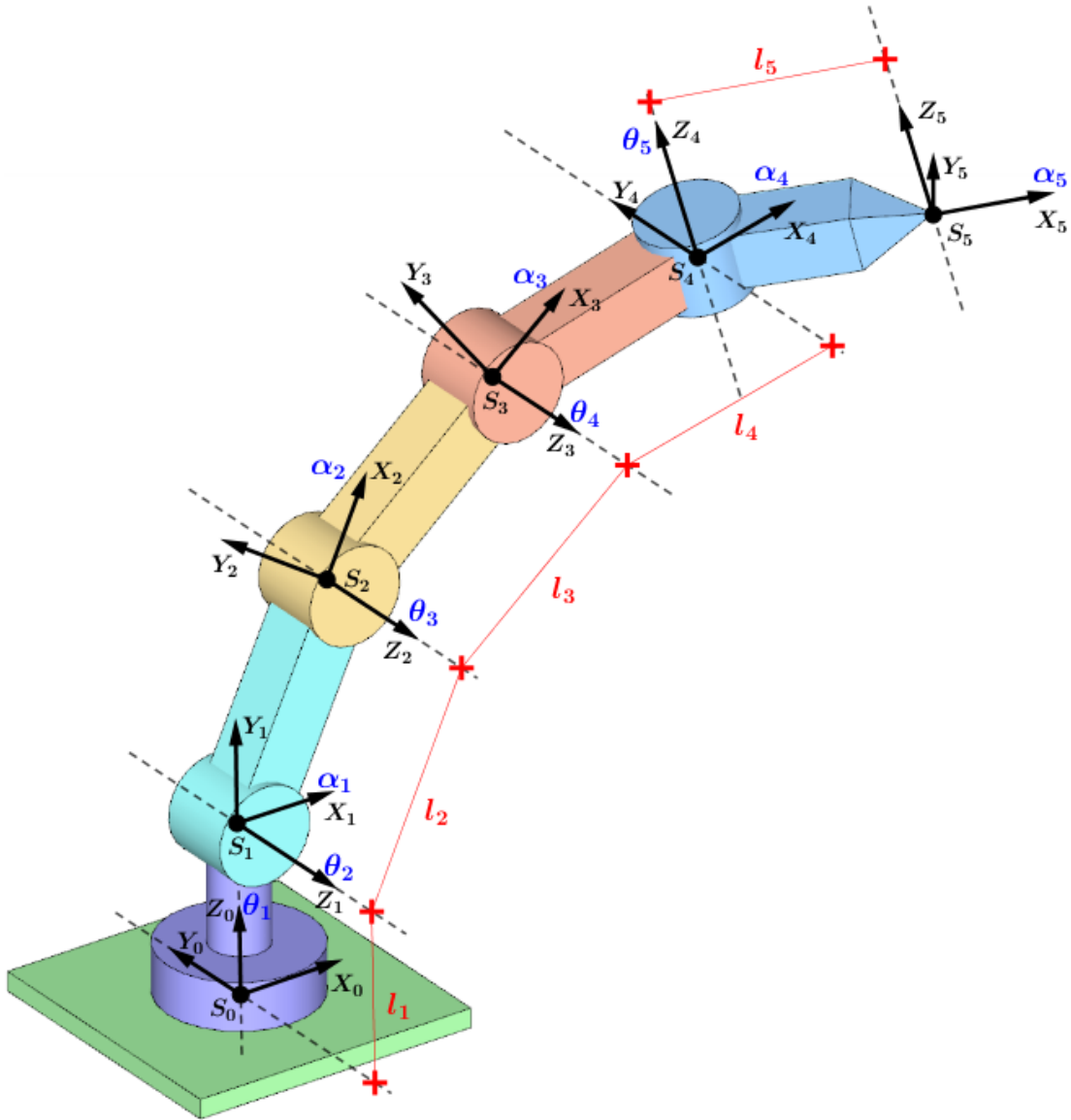


Figura 2.4: Cadena cinemática del brazo robótico según la metodología de Denavit-Hartenberg. Elaboración propia.

Las variables l_i que aparecen en la Figura 2.4 representan las distancias entre ejes de rotación y pueden corresponder tanto a los parámetros d_i como a_i dependiendo de la disposición relativa de los sistemas de referencia S_i , como se muestra en la Tabla 2.1.

Con la estructura bien definida se procedió a realizar las mediciones pertinentes para los parámetros de Denavit-Hartenberg θ_i , d_i , a_i y α_i (ver Tabla 2.2), ya definidos en la Sección 1.1.

Tabla 2.1: Correspondencia de variables l_i con parámetros de Denavit-Hartenberg.

l_i	Parámetro D-H
l_1	d_1
l_2	a_2
l_3	a_3
l_4	a_4
l_5	a_5

Tabla 2.2: Valores de los parámetros de Denavit-Hartenberg para el i -ésimo GDL.

i	θ_i [°]	d_i [mm]	a_i [mm]	α_i [°]
1	θ_1	62	0	90
2	θ_2	0	70	0
3	θ_3	0	70	0
4	θ_4	0	61	-90
5	θ_5	-30*	48**	0

Nótese que los parámetros θ_i no tienen asignado un valor numérico. Esto es debido a que son las variables que controlan los grados de libertad y pueden tomar diferentes valores dependiendo de la posición que se desee alcanzar con el brazo.

El parámetro d_5 (*) en realidad tiene un valor de cero, pero se añadió con fines de corrección, debido a errores de ángulo en los servomotores que controlan los grados de libertad 2 y 3. En concreto, esta corrección se encarga de desplazar al sistema de referencia 5 (S_5) en sentido negativo del eje Z_4 (ver Figura 2.4 como referencia) para compensar la diferencia que se genera entre la posición objetivo y la real: en otras palabras, con la corrección los servomotores provocan que el extremo quede en un mayor z (medido desde S_0 , según Figura 2.4) y con una orientación tal que el centro de la superficie del eslabón quede más cerca de la posición del objeto para la simulación de agarre. En las Figuras 2.5 y 2.6 se aprecia el efecto de la corrección.

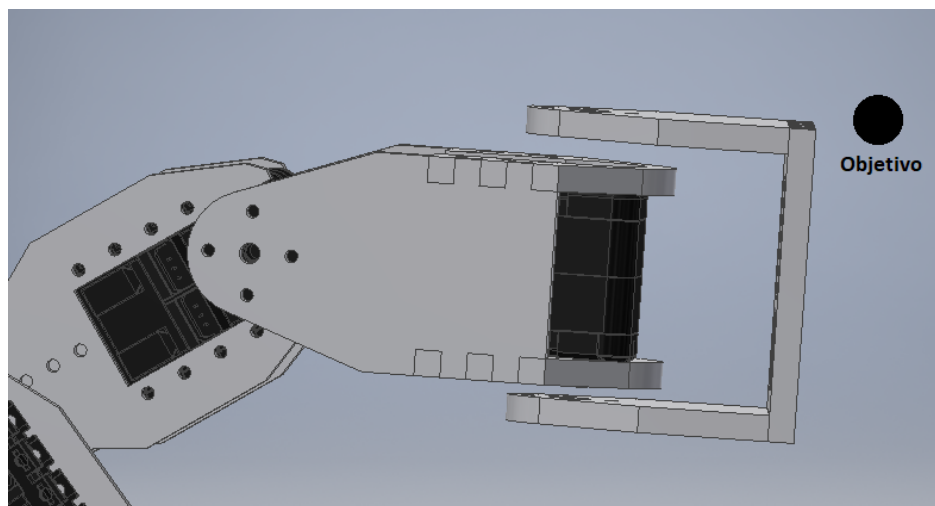


Figura 2.5: Posición alcanzada antes de la corrección del parámetro d_5 . Elaboración propia.

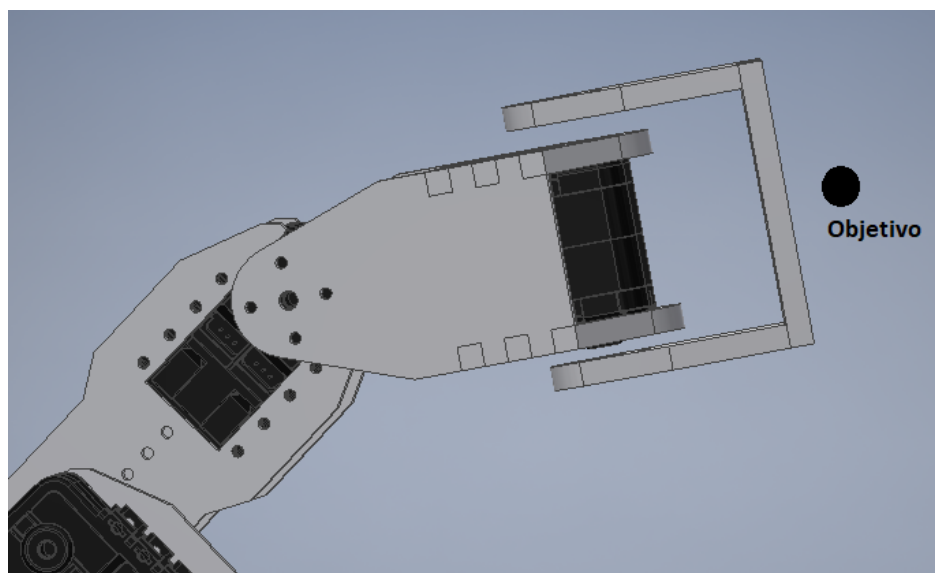


Figura 2.6: Posición alcanzada después de la corrección del parámetro d_5 . Elaboración propia.

En el caso del parámetro a_5 (**), el cual representa la distancia entre el eje de la última articulación y la superficie extrema del último eslabón (ver Figura 2.4 y Tabla 2.1), el valor real corresponde a 43 milímetros, pero se añadieron 5 milímetros para evitar la colisión entre el extremo del brazo y el objeto que se desee alcanzar.

2.2. Algoritmo de control de movimiento basado en la aplicación del método de Denavit-Hartenberg

En esta sección se presentarán los detalles relacionados al algoritmo de movimiento y las diferentes partes que lo componen. La función de este algoritmo es permitir controlar el movimiento del brazo a partir de la información que se capture a través de la visión artificial. En el diagrama de la Figura 2.7 se aprecian las relaciones que existen entre los diferentes componentes que permiten cumplir el objetivo de este algoritmo.

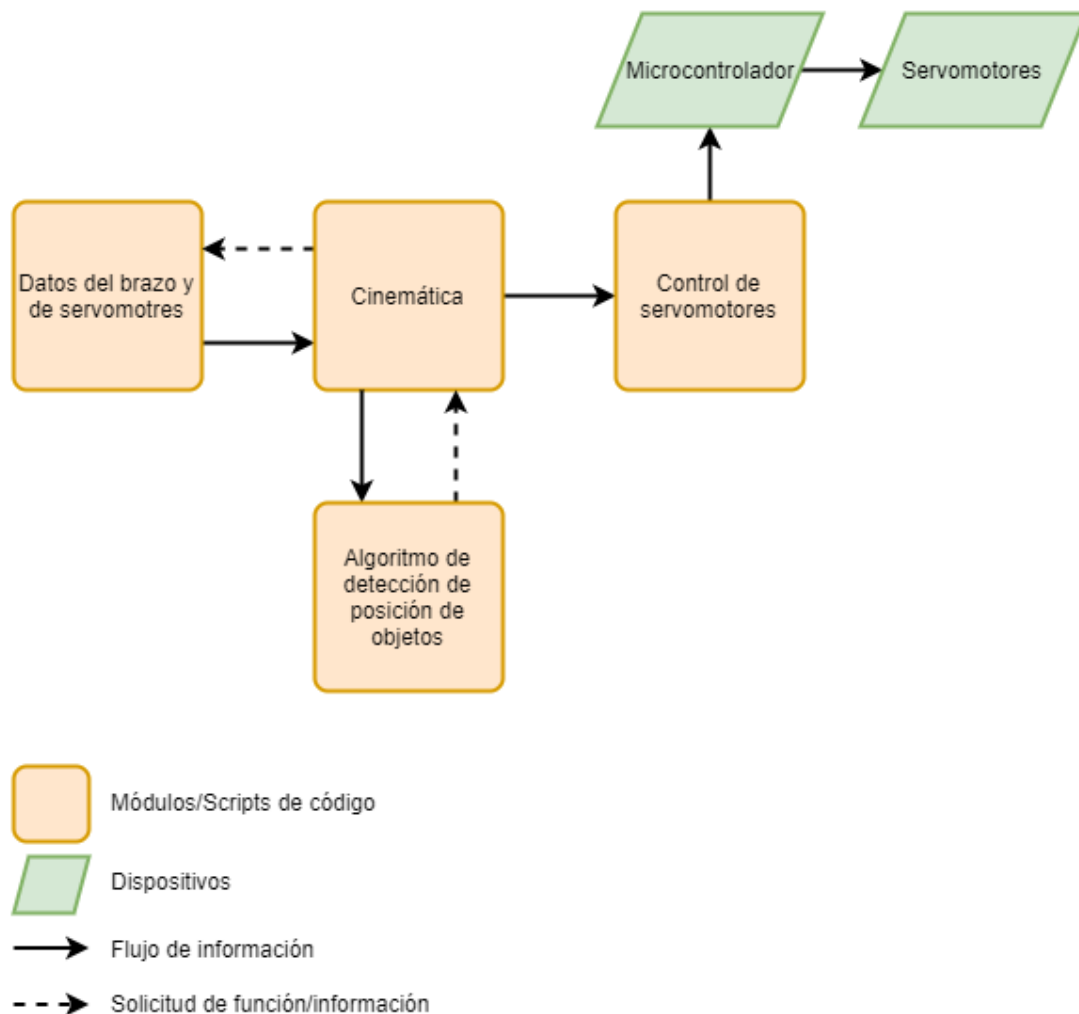


Figura 2.7: Diagrama de flujo de algoritmo de movimiento. Elaboración propia.

2.2.1. Datos del brazo robótico

Este módulo (ver Anexo A.1) escrito en Python¹, y que hace uso de la librería NumPy², tiene la función de proporcionar los parámetros relevantes para los algoritmos de cinemática directa e inversa que se presentarán la Sección 2.2.2. Estos parámetros son entregados por dos funciones.

I. Función de parámetros de Denavit-Hartenberg y punto alcanzado

Esta función se encuentra definida como `par_dh()` (línea 6 del código) y tiene como objetivo proporcionar los parámetros de Denavit-Hartenberg y el punto p alcanzado dados los ángulos θ_i .

Como entrada recibe:

- `thetai`: variable tipo lista con los ángulos (flotantes) θ_i de cada grado de libertad de la forma $[\theta_1, \theta_2, \theta_3, \theta_4, \theta_5]$.

Como retorno, esta función entrega:

- `dh`: variable tipo lista que contiene 5 sublistas con los parámetros de Denavit-Hartenberg (según Tabla 2.2); una sublista para cada grado de libertad.
- `p`: arreglo NumPy 3x1 que contiene las coordenadas (en milímetros) alcanzadas por el extremo del brazo respecto del sistema coordenado base S_0 (ver Figura 2.4).

Primero se declaran los parámetros de Denavit-Hartenberg para todos los grados de libertad (d_i y a_i en milímetros y α_i y θ_i en radianes) para después ser almacenados en la variable `dh`. Luego, estos parámetros son utilizados para calcular las matrices de transformación entre sistemas de referencia consecutivos correspondientes a la Ecuación 1.6 y poder obtener la matriz de transformación final de la Ecuación 1.7. Finalmente, se extraen las coordenadas del punto alcanzado desde la submatriz de traslación mencionada en el paso número 16 del algoritmo de Denavit-Hartenberg (Sección 1.1.1) y se almacenan en `p`.

¹Python, página web - <https://www.python.org>

²NumPy, página web - <https://numpy.org>

II. Función de parámetros de los servomotores

La función `par_fu()` (línea 97), cuyo nombre responde a “*Parámetros de las funciones $u_i(\theta_i)$* ”, tiene como objetivo entregar los parámetros relacionados con los servomotores (Dynamixel AX-12A³):

- $u_{i,0}$: ubicación, en unidades del servomotor, donde θ_i es cero.
- $u_{i,min}$: límite mínimo, en unidades del servomotor, que este puede alcanzar. Este valor puede estar determinado para evitar colisiones dada la estructura del brazo o bien, por el límite mínimo intrínseco del servomotor (en este caso 0 unidades [22]).
- $u_{i,max}$: límite máximo, en unidades del servomotor, que este puede alcanzar. Al igual que el mínimo, puede estar determinado para evitar colisiones o por el límite máximo intrínseco, el cual es 1023 unidades [22].
- $\theta_{i,min}$: ángulo mínimo que puede tomar θ_i .
- $\theta_{i,max}$: ángulo máximo que puede tomar θ_i .
- k : constante de conversión en unidades del servomotor por grado, correspondiente a $1/0,29$ [22].

Las funciones $u_i(\theta_i)$, que se utilizarán en el módulo de cinemática de la Sección 2.2.2, se encargan de convertir la variables θ_i en grados al universo de las unidades del servomotor con el fin de poder comunicar las instrucciones de movimiento a los mismos. Estas funciones tienen la siguiente forma:

$$u_i(\theta_i) = u_{i,0} + \theta_i \cdot k \text{ [u]}; \quad \theta_i \in [\theta_{i,min}, \theta_{i,max}] \text{ [}^\circ\text{]} \quad (2.1)$$

Las unidades de rotación de los servomotores se denotarán por “u”.

³Dynamixel AX-12A, página web - <http://emanual.robotis.com/docs/en/dxl/ax/ax-12a/>

Esta función (`par_fu()`) no requiere ninguna variable de entrada y el retorno corresponde a:

- `pfu`: variable tipo lista que contiene una sublista con los parámetros $u_{i,0}$, $u_{i,min}$, $u_{i,max}$, $\theta_{i,min}$, $\theta_{i,max}$ por cada grado de libertad más la constante de conversión k .

A lo largo de las líneas del código, se declaran todas las variables de la Tabla 2.3 para luego almacenarlas en la variable `pfu`.

Tabla 2.3: Valores de los parámetros de los servomotores para cada grado de libertad i .

i	$u_{i,0}$ [u]	$u_{i,min}$ [u]	$u_{i,max}$ [u]	$\theta_{i,min}$ [°]	$\theta_{i,max}$ [°]
1	512	0	1023	-148,48	148,19
2	202	133	885	-20,01	198,07
3	512	0	1023	-148,48	148,19
4	512	167	857	-100,05	100,05
5	512	167	857	-100,05	100,05

2.2.2. Cinemática

Este módulo (presentado en el Anexo A.2) escrito en Python utiliza las librerías NumPy, PySerial⁴ (para comunicación con el puerto serial COM al que se encuentre conectado el microcontrolador) e IKPy⁵ (para resolver los problemas de cinemática inversa).

El objetivo de este módulo es construir las instrucciones de movimiento para los servomotores a partir de la resolución del problema de cinemática directa o inversa.

I. Función de cinemática inversa

Esta función se encuentra definida bajo el nombre de `ik()` (línea 12 del código), por “*Inverse Kinematics*” en inglés, y tiene como objetivo resolver el problema de cinemática inversa (resolver para los ángulos θ_i) dada la posición del objeto en el espacio, elaborar las instrucciones para el movimiento de los servomotores y comunicarlas

⁴PySerial, página web - <https://pythonhosted.org/pyserial/>

⁵IKPy, página web - <https://github.com/Phylliade/ikpy>

a través del puerto serial COM al que se encuentre conectado el microcontrolador. Las instrucciones de movimiento son generadas y enviadas una vez se resuelve el problema de cinemática inversa.

Como parámetros de entrada, recibe:

- `p_obj`: arreglo NumPy 3x1 que contiene las coordenadas del punto objetivo en milímetros (flotantes).
- `p_ini`: arreglo NumPy 3x1 que contiene las coordenadas de la posición actual del brazo en milímetros (flotantes).
- `thetaini`: lista con los ángulos (flotantes) de la posición actual del brazo en grados en formato $[\theta_1, \theta_2, \theta_3, \theta_4, \theta_5]$.
- `segm`: variable tipo flotante. Magnitud de los segmentos que dividirán la trayectoria en milímetros, explicado más abajo.
- `tol`: variable tipo flotante. Límite superior de la diferencia entre el punto alcanzado `p` y el punto objetivo `p_obj` en milímetros para el que el movimiento sea declarado como exitoso. Esta variable tiene como rol discriminar principalmente entre aquellos movimientos que se dan bajo puntos objetivos difíciles de alcanzar debido a la posición inicial y forma del brazo.
- `com`: variable tipo string. Corresponde al puerto serial COM utilizado por el microcontrolador en formato `'COMX'`, donde X es el número del puerto.
- `baud`: variable tipo entero. Número de baudios que utiliza el puerto serial.

Los parámetros que retorna son:

- `success`: variable booleana. Define si el movimiento tuvo éxito (`True`) o no (`False`) dadas la posición final `p`, el objetivo `p_obj` y la variable de tolerancia `tol`. Esta variable es la que determinará si el agarre puede ser llevado a cabo o no, como se mostrará más adelante en el Capítulo 3.
- `thetaini`: variable tipo lista. Contiene los ángulos θ_i en grados (flotantes) de la posición en la que quedó el brazo luego de ejecutar el movimiento. El formato es el mismo que el de la variable `thetaini`.

- p : arreglo NumPy 3x1. Contiene las coordenadas en milímetros (flotantes) del extremo del brazo luego de ejecutar el movimiento.

Para poder utilizar la función de cinemática inversa de IKPy, se crea la estructura cinemática en formato URDF. Como se observa en el bloque de código comprendido entre las líneas 21 a 59, se crea el origen (`OriginLink()`) y el resto de los sistemas de referencia (`URDFLink()`) utilizando los parámetros de Denavit-Hartenberg obtenidos por medio de la función `par_dh()` del Apartado I de la Sección 2.2.1. [23–26]

Utilizando las variables de entrada p_{obj} , p_{ini} y $segm$ se calcula el vector director en el sentido desde el extremo del brazo hacia el punto objetivo para luego calcular la cantidad de tramos en los que será dividida la trayectoria que deberá seguir el extremo del brazo, según el tamaño del segmento de recta que une a p_{ini} con p_{obj} . Con esta información se calculan tantos movimientos a través de la función de cinemática inversa de IKPy (línea 121), como número de tramos se hayan obtenido. La sucesión de movimientos yacen en una trayectoria fija, pues los puntos intermedios (resultantes de la segmentación) que están entre el punto inicial y el objetivo, a los que debe moverse el extremo del brazo, están siempre sobre la misma recta (la cual une a p_{ini} con p_{obj}), como se muestra en la Figura 2.8.

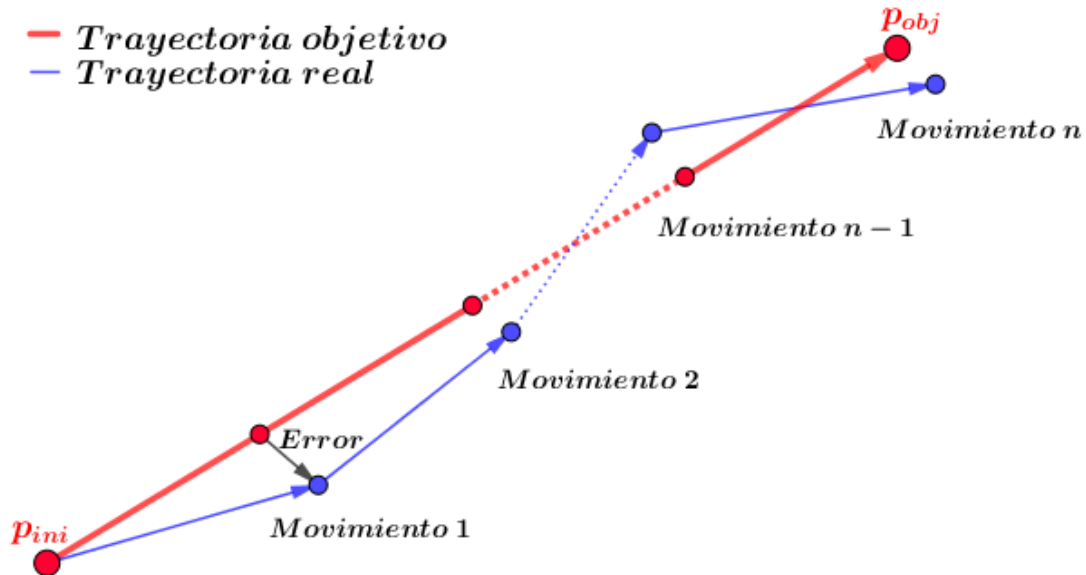


Figura 2.8: Trayectoria y movimiento del brazo generado por la función `ik()`. Elaboración propia.

Se decidió segmentar la trayectoria por tramos para evitar largos movimientos que puedan causar colisiones por parte del brazo consigo mismo o con su entorno.

Después de determinar los ángulos θ_i de un movimiento, estos son interpretados a través de la Ecuación 2.1 como unidades de posición u_i de los servomotores (líneas 139 y 140). Cada vez que se calcula un valor de u_i , este es comparado con $u_{i,min}$ y $u_{i,max}$ (presentados en la Tabla 2.3) entregados por la función `par_fu()` del Apartado II de la Sección 2.2.1: si el valor de u_i se encuentra fuera de rango, se le asigna el valor límite. Luego, los valores u_i son encadenados en un string, generando así una instrucción de movimiento. Estas son almacenadas en una lista que es utilizada al momento de comunicarse con el puerto serial COM para generar el movimiento.

Luego de que el último movimiento se haya calculado, se compara la distancia que existe entre el punto alcanzado `p` y el punto objetivo `p_obj` con la variable de entrada `tol` (líneas 176 a 195). En caso de que esta diferencia sea menor o igual que `tol`, la variable `success` se declara como `True`, permitiendo la posterior simulación de agarre y el código intentará establecer comunicación con el puerto serial COM para enviar las instrucciones mediante las funciones de la librería Serial (líneas 198 a 217). En caso contrario, si la diferencia es mayor que `tol`, no se generaran instrucciones de movimiento y por ende no establecerá comunicación con el puerto serial.

II. Función de cinemática directa

La función `fk()` por “*Forward Kinematics*”, definida en la línea 232 del Anexo A.2, tiene como objetivo elaborar las instrucciones de movimiento para los servomotores directamente desde los ángulos de posición objetivo θ_i y comunicarlás a través del puerto serial COM al que se encuentre conectado el microcontrolador.

Los argumentos de entrada de esta función son:

- `thetai_obj`: variable tipo lista que contiene los ángulos de la posición que se desea alcanzar en formato $[\theta_1, \theta_2, \theta_3, \theta_4, \theta_5]$.
- `com`: variable tipo string. Corresponde al puerto serial COM utilizado por el microcontrolador en formato `'COMX'`, donde X es el número del puerto.

- `baud`: variable tipo entero. Número de baudios que utiliza el puerto serial.

La función retorna:

- `thetai`: variable tipo lista. Contiene los ángulos θ_i en grados de la posición en la que quedó el brazo luego de ejecutar el movimiento. El formato es el mismo que el de la variable `thetai_obj`.
- `p`: arreglo NumPy 3x1. Contiene las coordenadas en milímetros (flotantes) del extremo del brazo luego de ejecutar el movimiento.

La función inmediatamente convierte los ángulos θ_i contenidos en `thetai_obj` a unidades de servomotor u_i mediante la Ecuación 2.1 (Apartado II de la Sección 2.2.1) y los encadena en un string para generar la instrucción de movimiento.

En caso de que algún valor de u_i esté fuera de rango, se detiene la generación de la instrucción y no se produce el movimiento. Por el contrario, si no existen conflictos con los límites, se comunica la instrucción a través del puerto serial COM correspondiente para generar el movimiento de los servomotores.

2.2.3. Control de los servomotores

Este script (presentado en el Anexo A.3) está escrito en Arduino⁶ y utiliza la librería Dynamixel Workbench⁷ para el control de los servomotores Dynamixel AX-12A a través del microcontrolador OpenCM 9.04⁸.

La función de este código es inicializar la actividad del puerto serial COM y los servomotores en modo “Joint” (modo de movimiento por posiciones) con sus respectivas velocidades y recibir las instrucciones generadas por las funciones de cinemática para producir el movimiento. Cada vez que se realiza un movimiento, el código comunica al módulo de cinemática que está listo para recibir la siguiente instrucción.

⁶Arduino, página web - <https://www.arduino.cc/>

⁷Dynamixel Workbench, página web - http://emanual.robotis.com/docs/en/software/dynamixel/dynamixel_workbench/

⁸OpenCM 9.04, página web - <http://emanual.robotis.com/docs/en/parts/controller/opencm904/>

Capítulo 3

Detección de la posición de objetos a través de estereovisión

En este capítulo se tratará la parte del algoritmo que se encarga de calibrar las cámaras y los sistemas de referencias y de encontrar la posición de los objetos a través de las imágenes estéreo.

Las relaciones entre los módulos de este capítulo más la interacción con el algoritmo de movimiento se puede observar en la Figura 3.1.

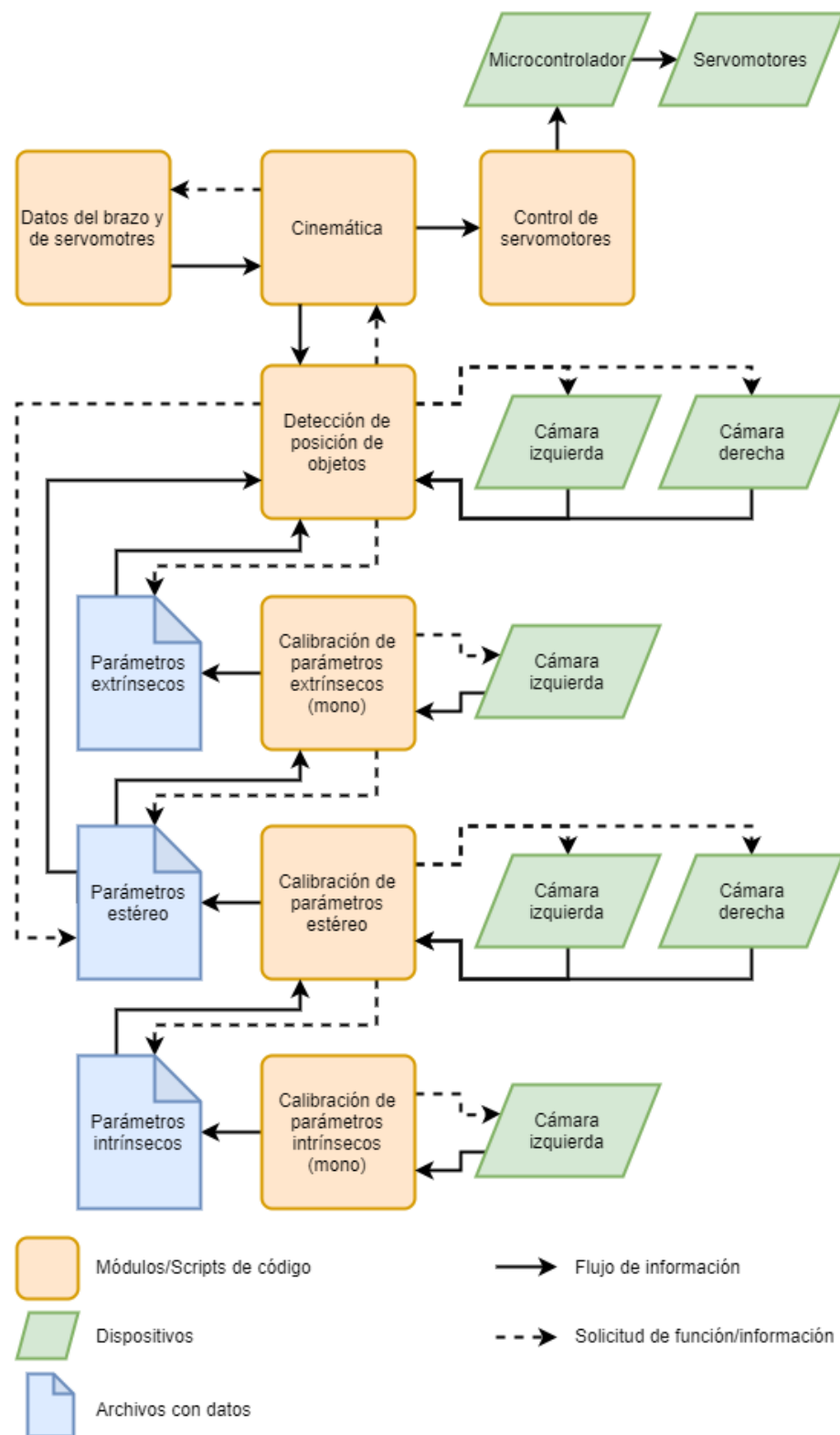


Figura 3.1: Diagrama de flujo del algoritmo completo (calibraciones, detección de posición de objetos y movimiento del brazo). Para las calibraciones “mono” ambas cámaras son iguales, consideración explicada más adelante. Elaboración propia.

3.1. Módulos de calibración

3.1.1. Calibración de parámetros intrínsecos

Este módulo (presentado en el Anexo B.1) se encuentra escrito en Python y utiliza las librerías NumPy y OpenCV (`cv2`). Su función es obtener la matriz de parámetros intrínsecos de las cámaras (matriz 3×3 de la Ecuación 1.8 de la Sección 1.2.1) y los coeficientes de distorsión para corregir los efectos de las distorsiones en la imagen mencionadas en el Apartado I de la Sección 1.2.3 para luego almacenar todo en un archivo `.npz` que será utilizado por el módulo de calibración estéreo que se presentará en la Sección 3.1.2.

La calibración de este módulo solo es necesario realizarla con una de las cámaras, las cuales, en este caso, corresponden a dos Logitech C920¹. Se puede realizar esta simplificación por el simple hecho de que son el mismo producto y siempre y cuando se mantengan los mismos parámetros (como configuración de color, largo focal, entre otros) para ambas cámaras a través del software de control del fabricante, en este caso Logitech Capture². **La cámara escogida para el proceso de calibración es la cámara izquierda.**

La construcción de este módulo toma como base el tutorial de calibración de cámara en Python que ofrece OpenCV en su página web [27] y la referencia [28].

Luego de que el usuario determine el número de dispositivo de la cámara a calibrar (puede ser 0, 1, 2, 3, etc.), deberá ingresar los parámetros referentes a las características del patrón de calibración (líneas 139 a 146), el cual corresponde a un tablero de ajedrez con un tamaño `size` de los cuadros y `nx` y `ny` intersecciones interiores horizontales y verticales respectivamente, como se muestra en la Figura 3.2. Todas las unidades deben estar en milímetros para mantener la consistencia con los valores de los parámetros de Denavit-Hartenberg presentados en la Tabla 2.2 de la Sección 2.1.

¹Logitech C920, página web - <https://www.logitech.com/es-mx/product/hd-pro-webcam-c920>

²Logitech Capture, página web - <https://www.logitech.com/es-es/product/capture>

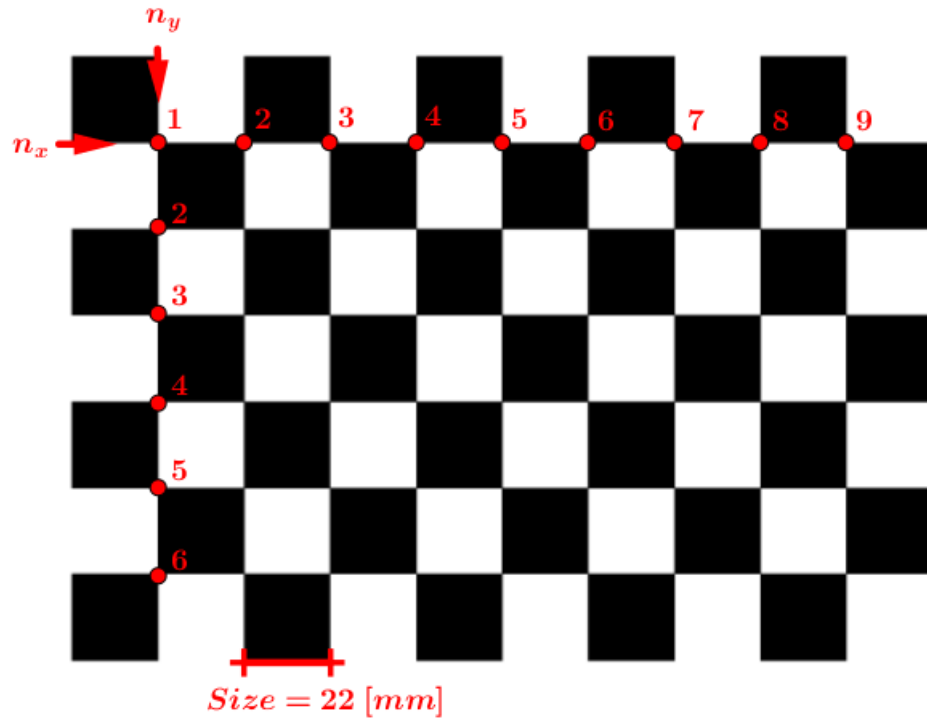


Figura 3.2: Tablero de ajedrez como patrón de calibración y sus parámetros relevantes. En este caso particular, el tamaño de los cuadros `size` tiene un valor de 22 milímetros y las intersecciones interiores `nx` y `ny` corresponden a 9 y 6 respectivamente. [29]

La cantidad mínima de imágenes válidas (esto es, imágenes en que el algoritmo pueda reconocer exitosamente el patrón de calibración) requerida debe ser de 10, según [28].

Luego, a partir de la información proporcionada sobre el patrón de calibración, se crea un arreglo (`ptnpts`) con los puntos de objeto (coordenadas 3D del “mundo real”) del patrón respecto a su propio sistema de referencia (líneas 183 y 184 del código). Estos puntos corresponden a todas las intersecciones interiores del tablero de ajedrez.

La función de OpenCV que permite reconocer el patrón del tablero de ajedrez es `findChessboardCorners()` [30]. Esta función, luego de analizar la imagen en escala de grises (línea 227), entrega los pixeles en los que se encuentran las intersecciones interiores del patrón. Cada vez que se identifica el patrón con éxito se almacenan sus puntos de objeto (los mismos de la variable `ptnpts`) en la variable `objpts` al mismo tiempo que se almacenan los puntos en coordenadas de imagen (pixeles) en la

variable `imgpts` (líneas 234 a 237), tal que se encuentren en el mismo orden y queden asociados los puntos físicos con los de imagen (ver Figura 3.3).

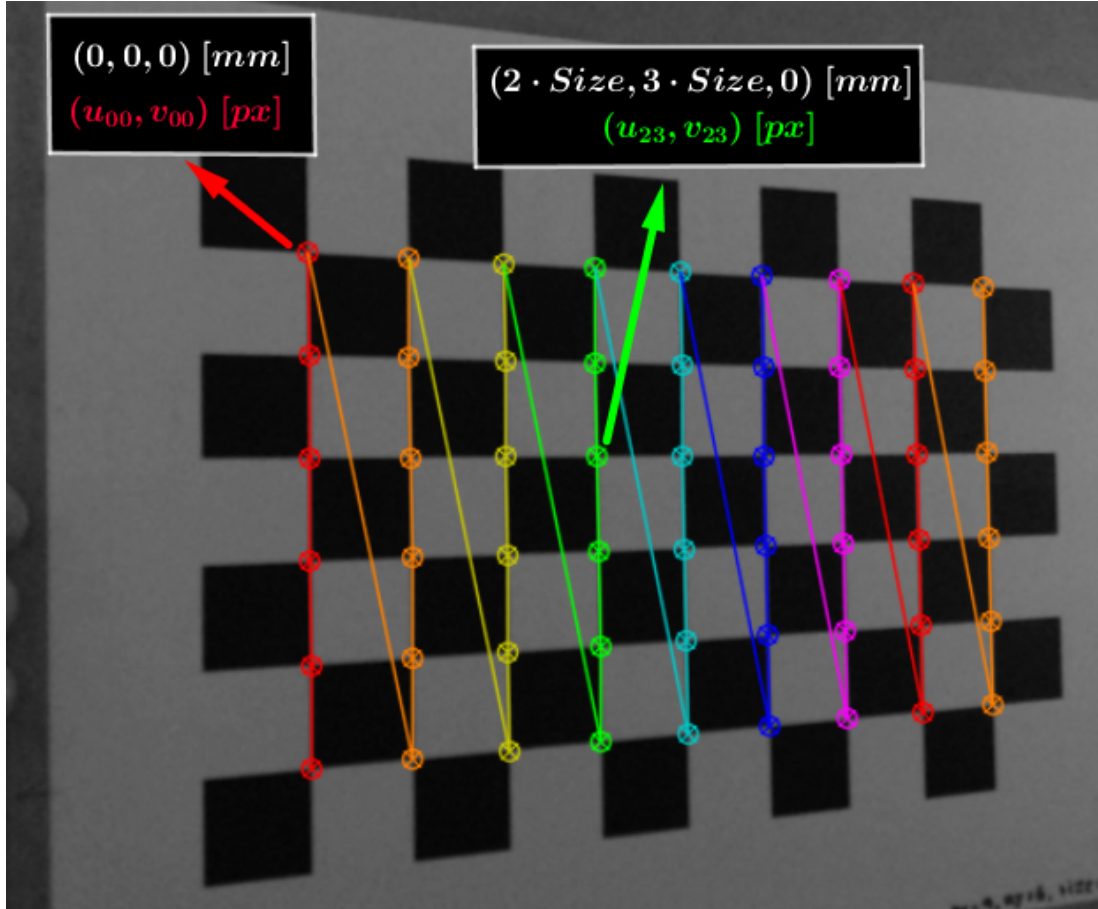


Figura 3.3: Identificación de las intersecciones interiores del patrón de calibración (puntos de colores) y asociación de los puntos de objeto en milímetros (color blanco) con los puntos de imagen (color rojo y verde) (u_{yx}, v_{yx}) en píxeles ([px]). Elaboración propia.

Luego de haber recopilado tantas veces los puntos en las variables `objpts` y `imgpts` como imágenes válidas el usuario haya definido, se obtiene la matriz de parámetros intrínsecos `imtx` y los coeficientes de distorsión `distc` a través de la función de OpenCV `calibrateCamera()` (línea 310) ya tratada en el Apartado I de la Sección 1.2.3. Estas dos variables son almacenadas en un archivo `.npz` (línea 317) para su posterior uso, siempre que se cumpla que el error de reproyección RMS entregado por la misma función sea menor a 0,5 píxeles (línea 312), según lo recomendado en [28].

3.1.2. Calibración de parámetros estéreo de las cámaras

Este módulo de calibración (ver Anexo B.2) se encuentra escrito en Python y utiliza las librerías NumPy, OS³ (para el manejo de directorios) y OpenCV (cv2). Su función corresponde a la de obtener los mapas de pixeles necesarios para la rectificación de las imágenes de ambas cámaras (ver Apartado III de la Sección 1.2.3), los parámetros c_u y c_v de las Ecuaciones 1.9 y 1.10 de la Sección 1.2.1 respectivamente y los parámetros B (distancia de la línea base) y f (largo focal), presentados en la Ecuación 1.12 de la Sección 1.2.2, para luego almacenarlos en un archivo .npz. Los mapas de pixeles serán utilizados tanto por el módulo de calibración de parámetros extrínsecos (Sección 3.1.3) como por el algoritmo de detección de posición de objetos (Sección 3.2) y c_u , c_v , B y f serán utilizados por este último para el cálculo de las coordenadas X , Y y Z del objeto.

Para obtener los parámetros mencionados en el párrafo anterior, primero se debe encontrar la relación geométrica que existe entre ambas cámaras, es decir, la matriz de rotación R y el vector de traslación $\vec{T}$ a través de la función `stereoCalibrate()` (ver Apartado II de la Sección 1.2.3).

El proceso de encontrar R y $\vec{T}$ es el mismo que se describió en la Sección 3.1.1 para la calibración de los parámetros intrínsecos, pero esta vez se debe presentar el patrón simultáneamente a ambas cámaras. Por cada par de imágenes en la que se encuentra el patrón exitosamente, a través de la función `findChessboardCorners()`, se almacenan los puntos de objeto del patrón (puntos físicos) en la variable `objpts` y los puntos de imagen (coordenadas en pixeles) para las cámaras izquierda y derecha en las variables `l_imgpts` y `r_imgpts` respectivamente, de tal forma que los puntos queden asociados entre sí (líneas 270 a 284 del código).

Luego de haber almacenado tantas veces los puntos en las variables mencionadas en el párrafo anterior, como imágenes válidas el usuario definió, se cargan la matriz de parámetros intrínsecos (`imtx`) y los coeficientes de distorsión (`distc`) desde del archivo .npz que se haya generado a partir de la calibración de parámetros intrínsecos (líneas 311 a 321) y se ingresan todas estas variables a la función `stereoCalibrate()` para obtener la matriz de rotación R (variable `R`) y el vector de traslación $\vec{T}$ (variable `T`) (líneas 325 y 332).

³OS, página web - <https://docs.python.org/3/library/os.html>

A partir de $\vec{T}$ se puede obtener B en milímetros como:

$$B = \sqrt{(T_x)^2 + (T_y)^2 + (T_z)^2} \quad (3.1)$$

donde T_x , T_y y T_z corresponden a las componentes de $\vec{T}$ en x , y y z respectivamente (líneas 335 a 339).

Mediante el uso de las variables `imtx`, `distc`, `R` y `T` y la función de OpenCV `stereoRectify()` (presentada en el Apartado III de la Sección 1.2.3) se obtienen las matrices de rotación (`l_R` y `r_R`) y matrices aparentes (`l_P` y `r_P`) de las cámaras izquierda y derecha respectivamente para alinear los sistemas de referencia (rectificación) (líneas 343 a 349). Las variables c_u , c_v y f serán obtenidas a partir de la matriz de cámara aparente izquierda `l_P` [14].

A través de la función `initUndistortRectifyMap()` (presentada en Apartado III, Sección 1.2.3) y las matrices obtenidas desde la función `stereoRectify()`, se logran encontrar los mapas de pixeles para la rectificación de las imágenes de la cámara izquierda (variables `l_map1` y `l_map2`) y cámara derecha (variables `r_map1` y `r_map2`) (líneas 353 y 354).

Finalmente, los mapas de pixeles, la matriz aparente de la cámara izquierda y la distancia de la línea base son almacenadas en un archivo `.npz` para su posterior uso (líneas 383 a 385).

Para tener un punto de refencia sobre si la calibración estéreo a través de la función `stereoCalibrate()` ha sido buena, se calcula el error RMS a través de la matriz fundamental F (variable `F` de la línea 333) según la ecuación:

$$[p_r|1]^T \cdot F \cdot [p_l|1] = 0 \quad (3.2)$$

donde p_l y p_r son los puntos de imagen correspondientes a la imagen izquierda y derecha respectivamente [31] (líneas 362 a 376).

3.1.3. Calibración de parámetros extrínsecos

Este módulo (presentado en el Anexo B.3) está escrito en Python y hace uso de las librerías NumPy, OS, OpenCV (`cv2`) y ArUco⁴ (`cv2.aruco`). Los marcadores ArUco son marcadores cuadrados binarios (ver Figura 3.4). El uso del marcador ArUco jugará un papel principal en el cumplimiento del objetivo de este módulo de calibración.

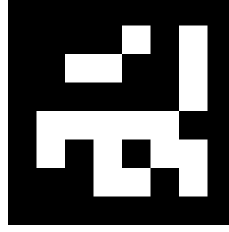


Figura 3.4: Marcador ArUco de 6x6 (cuadros internos). [29]

La función de este módulo es obtener la matriz de transformación homogénea que contiene los *parámetros extrínsecos* (esto es, parámetros que son externos a la cámara), los cuales corresponden a las componentes de la submatriz de rotación 3x3 y a las del vector de traslación 3x1 que componen la matriz de transformación homogénea y que describen la rotación y traslación del sistema de referencia de la cámara izquierda con respecto del sistema del brazo robótico. Esta matriz transforma las coordenadas obtenidas desde el sistema referencia de la cámara izquierda (rectificado) a las coordenadas del sistema de referencia del brazo robótico (base del brazo), para que este pueda realizar el movimiento de alcance y agarre al calcularse la posición del objeto desde el módulo de detección de posición de objetos.

El modelo propuesto para poder encontrar la matriz de transformación homogénea que lleva las coordenadas desde el sistema de referencia de la cámara al del brazo es el siguiente:

$${}^YHT_{l,rect} = {}^YHT_C \cdot {}^CHT_{l,rect} \quad (3.3)$$

donde ${}^YHT_{l,rect}$ corresponde a la matriz que transforma las coordenadas desde el sistema de referencia de la cámara izquierda $O_{l,rect}$ al del brazo O_Y , ${}^YHT_C$ es la que lleva las

⁴OpenCV, ArUco marker detection, página web - https://docs.opencv.org/master/d9/d6d/tutorial_table_of_content_aruco.html

coordenadas medidas desde el sistema de referencia del patrón de calibración (ArUco) O_C al sistema del brazo y ${}^C HT_{l,rect}$ es la matriz que transforma las coordenadas medidas desde la cámara izquierda a coordenadas medidas desde el patrón de calibración.

Para que la calibración sea exitosa se deben cumplir dos condiciones con respecto a la disposición física del patrón de calibración:

1. El eje $Z+$ del sistema de referencia del marcador ArUco debe tener la misma dirección y sentido que el eje $Y-$ del sistema de referencia del brazo.
2. El eje $Y+$ del sistema de referencia del marcador ArUco debe tener la misma dirección y sentido que el eje $Z+$ del sistema de referencia del brazo.

En la Figura 3.5 se observa la configuración recién mencionada.

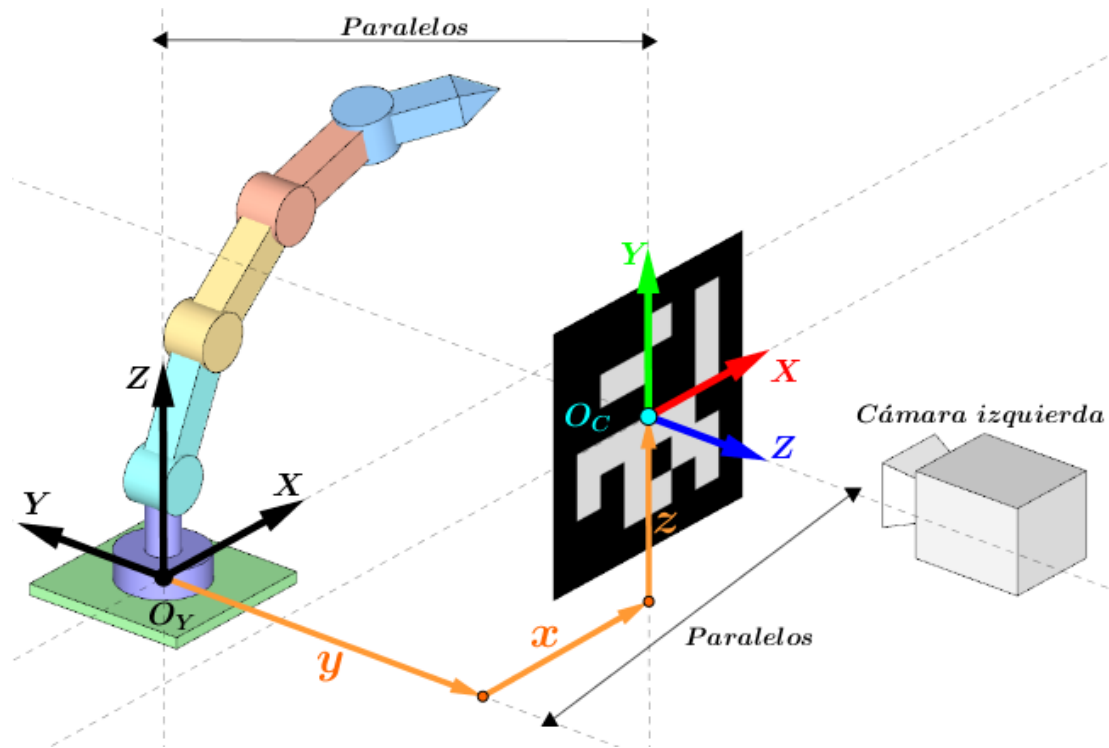


Figura 3.5: Sistemas de referencia y disposición física del brazo robótico y del marcador ArUco para la calibración. Las variables x , y y z en color naranja representan la posición, en las respectivas direcciones, del sistema O_C respecto de O_Y . Elaboración propia.

Para encontrar la matriz de transformación homogénea ${}^C HT_{l,rect}$, se estima la pose (proceso abordado en el Apartado IV de la Sección 1.2.3) del marcador ArUco haciendo uso de las funciones de la librería ArUco de OpenCV. Se tomó como base la implementación en Python que se hace en la referencia [32].

La única característica necesaria a ingresar respecto del patrón (marcador ArUco) para la estimación de pose es el largo de borde externo `size` (línea 15 del código).

Luego de que el usuario haya ingresado el valor para `size`, se carga la matriz aparente de la cámara izquierda `imtx_rect`⁵ y los mapas de píxeles para la rectificación de la imagen izquierda `map1` y `map2` desde el archivo `.npz` generado por el módulo de calibración de parámetros estéreo (líneas 20 a 33) y se declaran las variables de la librería ArUco (líneas 54 a 57).

El código, en las líneas 67 a 76, procede a encontrar las intersecciones interiores como coordenadas en píxeles, el número de identificación y puntos que se han rechazado correspondientes al marcador ArUco (función `detectMarkers()` [33] de la librería ArUco) a partir de la imagen en escala de grises rectificada a través de la función `remap()` (ver Apartado III de la Sección 1.2.3). Después, con esta información más la variable `imtx_rect`, se estima la pose (vector de traslación `tvec` y rotación en ángulos de Euler `rvec`) del marcador ArUco respecto de la cámara izquierda a través de la función de la librería ArUco `estimatePoseSingleMarkers()` [34] (líneas 83 a 86), la cual es una función diseñada para los marcadores ArUco que cumple el mismo rol que `solvePnP()` (enunciada en el Apartado IV de la Sección 1.2.3). En la Figura 3.6 se puede apreciar la identificación del marcador ArUco y su sistema de referencia.

En la línea 153 se obtiene la submatriz de rotación 3x3 a través de la función de OpenCV `Rodrigues()` [35] y en las líneas 156 a 161 se crea la matriz ${}^C HT_{l,rect}$ en la variable `HT_Olrect_Oc`.

⁵Esta variable, a pesar de estar bajo el mismo nombre que la matriz `l_P` obtenida mediante la función `stereoRectify()` en la Sección 3.1.2, difiere ligeramente en cuanto a estructura, pues esta última contiene una columna adicional (3x4) [14]. La matriz `imtx_rect` posee la misma estructura que la matriz de parámetros intrínsecos de tamaño 3x3.

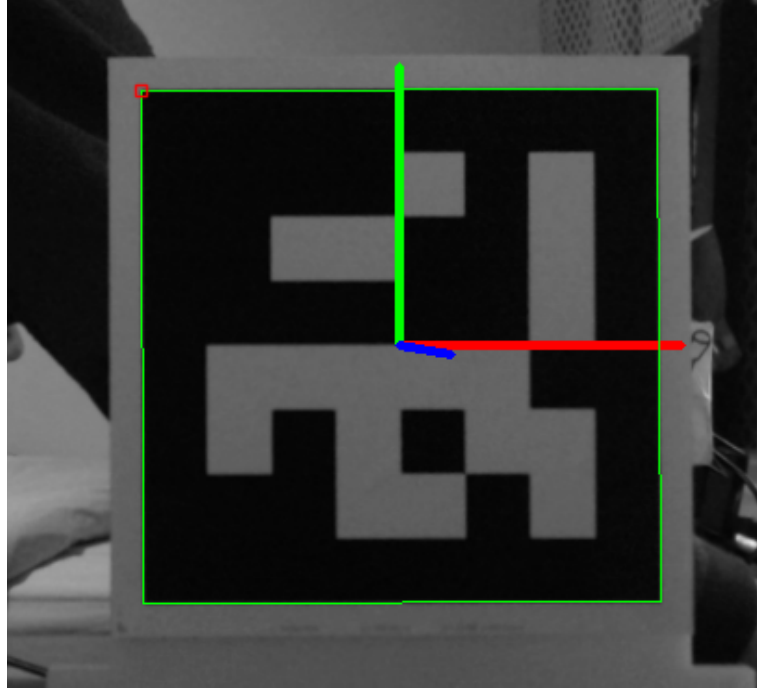


Figura 3.6: Estimación de pose de marcador ArUco 6x6. Los segmentos que se encuentran en el centro del marcador representan a sus ejes coordenados (rojo para el eje X , verde para el eje Y y azul para el eje Z) y el borde verde que encierra al marcador representa su reconocimiento en la imagen. Elaboración propia.

Para obtener la matriz ${}^YHT_C$, se propuso que el usuario deba medir e ingresar las distancias en los ejes X , Y y Z (x , y y z , según la Figura 3.5), respecto del sistema coordenado del brazo, que existen entre el origen de este último y el del sistema coordenado del marcador ArUco (líneas 116 a 118), pudiéndose así obtener el vector de traslación. En cuanto a la submatriz de rotación, con la configuración propuesta, su cálculo queda resuelto. Con toda esta información se obtiene la matriz de transformación homogénea ${}^YHT_C$ (variable `HT_Oc_Oy`) (líneas 164 a 174).

Con las matrices de transformación ${}^YHT_C$ y ${}^CHT_{l,rect}$ conocidas, se calcula ${}^YHT_{l,rect}$ (variable `HT_Ol_rect_Oy`) según la Ecuación 3.3 (línea 177) y se almacena en un archivo `.npz` (línea 181) para su uso en el módulo de detección de posición de objetos.

3.2. Algoritmo de detección de posición de objetos

Este algoritmo (presentado en el Anexo B.4) corresponde al módulo principal de todo el programa. Se encuentra escrito en Python y hace uso de las librerías NumPy, OS, OpenCV (`cv2`) y Time⁶ (utilizado para los movimiento de agarre y de llevar el objeto a una posición definida). Además, integra las funciones `ik()` y `fk()` del módulo de cinemática (Sección 2.2.2) para la ejecución del movimiento del brazo robótico.

El objetivo de este módulo es encontrar, a través de la información capturada por las cámaras, la posición de un objeto con respecto al sistema de referencia del brazo y enviar las órdenes para la ejecución del movimiento para simular el agarre. El algoritmo está diseñado para identificar la posición de **objetos esféricos** y de **diámetro conocido**.

Hasta este punto, para poder calcular la posición del objeto de interés según las Ecuaciones 1.9 y 1.10 de la Sección 1.2.1 y la Ecuación 1.12 de la Sección 1.2.2, solo falta la disparidad d . El valor de esta variable es encontrado través de la generación de un mapa disparidad, el cual contiene los valores de disparidad para cada pixel de la imagen. Para poder construir este mapa se utilizará uno de los métodos ofrecidos por OpenCV llamado **Semi-Global Block Matching** (abreviado como **SGBM**) en conjunto con el **Filtro de disparidad WLS** (por *Weighted Least Squares* en inglés) perteneciente a la librería de contribución de OpenCV.

El SGBM es uno de los dos algoritmos de correspondencia estéreo (detectar coincidencia de un punto 3D en base a dos imágenes en las cuales la escena se traslapa) que ofrece OpenCV. Su función es convertir dos imágenes en una sola que represente la profundidad para cada pixel. Las diferencias con respecto del otro algoritmo (Block Matching o BM) es que es más confiable y preciso, pero más costoso. [36]

El algoritmo SGBM corresponde a una modificación del algoritmo Semi-Global Matching propuesto por H. Hirschmüller (ver referencia [37] para mayor información sobre SGM). Las diferencias del SGBM con respecto del SGM son que considera menos direcciones (5 en vez de 8), analiza bloques de pixeles y no pixeles individuales para encontrar coincidencias, incluye etapas de pre- y post-procesado y reemplaza la función de costo original. [38]

⁶Time, página web - <https://docs.python.org/3/library/time.html>

El Filtro de disparidad WLS se encuentra basado en el filtro llamado Weighted Least Squares Filter, pero aplicando la forma de Fast Global Smoother (ver referencia [39] para mayor información), resultando así mucho más rápido que el tradicional Weighted Least Squares Filter. El Filtro de disparidad WLS tiene como objetivo refinar y propagar los valores de disparidad en zonas del mapa de disparidad en donde se hayan producido errores debido a áreas con bajo nivel de textura, half-occlusions (ambas cámaras están expuestas a la misma escena, pero un punto en particular se ve desde una y no desde la otra debido a algún obstáculo) y a regiones donde existen discontinuidades de profundidad (ver Figura 3.7a). Con este filtro se mejoran los resultados obtenidos desde el SGBM (ver Figura 3.7b). [40] [41]

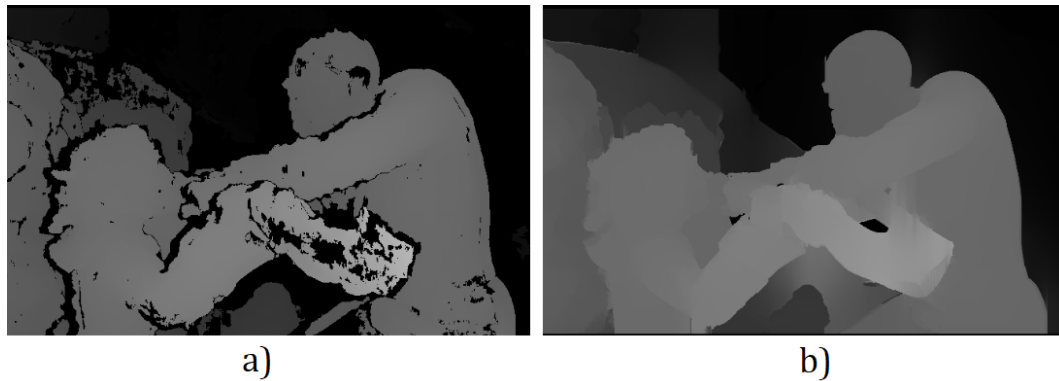


Figura 3.7: (a) Mapa de disparidad sin filtrar. Los elementos más cercanos a la cámara están representado por un color claro y los más lejanos por un color oscuro. (b) Corrección del mapa de disparidad realizada por el Filtro de disparidad WLS. [41]

3.2.1. Descripción del código

El módulo cuenta con un panel de deslizadores (ver Figura 3.8) que permite al usuario modificar los valores de los parámetros asociados al SGBM, Filtro de disparidad WLS e identificación de figuras conforme las características de la escena cambien (iluminación, distancia de los objetos respecto de las cámaras, colores existentes en la escena, etc.). Todos los parámetros se mantienen almacenados en un archivo .npz con el fin de que el usuario pueda guardar los cambios si lo desea. Los deslizadores son creados a través de la función de OpenCV `createTrackbar()` [42] (ver líneas 177 a 214) y en las líneas 524 a 636 se realiza la actualización de sus variables mediante el uso de la función `getTrackbarPos()` [43].

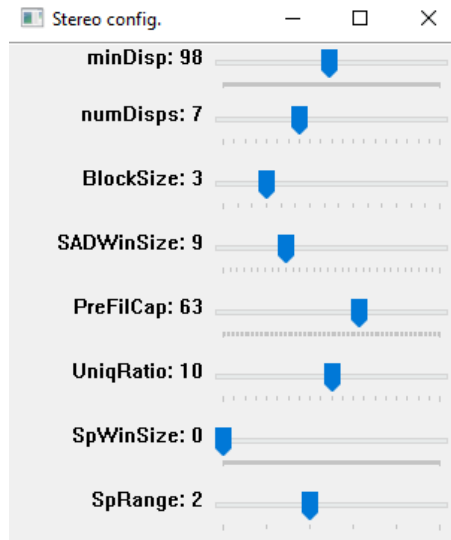


Figura 3.8: Sección del panel de deslizadores para configuración de parámetros del módulo. Elaboración propia.

Cuando el módulo comienza, se lleva al brazo a una posición de reposo dictada por los ángulos contenidos en la variable `thetai_reset` a través de la función de cinemática directa `fk()` (línea 26), detallada en el Apartado II de la Sección 2.2.2. Luego, son cargados todos los parámetros almacenados en los archivos generados desde las calibraciones abordadas en las secciones anteriores de este capítulo (líneas 30 a 62): mapas de rectificación para ambas cámaras (`l_map1`, `l_map2`, `r_map1` y `r_map2`), largo focal f (`f`), coordenadas del centro óptico en píxeles c_u (`l_cx`) y c_v (`l_cy`) de la imagen izquierda, distancia de la línea base B (`B`) y la matriz de transformación homogénea ${}^YHT_{l,rect}$ (`HT_Olrect_Oy`) para llevar las coordenadas desde el sistema de referencia de la cámara izquierda al sistema de la base del brazo.

Todo lo que se refiere al uso del SGBM en conjunto con el Filtro de disparidad WLS en este código está basado en la implementación en Python de la referencia [44].

En las líneas 137 a 154 se inicializan el SGBM y el Filtro de disparidad WLS a través de `StereoSGBM_create()` [45] y `createDisparityWLSFilter()` [46] respectivamente y se crean los *matchers*, los cuales tienen la función de encontrar las coincidencias en las imágenes, en las variables `l_matcher` y `r_matcher` para las imágenes izquierda y derecha respectivamente.

Una vez se inicia la captura de imágenes desde las cámaras y comienza el bucle de detección de posición de objetos (se analiza un par de imágenes por cada ciclo del bucle), se rectifican las imágenes izquierda y derecha con el uso de la función `remap()` (ver Apartado III de la Sección 1.2.3) y los mapas de rectificación correspondientes y luego éstas son transformadas a escalas de grises (`l_gray_rect` para la imagen izquierda y `r_gray_rect` para la derecha) (líneas 293 a 301). Con estas imágenes y con la aplicación del filtro, se obtiene el mapa de disparidad `disp` en base a la imagen de la cámara izquierda (líneas 304 a 312). También, en las líneas 315 a 320, se genera una versión normalizada (tal que los valores queden dentro de los límites del canal de los pixeles, en este caso de 0 a 255) del mapa de disparidad (`disp_norm`) que tiene solo el fin de poder ser presentado gráficamente (ver Figura 3.9).

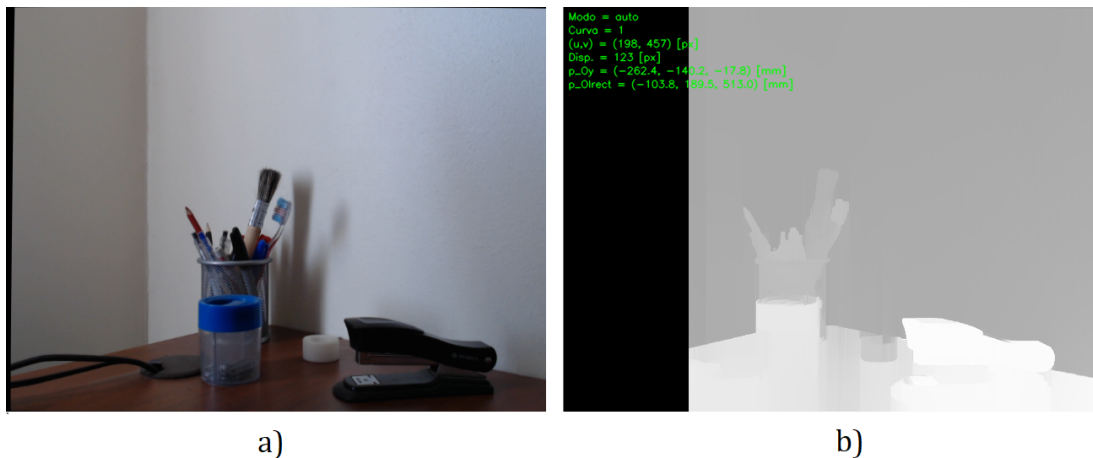


Figura 3.9: (a) Imagen izquierda rectificada. (b) Representación gráfica del mapa de disparidad. Las zonas más claras representan a los elementos más cercanos a las cámaras, mientras que las regiones más oscuras representan a los más lejanos. Elaboración propia.

A modo de aclaración, en el código existe una variable llamada `mode`, la cual puede tomar dos valores: `'manual'` o `'auto'`. El primero le permite al usuario poder seleccionar manualmente un pixel de la imagen para encontrar su posición y solo tiene como propósito hacer pruebas, por lo que no será abordado. El segundo valor se refiere a que el algoritmo encuentra la posición del objeto de forma automática y es el que será detallado en esta sección.

Ya con el mapa de disparidad generado, se procede a identificar la forma del objeto de interés. La forma propuesta es a través de la detección de la curva que encierra a la figura del objeto proyectada en la imagen. Para encontrar la figura en la imagen se descartan los pixeles cuyo color esté fuera de un rango establecido.

Para obtener la figura del objeto, la imagen izquierda rectificada a color es transformada, pasando desde canales BGR (*Blue, Green, Red*) a canales HSV (*Hue, Saturation, Value*) (línea 333). Luego, utilizando los valores de color HSV límites (`lowerHSV` para el mínimo y `upperHSV` para el máximo) escogidos por el usuario desde el panel de deslizadores, se crea una imagen binaria (`l_hsv_mask`) a través de la función `inRange()` [47] (líneas 336 a 341). Esta imagen binaria representa a aquellos pixeles que se encuentran dentro de los límites HSV en color blanco y a aquellos que se encuentran fuera, en color negro (ver Figura 3.10).

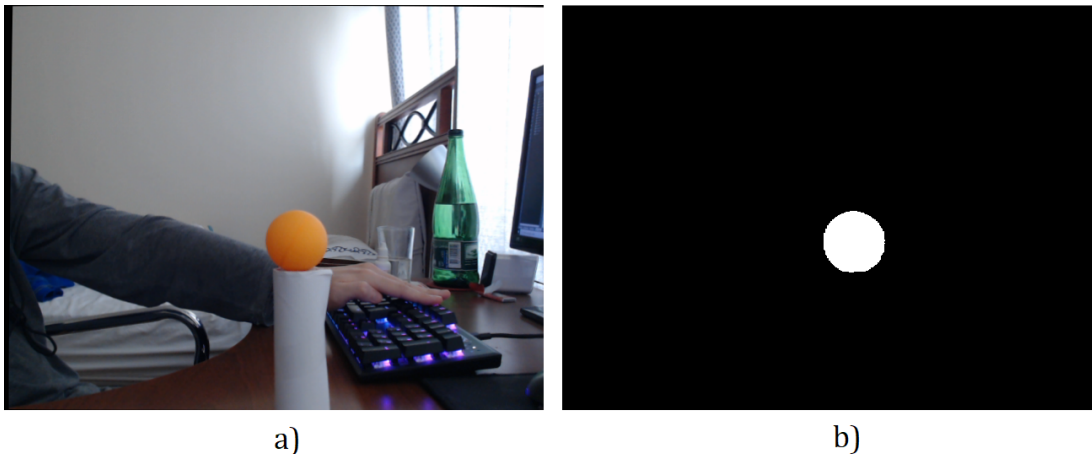


Figura 3.10: (a) Imagen izquierda rectificada a color con objeto de interés (esfera color naranja). (b) Imagen binaria con figura del objeto luego de aplicar la función `inRange()`. Elaboración propia.

Para encontrar la curva que encierra a la figura del objeto de interés se obtienen, primero, todos los contornos de las regiones en blanco de la imagen binaria `l_hsv_mask` a través de la función de OpenCV `findContours()` [48] (línea 344). Luego, todos los contornos son aproximados a polígonos según el nivel de precisión `prec_lvl` definido por el usuario en el panel de deslizadores (líneas 353 a 355); entre mayor sea el valor de `prec_lvl`, mayor será la cantidad de lados de los polígonos cuando se aproximan curvas. Todos los polígonos serán considerados contornos válidos (que tengan

una forma lo más cercana a un círculo) si cumplen tener a lo menos `min_sides` lados (parámetro también definido por el usuario desde el panel de deslizadores). Todos aquellos que sean válidos se almacenan en la lista `validCurves` (líneas 357 a 359).

La posibilidad de ajustar los valores de los parámetros `prec_lvl` y `min_sides` tiene la finalidad de poder eliminar, en la medida de lo posible, aquellos contornos correspondientes a regiones no deseadas que hayan sido consideradas como válidas en la imagen binaria debido a los colores presentes en la escena (colores similares al del objeto de interés debidos al color natural de los elementos, por efecto de la reflexión de la luz sobre sus superficies, por el filtro de colores de las cámaras, etc.).

Ahora que se han encontrado los contornos presentes en la imagen, se calculan las posiciones de sus centros geométricos (en la imagen izquierda y en píxeles) a través del uso de la función `moments()` [49] de OpenCV y son almacenadas en la lista `centroids` (líneas 362 a 380) para luego encontrar sus valores de disparidad (`disp_uv`) a partir del mapa de disparidad `disp` según las coordenadas (u, v) de la imagen que tengan y almacenarlos en la variable `centdisps` (líneas 383 a 410).

Utilizando el teclado numérico (números del 1 al 5), el usuario puede seleccionar la curva del objeto que desea alcanzar, extrayéndose las coordenadas (u y v) y el valor de disparidad (`disp_uv`) en píxeles de su centro geométrico (líneas 413 a 415). Esta opción se ha implementado con dos fines: (1) permitir elegir entre múltiples objetos para simular el agarre y (2) poder sortear curvas no deseadas producto de las condiciones de la escena (ya explicado en uno de los párrafos anteriores). En la Figura 3.11 se puede observar lo descrito.

Con las variables calculadas hasta ahora, ya se puede determinar el punto de agarre (o posición del objeto). Para ello, se realizan una serie de operaciones con vectores para primero encontrar el centro del objeto esférico y luego el punto que deberá alcanzar el brazo robótico.

Como el centro geométrico de la figura proyectada del objeto sobre el plano de la imagen comparte la misma recta que el centro de la esfera (recta trazada desde el origen del sistema de referencia de la cámara izquierda, según del modelo de la cámara estenopeica, abordado en la Sección 1.2.1), se pueden determinar fácilmente las coordenadas

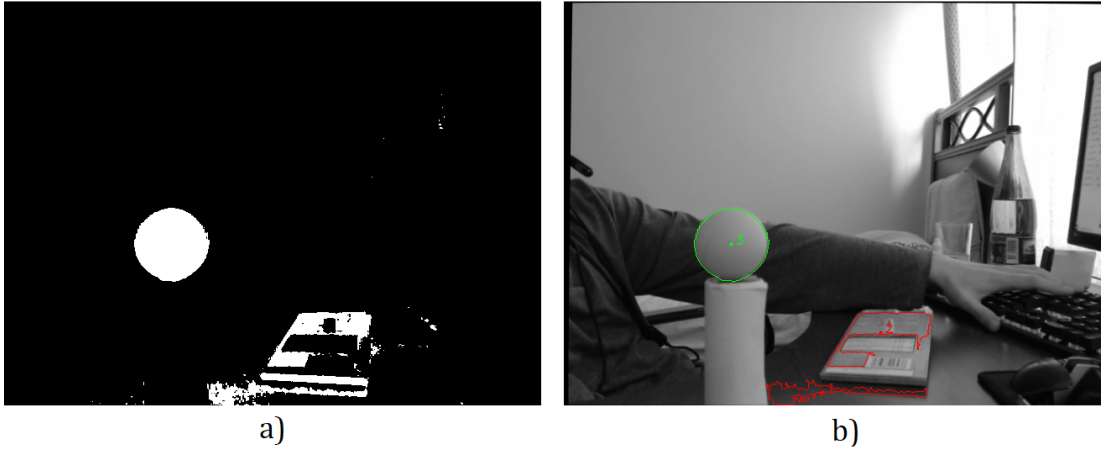


Figura 3.11: (a) Imagen binaria con figura del objeto de interés (círculo blanco) y regiones no deseadas. (b) Imagen con curvas encontradas. La curva de color verde corresponde a la seleccionada por el usuario, mientras que las rojas representan al resto de las curvas encontradas. Elaboración propia.

de este último. El valor de disparidad del punto de imagen que corresponde al centro geométrico de la figura proyectada representa al punto que yace en la superficie de la esfera, por lo que para encontrar la posición del centro de ésta solo se deberá sumar el valor de su radio en la dirección de la recta mencionada y en el sentido que va desde la cámara izquierda hacia el punto de la superficie.

Para encontrar las coordenadas del centro de la esfera, primero se utilizan las Ecuaciones 1.9 y 1.10 de la Sección 1.2.1 y la Ecuación 1.12 de la Sección 1.2.2 para el cálculo de las coordenadas del punto que yace sobre la superficie de la esfera respecto del sistema de referencia de la cámara izquierda (líneas 447 a 449 del código). Luego, en las líneas 453 a 462 se procede de la siguiente manera: se calcula el vector director $\vec{d}_{l,rect}$, el cual se encuentra en la recta que une al origen del sistema coordenado de la cámara izquierda con el punto recién calculado, como:

$$\vec{d}_{l,rect} = \frac{\vec{p}_{sup,lrect}}{\|\vec{p}_{sup,lrect}\|} \quad (3.4)$$

donde $\vec{p}_{sup,lrect}$ corresponde al vector formado por las coordenadas del punto de la superficie de la esfera. Luego, se obtiene el vector del centro del objeto esférico $\vec{p}_{centro,lrect}$ como:

$$\vec{p}_{centro, lrect} = \vec{p}_{sup, lrect} + \frac{D}{2} \cdot \vec{d}_{l, rect} \quad (3.5)$$

donde D representa el diámetro del objeto. En la Figura 3.12 se muestra el cálculo descrito.

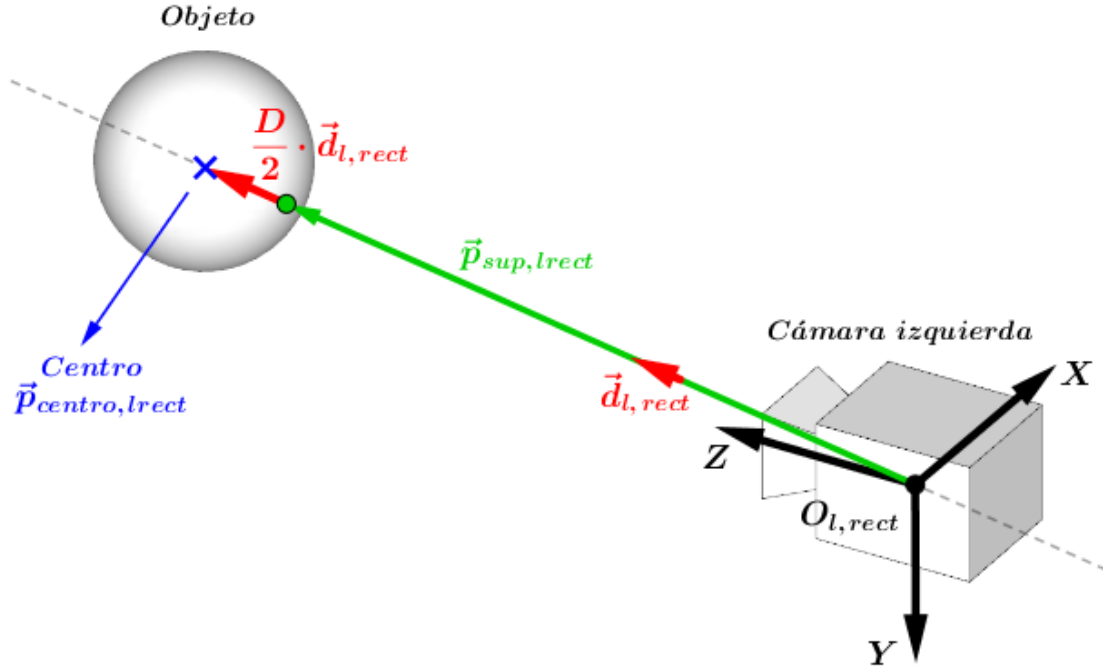


Figura 3.12: Determinación del centro del objeto esférico con respecto del sistema coordenado de la cámara izquierda. Elaboración propia.

En las líneas 464 a 471, las coordenadas del vector $\vec{p}_{centro, lrect}$ (p_Olrect) son transformadas al sistema coordenado del brazo a través de la matriz de transformación homogénea ${}^YHT_{l, rect}$ (HT_Olrect_Oy), como sigue:

$$[\vec{p}_{centro, Y}|1] = {}^YHT_{l, rect} \cdot [\vec{p}_{centro, lrect}|1] \quad (3.6)$$

donde $[\vec{p}_{centro, Y}|1]$ y $[\vec{p}_{centro, lrect}|1]$ corresponden a los vectores columna con las coordenadas del centro del objeto con respecto del sistema de referencia del brazo y de la cámara izquierda respectivamente, ambos ampliados con 1.

Ahora que se ha calculado el punto del centro de la esfera con respecto al sistema coordenado del brazo robótico, se procede a encontrar el punto de agarre. Para ello, se obtiene el vector director que va en la misma dirección y sentido que el vector formado por las coordenadas x e y de $\vec{p}_{centro,Y}$ (líneas 475 a 482):

$$\vec{d}_Y = \frac{x_{centro,Y} \cdot \hat{x} + y_{centro,Y} \cdot \hat{y} + 0 \cdot \hat{z}}{\sqrt{(x_{centro,Y})^2 + (y_{centro,Y})^2}} \quad (3.7)$$

Luego, se obtiene el vector del punto de agarre $\vec{p}_Y$ (`p_OY` en el código, línea 484) como sigue:

$$\vec{p}_Y = \vec{p}_{centro,Y} - \frac{D}{2} \cdot \vec{d}_Y \quad (3.8)$$

En la Figura 3.13 se aprecia gráficamente el cálculo realizado.

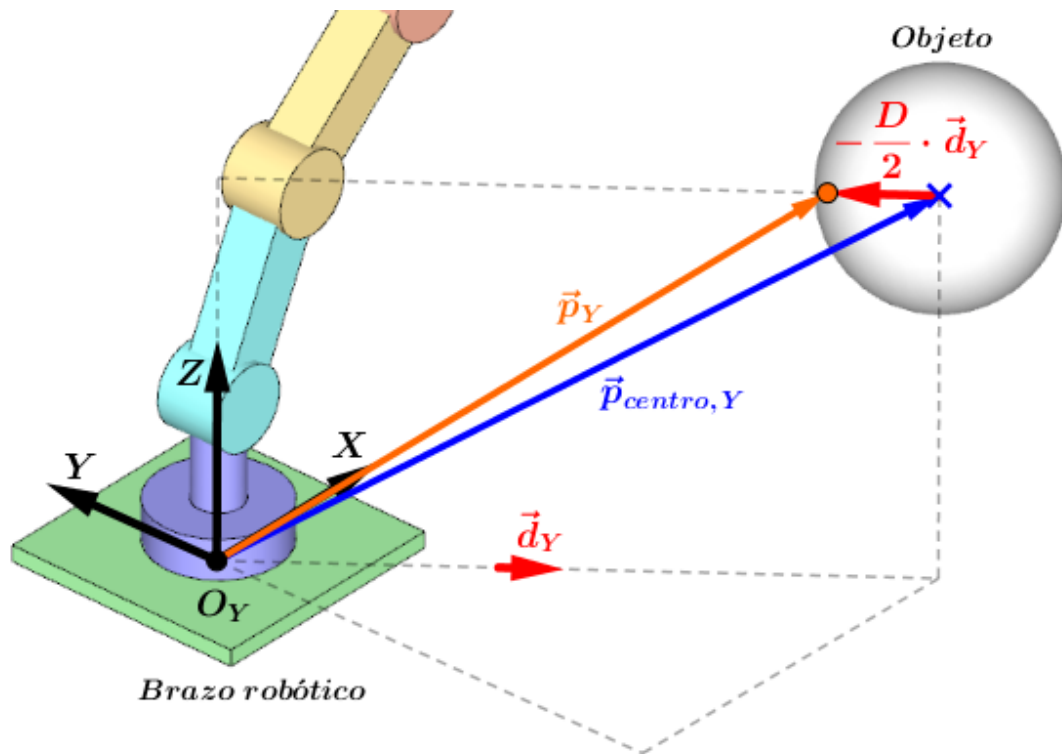


Figura 3.13: Determinación del punto de agarre del objeto esférico respecto del sistema de referencia del brazo robótico. Elaboración propia.

Con el punto de agarre, si el usuario presiona la tecla *Enter*, se ejecuta el movimiento de alcance hacia el punto de agarre `p_Oy` a través de la función de cinemática inversa `ik()` (detallada en el Apartado I de la Sección 2.2.2) (líneas 706 a 708). Si la variable de retorno `success` resulta `True` (esto es, que el brazo pudo alcanzar con éxito el punto de agarre), al presionarse la *Barra espaciadora* se simulará el agarre dejando de manera representativa el objeto en una posición determinada por los ángulos contenidos en `thetai_bring` y luego volverá a su posición de reposo según los ángulos de `thetai_reset` (líneas 710 a 736); ambos movimientos se realizan por medio de la función de cinemática directa `fk()`.

Conclusiones

Con el método de Denavit-Hartenberg, se determinó la estructura cinemática del brazo robótico utilizado, el cual se encuentra compuesto por 5 grados de libertad, todos en forma de articulaciones de tipo rotatoria; la disposición de las articulaciones permiten un movimiento azimutal (giro en dirección perpendicular a la superficie en la que se encuentra apoyado el brazo) de todos los eslabones a través del primer grado de libertad (ubicado en la fundación), movimientos cenitales (que se encuentran en el plano perpendicular al de la superficie de apoyo) a través de los 3 siguientes y un giro perpendicular a estos últimos para el último eslabón a través de la última articulación. La determinación de la estructura cinemática a través del método de Denavit-Hartenberg implica la obtención de los parámetros geométricos que definen la estructura física del brazo y la capacidad de resolver la posición del extremo de este a través del conocimiento de los ángulos de cada grado de libertad, ambos aspectos utilizados como base en la construcción del algoritmo que controla los movimientos del brazo.

Con el uso de la librería de cinemática inversa IKPy, en conjunto con los resultados de aplicación del método de Denavit-Hartenberg, el uso de Arduino y la librería Serial de Python, se elaboró un sistema de módulos que conforman un algoritmo capaz originar el movimiento del brazo de dos formas: (1) a partir de los ángulos dados para cada grado de libertad, mover el brazo con el fin de llevarlo a una posición determinada (por ejemplo, una posición de reposo) y (2) conociendo las coordenadas que el extremo del brazo debe alcanzar, encontrar los ángulos de los grados de libertad que satisfagan una posición del brazo tal que su extremo quede en las coordenadas deseadas, en este caso, el punto de agarre del objeto.

Mediante la librería de OpenCV para Python, se desarrolló un algoritmo capaz de calcular automáticamente la posición del punto de agarre de un objeto esférico de

diámetro conocido mediante estereovisión. Este algoritmo logra encontrar el punto de agarre determinando el centro geométrico de la figura circular proyectada del objeto en el plano de imagen, la cual es obtenida (la figura) a través de la generación de una imagen binaria (con píxeles blancos y negros) originada desde la segmentación por colores de la imagen capturada, y haciendo uso de la geometría del modelo de la cámara estereopeica y del mapa de disparidad para determinar las coordenadas de la superficie del objeto accesible por el brazo. Como este algoritmo integra las funciones del algoritmo de movimiento, logran trabajar conjuntamente para generar el movimiento del brazo y la simulación de agarre del objeto.

Finalmente, se desarrolló un entramado de módulos, formados por aquellos que componen los algoritmos mencionados en los párrafos anteriores, que conforman un algoritmo de control capaz de producir el movimiento del brazo robótico para simular el agarre de un objeto esférico a partir de la determinación automática de su posición, a través de la captura y procesamiento de información por estereovisión.

Trabajos futuros

Este trabajo es solo un primer paso para llevar a cabo el objetivo final: una mano protésica inteligente, cuyas funciones de movimiento y agarre de objetos se basan en la toma de decisiones a partir de la información procesada a través de estereovisión.

Un siguiente paso en la dirección del objetivo final sería el diseño de un prototipo funcional de una mano (o garra) capaz de tomar objetos a través de la estereovisión, implicando tener que extrapolar los algoritmos de movimiento y detección de la posición de objetos a este nuevo trabajo. El nuevo algoritmo debería poder decidir cómo mover las partes de la mano (o garra) y cuándo se puede realizar el agarre del objeto de forma exitosa.

En cuanto a mejoras del algoritmo presentado en este escrito, sería interesante la implementación de una detección de posición de objetos más robusta, en el sentido de que el código sea capaz de detectar las dimensiones del objeto de forma automática y que además integre la solución del cálculo de la posición para una mayor variedad de cuerpos geométricos, brindándole así al usuario, un espectro de mayor tamaño en el que pueda desenvolverse con el uso de una futura prótesis de mano.

En adición a lo mencionado en el párrafo anterior, tendría un impacto muy positivo en el cálculo de la posición de objetos y en la experiencia del usuario, encontrar una solución que presente un mayor nivel de autonomía para detectar los objetos de interés, pues en la propuesta expuesta en este trabajo, se depende de una modificación manual por parte del usuario de los parámetros para segmentar la imagen y distinguir los objetos del resto de la escena. Una opción sería investigar si es posible una solución en que el algoritmo pueda encontrar las figuras de los objetos de interés evaluando, automáticamente, la diferencia de color entre los píxeles de los elementos de la escena, tal que cuando se encuentre un cambio que supere cierto umbral, se realice la segmentación de la imagen.

Bibliografía

- [1] REYES, Fernando. Robótica. Control de robots manipuladores. 1a. ed. México, Alfaomega Grupo Editor, 2011. pp. 212-213.
- [2] Fundamentos de robótica por Barrientos, Antonio [et al]. 2a ed. España, McGraw-Hill, 1997. pp. 96-99.
- [3] DAWSON-HOWE, Kenneth. A practical introduction to computer vision with OpenCV. 1 ed. Reino Unido, Wiley, 2014. pp. 9-10.
- [4] BRAHMBHATT, Samarth. Practical OpenCV. 1 ed. Estados Unidos, Apress, 2013. pp. 173-175.
- [5] BRAHMBHATT, Samarth. Practical OpenCV. 1 ed. Estados Unidos, Apress, 2013. pp. 179-181.
- [6] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. 641 p.
- [7] DAWSON-HOWE, Kenneth. A practical introduction to computer vision with OpenCV. 1 ed. Reino Unido, Wiley, 2014. pp. 81-82.
- [8] OPENCV. OpenCV 2.4.13.7 documentation, Camera calibration and 3D reconstruction [en línea]
<https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html> [consulta: 1 octubre 2019]
- [9] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. 647 p.

- [10] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, calibrateCamera [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#calibratecamera> [consulta: 1 octubre 2019]
- [11] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. pp. 721-723.
- [12] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, stereoCalibrate [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#stereocalibrate> [consulta: 2 octubre 2019]
- [13] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. 726 p.
- [14] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, stereoRectify [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#stereorectify> [consulta: 2 octubre 2019]
- [15] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. 708 p.
- [16] OPENCV. OpenCV 3.0.0-dev documentation, Geometric image transformations, initUndistortRectifyMap [en línea]
<https://docs.opencv.org/3.0-beta/modules/imgproc/doc/geometric_transformations.html#initundistortrectifymap> [consulta: 2 octubre 2019]
- [17] OPENCV. OpenCV 3.0.0-dev documentation, Geometric image transformations, remap [en línea]
<https://docs.opencv.org/3.0-beta/modules/imgproc/doc/geometric_transformations.html#remap> [consulta: 2 octubre 2019]
- [18] BRAHMBHATT, Samarth. Practical OpenCV. 1 ed. Estados Unidos, Apress, 2013. pp. 186-187.

- [19] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. 511 p.
- [20] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. pp. 700-703.
- [21] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, solvePnP [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#solvepnp> [consulta: 5 octubre 2019]
- [22] ROBOTIS. Dynamixel AX-12A, Specifications [en línea]
<<http://emanual.robotis.com/docs/en/dxl/ax/ax-12a/#specifications>> [consulta: 19 septiembre 2019]
- [23] MANCERON, Pierre. IKPy Wiki, Getting Started [en línea]
<<https://github.com/Phylliade/ikpy/wiki>> [consulta: 24 septiembre 2019]
- [24] MANCERON, Pierre. IKPy Wiki, Chain [en línea]
<<https://github.com/Phylliade/ikpy/wiki/Chain>> [consulta: 24 septiembre 2019]
- [25] ROS. Create your own URDF file [en línea]
<<http://wiki.ros.org/urdf/Tutorials/Create%20your%20own%20urdf%20file>> [consulta: 24 septiembre 2019]
- [26] MANCERON, Pierre. IKPy Wiki, Inverse Kinematics [en línea]
<<https://github.com/Phylliade/ikpy/wiki/Inverse-Kinematics>> [consulta: 24 septiembre 2019]
- [27] OPENCV. OpenCV Python Tutorials, Calibrate camera [en línea]
<https://docs.opencv.org/master/dc/dbb/tutorial_py_calibration.html> [consulta: 6 octubre 2019]
- [28] BRAHMBHATT, Samarth. Practical OpenCV. 1 ed. Estados Unidos, Apress, 2013. pp. 176-179.

- [29] NARASIMAMURTHY, Anirudh. ArUco markers for pose estimation, images [en línea]
<https://github.com/njanirudh/Aruco_Tracker/tree/master/images> [consulta: 6 octubre 2019]
- [30] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, findChessboardCorners [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findchessboardcorners> [consulta: 6 octubre 2019]
- [31] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, findFundamentalMat [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#findfundamentalmat> [consulta: 8 octubre 2019]
- [32] NARASIMAMURTHY, Anirudh. ArUco markers for pose estimation, aruco_tracker.py [en línea]
<https://github.com/njanirudh/Aruco_Tracker/blob/master/aruco_tracker.py> [consulta: 10 octubre 2019]
- [33] OPENCV. ArUco marker detection, detectMarkers() [en línea]
<https://docs.opencv.org/master/d9/d6a/group__aruco.html#gab9159aa69250d8d3642593e508cb6baa> [consulta: 10 octubre 2019]
- [34] OPENCV. ArUco marker detection, estimatePoseSingleMarkers() [en línea]
<https://docs.opencv.org/master/d9/d6a/group__aruco.html#ga84dd2e88f3e8c3255eb78e0f79571bd1> [consulta: 10 octubre 2019]
- [35] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, Rodrigues [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#rodrigues> [consulta: 10 octubre 2019]
- [36] KAEHLER, Adrian y BRADSKI, Gary. Learning OpenCV 3. 1 ed. Estados Unidos, O'Reilly, 2017. pp. 737-738.

- [37] HIRSCHMÜLLER, Heiko. Stereo processing by semiglobal matching and mutual information. IEEE Transactions on Pattern Analysis and Machine Intelligence, 30(2):328–341, Feb. 2008.
- [38] OPENCV. Camera calibration and 3D reconstruction, StereoSGBM class reference, Detailed description [en línea]
 <https://docs.opencv.org/3.4.1/d2/d85/classcv_1_1StereoSGBM.html#details>
 [consulta: 15 octubre 2019]
- [39] Fast global image smoothing based on weighted least squares por Dongbo Min... [et al]. IEEE Transactions on Image Processing, 23(12):5638–5653, Dic. 2014.
- [40] OPENCV. Extended image processing, Filters, DisparityWLSFilter class reference, Detailed description [en línea]
 <https://docs.opencv.org/3.4/d9/d51/classcv_1_1ximgproc_1_1DisparityWLSFilter.html#details> [consulta: 16 octubre 2019]
- [41] OPENCV. Tutorials for contrib modules, Disparity map post-filtering [en línea]
 <https://docs.opencv.org/3.1.0/d3/d14/tutorial_ximgproc_disparity_filtering.html> [consulta: 16 octubre 2019]
- [42] OPENCV. OpenCV 2.4.13.7 documentation, User interface, createTrackbar [en línea]
 <https://docs.opencv.org/2.4/modules/highgui/doc/user_interface.html#createtrackbar> [consulta: 17 octubre 2019]
- [43] OPENCV. OpenCV 2.4.13.7 documentation, User interface, getTrackbarPos [en línea]
 <https://docs.opencv.org/2.4/modules/highgui/doc/user_interface.html#gettrackbarpos> [consulta: 17 octubre 2019]
- [44] SAMARTZIDIS, Timotheos. OpenCV Stereo – Depth image generation and filtering with python 3+, ximgproc and OpenCV 3+ [en línea]
 <http://timosam.com/python_opencv_depthimage> [consulta: 17 octubre 2019]

- [45] OPENCV. OpenCV 3.0.0-dev documentation, Camera calibration and 3D reconstruction, createStereoSGBM [en línea]
<https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#createstereosgmb> [consulta: 17 octubre 2019]
- [46] OPENCV. Extended image processing, Filters, createDisparityWLSFilter() [en línea]
<https://docs.opencv.org/3.4/da/d17/group__ximgproc__filters.html#ga8a351f67b897bb7cdaccaef115bafcac> [consulta: 17 octubre 2019]
- [47] OPENCV. OpenCV 3.0.0-dev documentation, Operation on arrays, inRange [en línea]
<https://docs.opencv.org/3.0-beta/modules/core/doc/operations_on_arrays.html#inrange> [consulta: 18 octubre 2019]
- [48] OPENCV. OpenCV 3.0.0-dev documentation, Structural Analysis and Shape Descriptors, findContours [en línea]
<https://docs.opencv.org/3.0-beta/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#findcontours> [consulta: 18 octubre 2019]
- [49] OPENCV. OpenCV 3.0.0-dev documentation, Structural Analysis and Shape Descriptors, moments [en línea]
<https://docs.opencv.org/3.0-beta/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html#moments> [consulta: 18 octubre 2019]

Anexo A

Códigos del algoritmo de control de movimiento del brazo robótico

A.1. Módulo de Python con funciones de parámetros de Denavit-Hartenberg y de servomotores

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4
5
6 def par_dh(thetai): #Parámetros de Denavit-Hartenberg de YARBIZ.
7
8     #Distancias di en [mm].
9     d1 = 62 #Apoyo a eje servo 2.
10    d2 = 0
11    d3 = 0
12    d4 = 0
13    d5 = -30 #Corrección de sistema de referencia extremo.
14
15    #Distancias ai en [mm].
16    a1 = 0
17    a2 = 70 #Eje servo 2 a eje servo 3.
18    a3 = 70 #Eje servo 3 a eje servo 4.
19    a4 = 61 #Eje servo 4 a eje servo 5.
20    a5 = 43+5 #Eje servo 5 a superficie eslabón extremo.
21
22    #Ángulos alpha en grados.
23    alpha1 = 90.0
24    alpha2 = 0
25    alpha3 = 0
26    alpha4 = -90.0
```

```

27     alpha5 = 0
28
29     #Ángulos alpha en radianes.
30     alpha1 = alpha1/180*np.pi
31     alpha2 = alpha2/180*np.pi
32     alpha3 = alpha3/180*np.pi
33     alpha4 = alpha4/180*np.pi
34     alpha5 = alpha5/180*np.pi
35
36     #Ángulos theta en radianes.
37     theta1 = float(thetai[0])/180*np.pi
38     theta2 = float(thetai[1])/180*np.pi
39     theta3 = float(thetai[2])/180*np.pi
40     theta4 = float(thetai[3])/180*np.pi
41     theta5 = float(thetai[4])/180*np.pi
42
43     dh = [[theta1,d1,a1,alpha1],
44           [theta2,d2,a2,alpha2],
45           [theta3,d3,a3,alpha3],
46           [theta4,d4,a4,alpha4],
47           [theta5,d5,a5,alpha5]]
48
49
50     #Matrices de transformación.
51     i = 1
52     A01 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0]),
53            ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i-1][0])],
54            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0]),
55            ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
56            sin(dh[i-1][0])],
57            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
58            [0,0,0,1]]
59
60     i = 2
61     A12 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0]),
62            ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i-1][0])],
63            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0]),
64            ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
65            sin(dh[i-1][0])],
66            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
67            [0,0,0,1]]
68
69     i = 3
70     A23 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0]),
71            ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i-1][0])],
72            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0]),
73            ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
74            sin(dh[i-1][0])],
75            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
76            [0,0,0,1]]

```

```

67         [0,0,0,1]]
68
69     i = 4
70     A34 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
71             ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i-1][0])],
72            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
73             ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
74             sin(dh[i-1][0])],
75            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
76            [0,0,0,1]]
77
78     i = 5
79     A45 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
80             ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i-1][0])],
81            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
82             ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
83             sin(dh[i-1][0])],
84            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
85            [0,0,0,1]]
86
87     A02 = np.dot(A01, A12)
88     A03 = np.dot(A02, A23)
89     A04 = np.dot(A03, A34)
90     A05 = np.dot(A04, A45) #Matriz de transformación final.
91
92     #Coordenadas cartesianas del extremo de YARBIZ.
93     x = A05[0,3]
94     y = A05[1,3]
95     z = A05[2,3]
96     p = np.array([x,
97                  y,
98                  z])
99
100    return dh, p
101
102    def par_fu(): #Parámetros de las funciones ui(thetai) (pfu).
103
104        k = 1/0.29 #Constante de conversión servos en unidades/grado.
105
106        u1_0 = 512 #Unidades u en la que thetai es cero.
107        u1_min = 0 #Unidades u mín.
108        u1_max = 1023 #Unidades u máx.
109        thetai_min = (u1_min - u1_0)/k
110        thetai_min = round(thetai_min,2) #Ángulo theta mín.
111        thetai_max = (u1_max - u1_0)/k
112        thetai_max = round(thetai_max,2) #Ángulo theta máx.
113
114        u2_0 = 202
115        u2_min = 133

```

```

111     u2_max = 885
112     theta2_min = (u2_min - u2_0)/k
113     theta2_min = round(theta2_min,2)
114     theta2_max = (u2_max - u2_0)/k
115     theta2_max = round(theta2_max,2)
116
117     u3_0 = 512
118     u3_min = 0
119     u3_max = 1023
120     theta3_max = (u3_max - u3_0)/k
121     theta3_max = round(theta3_max,2)
122     theta3_min = (u3_min - u3_0)/k
123     theta3_min = round(theta3_min,2)
124
125     u4_0 = 512
126     u4_min = 167
127     u4_max = 857
128     theta4_min = (u4_min - u4_0)/k
129     theta4_min = round(theta4_min,2)
130     theta4_max = (u4_max - u4_0)/k
131     theta4_max = round(theta4_max,2)
132
133     u5_0 = 512
134     u5_min = 167
135     u5_max = 857
136     theta5_min = (u5_min - u5_0)/k
137     theta5_min = round(theta5_min,2)
138     theta5_max = (u5_max - u5_0)/k
139     theta5_max = round(theta5_max,2)
140
141     pfu = [[u1_0,u1_min,u1_max,theta1_min,theta1_max], #Arreglo de
142             los pfu.
143             [u2_0,u2_min,u2_max,theta2_min,theta2_max],
144             [u3_0,u3_min,u3_max,theta3_min,theta3_max],
145             [u4_0,u4_min,u4_max,theta4_min,theta4_max],
146             [u5_0,u5_min,u5_max,theta5_min,theta5_max],
147             [k]]
148     return pfu

```

A.2. Módulo de Python con funciones de cinemática inversa y cinemática directa para generación del movimiento

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4 import serial
5 import ikpy
6 from ikpy.chain import Chain
7 from ikpy.link import OriginLink, URDFLink
8 from yarbiz_data import par_dh
9 from yarbiz_data import par_fu
10
11
12 def ik(p_obj, p_ini, thetai_ini, segm, tol, com, baud): #Función de
    Cinemática Inversa.
13
14     print('\n')
15
16     dh,_ = par_dh(thetai_ini)
17
18     pfu = par_fu()
19
20     #Creación del brazo - librería ikpy.
21     yarbiz = Chain(name='YARBIZ', links=[
22         OriginLink(),
23         URDFLink(
24             name="Servo1",
25             translation_vector=[0,0,0],
26             orientation=[0,0,0],
27             rotation=[0,0,1],
28         ),
29         URDFLink(
30             name="Servo2",
31             translation_vector=[dh[0][2],0,dh[0][1]],
32             orientation=[dh[0][3],0,0],
33             rotation=[0,0,1],
34         ),
35         URDFLink(
36             name="Servo3",
37             translation_vector=[dh[1][2],0,dh[1][1]],
38             orientation=[dh[1][3],0,0],
39             rotation=[0,0,1],
40         ),
41         URDFLink(
42             name="Servo4",
43             translation_vector=[dh[2][2],0,dh[2][1]],
```

```

44         orientation=[dh[2][3],0,0],
45         rotation=[0,0,1],
46     ),
47     URDFLink(
48         name="Servo5",
49         translation_vector=[dh[3][2],0,dh[3][1]],
50         orientation=[dh[3][3],0,0],
51         rotation=[0,0,1],
52     ),
53     URDFLink(
54         name="Extremo",
55         translation_vector=[dh[4][2],0,dh[4][1]],
56         orientation=[dh[4][3],0,0],
57         rotation=[0,0,0],
58     )
59 ] )
60
61
62 v = p_obj - p_ini #Vector diferencia p del objeto con p inicial
63                   de yarbiz.
64
65 dist = np.sqrt([v[0,0]**2 + v[1,0]**2 + v[2,0]**2])
66 dist = dist[0] #Magnitud vector p_obj - p_ini.
67
68 d = (1/dist)*v #Vector director unitario.
69
70 #Cantidad de segmentos.
71 if segm == 0:
72     n = 1
73
74 else:
75
76     n = np.ceil([dist/segm])
77     n = int(n[0])
78
79
80 success = False #Variable que indica el éxito del movimiento.
81
82 thetai = thetai_ini[:] #Arreglo con ángulos actuales.
83
84 instrs = [] #Arreglo para instrucciones de todos los movimientos
85             para los servos.
86
87 for j in range(1,n+1): #Tantos movimientos como segmentos
88                         calculados.
89
90     if j == n:
91
92         p_j = p_obj
93
94     else:

```

```

93         t = segm*j #Distancia total entre p_ini y el nuevo p_j.
94
95         p_j = p_ini + t*d #Puntos intermedios.
96
97
98     print('p_' + str(j) + ' = (' + str(round(p_j[0,0],1)) + ', '
99           + str(round(p_j[1,0],1)) + ', ' + str(round(p_j[2,0],1))
100          + ') [mm]')
101
102     #Creación de la matriz objetivo j ([orientación_j | punto_j])
103     .
104     frame_j = np.eye(4)
105     frame_j[:-1,3] = np.transpose(p_j)
106
107     #Arreglo con ángulos actuales en formato [0, theta1, theta2,
108     ... , theta5, 0] en [rad].
109     thetai_act_ik = []
110
111     for i in range(1,8):
112
113         if i == 1 or i == 7:
114
115             thetai_act_ik.append(0)
116
117         else:
118
119             thetai_act_ik.append(round(thetai[i-2]*np.pi/180, 7))
120
121     #Solución ikpy.
122     thetai_ik = yarbiz.inverse_kinematics(frame_j, thetai_act_ik)
123
124     instr = '' #String con instrucciones para servos para
125     movimiento j.
126
127     for i in range(1,6):
128
129         #Ángulos theta en grados.
130
131         theta = thetai_ik[i]*180/np.pi #Ángulo solución en grados
132         para el servo i en el movimiento j.
133
134         if theta >= pfu[i-1][4] or theta <= pfu[i-1][3]: #
135             Corrección de ángulos fuera de rango.
136
137             theta = theta - round(theta/360,0)*360
138
139         #Generación de la instrucción para servos.

```

```

138
139     u = pfu[i-1][0] + theta*pfu[5][0] #Calcula unidades servo
140     u a partir de thetai.
141     u = int(round(u,0)) #Aproximar a unidad más cercana.
142
143     if u < pfu[i-1][1]: #Si u está bajo el mínimo...
144
145         u = pfu[i-1][1]
146
147     elif u > pfu[i-1][2]: #Si u está sobre el máximo...
148
149         u = pfu[i-1][2]
150
151
152     if i == 5:
153
154         instr = instr + str(u)
155
156         instrs.append(instr)
157
158     else:
159
160         instr = instr + str(u) + ','
161
162
163     #Actualizar posición actual para el siguiente movimiento.
164
165     theta = (u - pfu[i-1][0])/pfu[5][0] #theta para el servo
166     i con precisión de los servos (0.29[°/u]).
167     thetai[i-1] = round(theta,2)
168
169
170     print('Instrucción = ' + instr)
171     print('\n')
172
173
174     if j == n: #Si es el último movimiento, calcular p.
175
176         _,p = par_dh(thetai) #Punto alcanzado con thetai.
177
178
179         diff = p - p_obj #Vector diferencia entre resultado y
180         objetivo.
181         delta = np.sqrt([diff[0,0]**2 + diff[1,0]**2 + diff
182             [2,0]**2])
183         delta = delta[0] #Módulo del vector diferencia.
184
185         if delta <= tol:
186
187             success = True

```

```

186         print('delta = ' + str(round(delta,1)) + ' [mm]')
187         print('')
188
189     else:
190
191         thetai = thetai_ini
192         p = p_ini
193
194         print('(Info) El objetivo está fuera de alcance!')
195         print('\n')
196
197
198     if success:
199
200         try:
201
202             ser = serial.Serial(com,baud)
203
204             for i in range(1,n+1):
205
206                 instr = instrs[i-1]
207
208                 ser.write(instr.encode())
209
210                 ser.read(1)
211
212
213             except:
214
215                 print('')
216                 print('(Error) Ha habido problemas para comunicarse con
217                     el puerto serial ' + com + '!')
218                 print('\n')
219
220         print('')
221         print('Theta_i = ' + str(thetai) + ' [°]')
222         print('p = (' + str(round(p[0,0],1)) + ', ' + str(round(p[1,0],1))
223             + ', ' + str(round(p[2,0],1)) + ') [mm]')
224         print('Success: ', success)
225         print('\n')
226         print('\n')
227
228     return success, thetai, p
229
230
231
232 def fk(thetai_obj, com, baud): #Cinemática Directa.
233
234     print('\n')
235

```

```

236 pfu = par_fu()
237
238 range_error = False
239
240 thetai = [] #Arreglo para ángulos según precisión de servos
           (0.29[°/u])
241
242 #Generación de la instrucción para servos.
243 instr = ''
244
245 for i in range(1,6):
246
247     u = pfu[i-1][0] + float(thetai_obj[i-1])*pfu[5][0] #Calcula
           unidades servo u a partir de thetai.
248     u = int(round(u,0)) #Aproximar a unidad más cercana.
249
250     theta = (u - pfu[i-1][0])/pfu[5][0] #theta para el servo i
           con precisión de los servos (0.29°/u).
251     thetai.append(round(theta,2))
252
253
254     if u >= pfu[i-1][1] and u <= pfu[i-1][2]:
255
256         if i == 5:
257
258             instr = instr + str(u)
259
260         else:
261
262             instr = instr + str(u) + ', '
263
264     else:
265
266         range_error = True
267
268         break
269
270
271
272 if range_error == False:
273
274     print('Instrucción = ' + instr)
275     print('')
276
277
278     _,p = par_dh(thetai) #Punto alcanzado con thetai.
279
280
281     try:
282
283         ser = serial.Serial(com,baud)
284

```

```

285         ser.write(instr.encode())
286
287         ser.read(1)
288
289     except:
290
291         print('')
292         print('(Error) Ha habido problemas para comunicarse con
           el puerto serial ' + com + '!')
293         print('\n')
294
295
296         print('Theta_i = ' + str(thetai) + ' [°]')
297         print('p = (' + str(round(p[0,0],1)) + ', ' + str(round(p
           [1,0],1)) + ', ' + str(round(p[2,0],1)) + ') [mm]')
298         print('\n')
299         print('\n')
300
301     else:
302
303         print('')
304         print('(Error) Theta' + str(i) + ' FUERA DE RANGO! (' + str(
           round(theta,2)) + ' [°]), Theta'+str(i) + '[MÍN,MÁX]: ['
           + str(pfu[i-1][3]) + ', ' + str(pfu[i-1][4]) + '] [°]!')
305         print('\n')
306
307         thetai = []
308         p = []
309
310
311     return thetai, p

```

A.3. Script de Arduino para control del movimiento de los servomotores

```
1 #include <DynamixelWorkbench.h>
2
3 #if defined(__OPENCM904__)
4     #define DEVICE_NAME "1" //Dynamixel on Serial3(USART3)  <-OpenCM
5         485EXP
6 #elif defined(__OPENCR__)
7     #define DEVICE_NAME ""
8 #endif
9
10 #define BAUDRATE 1000000
11
12 //Se declaran las variables.
13 String instr;
14 String us;
15 int len;
16 int32_t u;
17
18 uint8_t dxl_1 = 1;
19 uint8_t dxl_2 = 2;
20 uint8_t dxl_3 = 3;
21 uint8_t dxl_4 = 4;
22 uint8_t dxl_5 = 5;
23
24 uint8_t dxl_id;
25
26 int32_t dxl_vel = 40;
27
28 DynamixelWorkbench dxl_wb;
29
30 void setup() {
31
32     Serial.begin(115200); //Incializar puerto serial.
33
34     const char *log;
35     bool result = false;
36
37     //Se inicializan los servomotores.
38     dxl_wb.init(DEVICE_NAME, BAUDRATE, &log);
39
40     dxl_wb.ping(dxl_1, &log);
41     dxl_wb.ping(dxl_2, &log);
42     dxl_wb.ping(dxl_3, &log);
43     dxl_wb.ping(dxl_4, &log);
44     dxl_wb.ping(dxl_5, &log);
45 }
```

```

46 //Configurar servos en modo Joint y asignar velocidad.
47 dxl_wb.jointMode(dxl_1, dxl_vel, 0, &log);
48 dxl_wb.jointMode(dxl_2, dxl_vel, 0, &log);
49 dxl_wb.jointMode(dxl_3, dxl_vel, 0, &log);
50 dxl_wb.jointMode(dxl_4, dxl_vel, 0, &log);
51 dxl_wb.jointMode(dxl_5, dxl_vel, 0, &log);
52
53
54 // int32_t get_data;
55 //
56 // dxl_wb.getPresentPositionData(dxl_3, &get_data, &log);
57 //
58 // Serial.print("Succeed to get present position(value : ");
59 // Serial.println(get_data);
60
61
62 }
63
64 void loop() {
65
66   if(Serial.available()>=1){ //Si hay información en el puerto serial
67     ...
68     instr = Serial.readString(); //Leer instrucción proveniente del m
        ódulo de cinemática.
69
70     for (int i=1; i<=5; i++) {
71
72       dxl_id = i;
73
74       us = instr.substring(0,instr.indexOf(","));
75
76       len = us.length();
77
78       instr.remove(0,len+1); //Eliminar u_i de la instrucción.
79
80       u = us.toInt(); //Obtener unidades de posición.
81       dxl_wb.goalPosition(dxl_id, u); //Ejecutar movimiento.
82
83       if(i < 5){
84         delay(0);
85       }
86       else{
87         delay(0);
88       }
89     }
90
91     Serial.write(1); //Enviar información sobre el término de la
        operación.
92
93   }
94 }

```

Anexo B

Códigos de algoritmo de detección de posición de objetos

B.1. Módulo de Python con código para calibración de parámetros intrínsecos de las cámaras

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4 import cv2 as cv
5 import glob
6
7
8 def generate_vimg_name(cam_name, i): #Función para generar el nombre
   (str) de imágenes válidas (patrón encontrado).
9
10     if i >= 100:
11
12         vimg_name = cam_name+'_0'+str(i)+'.jpg'
13
14     elif i >= 10:
15
16         vimg_name = cam_name+'_00'+str(i)+'.jpg'
17
18     else:
19
20         vimg_name = cam_name+'_000'+str(i)+'.jpg'
21
22     return vimg_name
23
24
25 #Menú de opciones para calibración.
```

```

26 print('')
27
28
29 #Información de la cámara a calibrar.
30 print('Cámara a calibrar:')
31 print('')
32
33 flag = True
34
35 while flag:
36
37     print('    [1] Cámara izquierda')
38     print('    [2] Test')
39     print('')
40
41     o = input('    >> Opción: ')
42
43     print('\n')
44
45     if o == '1':
46
47         cam_name = 'left'
48         rel_dir = 'mono/'
49         win_name = 'Left img.: '
50         flag = False
51
52     elif o == '2':
53
54         cam_name = 'test'
55         rel_dir = 'test/'
56         win_name = 'Test img.: '
57         flag = False
58
59     else:
60
61         print('    (Error) Opción inválida!')
62         print('\n')
63
64
65 #Determinar número de dispositivo de la cámara a calibrar.
66 print('Determinación de número de dispositivo de cámara:')
67 print('')
68
69 flag0 = True
70 flag1 = False
71
72 while flag0:
73
74     print('    Números de dispositivo:')
75     print('')
76     cam_num = int(input('    >> Número de dispositivo: '))
77     print('\n')

```

```

78
79     print('      (Info) Intentando inicializar cámara...')
80
81     cam = cv.VideoCapture(cam_num)
82
83     ret, img = cam.read() #Intentar leer frame de la cámara
84
85     if ret:
86
87         print('')
88         print('      (Info) Cámara inicializada!')
89         print('\n')
90
91         print('      Es correcto el número de dispositivo?')
92         print('')
93         print('      >> Durante ventana de captura activa,
94             presionar [key]:')
95         print('')
96         print('      [y] Sí')
97         print('      [n] No')
98         print('\n')
99         flag1 = True
100
101     else:
102
103         print('\n')
104         print('      (Error) No se ha podido inicializar la cámara!')
105         print('      Ingrese nuevamente el número de
106             dispositivo!')
107         print('\n')
108
109     while flag1:
110
111         ret, img = cam.read() #Lee frame.
112
113         if ret:
114
115             cv.imshow(win_name + 'BGR', img)
116
117             key = cv.waitKey(33)
118
119             if key == ord('y'):
120
121                 flag1 = False
122                 flag0 = False
123
124                 cam.release()
125                 cam.release()
126                 cv.destroyAllWindows()
127
128             elif key == ord('n'):

```

```

128
129         flag1 = False
130
131         cam.release()
132         cam.release()
133         cv.destroyAllWindows()
134
135
136
137 #Información del patrón de calibración.
138
139 print('Datos sobre el patrón de calibración:')
140 print('')
141
142 size = float(input('    >> Tamaño de lado de los cuadros en [mm] = '))
143     )
144 print('')
145
146 nx = int(input('    >> Nro. de intersecciones horizontales = '))
147 ny = int(input('    >> Nro. de intersecciones verticales    = '))
148
149 print('\n')
150
151 #Selección de fuente de datos para la calibración.
152 print('Modalidad de calibración:')
153 print('')
154
155 flag = True
156
157 while flag:
158
159     print('    [1] Calibración desde imagenes en tiempo real')
160     print('    [2] Calibración desde imágenes pre-tomadas')
161     print('')
162
163     o = input('    >> Opción: ')
164
165     print('\n')
166
167     if o == '1' or o == '2':
168
169         flag = False
170
171     else:
172
173         print('    (Error) Opción inválida!')
174         print('\n')
175
176
177 cont = 0 #Nro. de iteraciones.
178 i = 0 # Nro. de imágenes válidas.

```

```

179
180 criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
    0.001) #Criterio de terminación.
181
182 #Preparar puntos de objeto en formato (0,0,0), (1*size,0,0), (2*size
    ,0,0), ..., ((ny-1)*size, (nx-1)*size,0).
183 ptnpts = np.zeros((nx*ny,3), np.float32)
184 ptnpts[:,2] = np.mgrid[0:ny*size:size, 0:nx*size:size].T.reshape
    (-1,2)
185
186 #Arreglos para almacenar puntos de objeto y puntos de imagen desde
    todas las imágenes válidas.
187 objpts = [] #Puntos 3D del espacio real.
188 imgpts = [] #Puntos 2D en el plano de la imagen.
189
190
191 if o == '1': #Recolección de imágenes válidas en tiempo real y de
    información necesaria para calibración.
192
193     n = int(input('>> Número de imágenes válidas requeridas = ')) #
        Cantidad de imágenes válidas requeridas.
194     print('\n')
195
196     print('>> Procesar imágenes:')
197     print('')
198     print('    Durante ventana de captura activa, presionar [key]:')
199     print('')
200     print('    [1] Procesar imagen')
201     print('\n')
202
203     print('(Info) Inicializando cámara...')
204     print('')
205
206     cam = cv.VideoCapture(cam_num) #Inicia la captura de video.
207
208     print('(Info) Cámara inicializada!')
209     print('\n')
210
211
212     print('(Info) Válidas = '+str(i))
213
214     while i < n: #Mientras la cantidad de imágenes válidas sea menor
        a lo requerido...
215
216         ret, img = cam.read() #Lee un frame de la cámara.
217
218         if ret: #Si la captura fue exitosa...
219
220             gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) #Imagen a
                escala de grises.
221
222             cv.imshow(win_name + 'Gray', gray)

```

```

223         k = cv.waitKey(33)
224
225         if k == ord('1'):
226
227             ret, corners = cv.findChessboardCorners(gray, (ny,nx)
228             , None) #Encontrar las esquinas del tablero de
229             ajedrez.
230
231             if ret == True: #Si encuentra las esquinas...
232
233                 i += 1 #Incrementa el número de imágenes válidas.
234                 print('(Info) Válidas = '+str(i))
235
236                 objpts.append(ptnpts) #Agrega puntos de objeto (3
237                 D).
238
239                 corners2 = cv.cornerSubPix(gray, corners, (11,11)
240                 , (-1,-1), criteria) #Refina esquinas.
241                 imgpts.append(corners2) #Agrega puntos refinados
242                 de imagen (2D, pixeles)
243
244                 #Dibujar y mostrar esquinas del tablero.
245                 gray_bgr = cv.cvtColor(gray, cv.COLOR_GRAY2BGR)
246                 cv.drawChessboardCorners(gray_bgr, (ny,nx),
247                 corners2, ret)
248                 cv.imshow(cam_name+'_v-img', gray_bgr)
249                 cv.waitKey(33)
250
251                 #Guardar imagen válida.
252                 vimg_name = generate_vimg_name(cam_name, i)
253                 cv.imwrite(rel_dir + vimg_name, gray)
254
255
256 elif o == '2': #Validación y recolección de información necesaria
257     para fase de calibración desde imágenes pre-tomadas.
258
259     print('Procesar imágenes:')
260     print('\n')
261
262     images = glob.glob(rel_dir+'*.jpg')
263
264     for fname in images:
265
266         cont += 1
267
268         print('')

```

```

268         print('      It = '+str(cont))
269
270         img = cv.imread(fname)
271
272         gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)
273
274         cv.imshow('img', gray)
275         cv.waitKey(100)
276
277         ret, corners = cv.findChessboardCorners(gray, (ny,nx), None)
278             #Encontrar las esquinas del tablero de ajedrez.
279
280         if ret == True: #Si encuentra las esquinas...
281
282             i += 1
283             print(' (Info) Válidas = '+str(i))
284
285             objpts.append(ptnpts) #Agrega los puntos de objeto (3D)
286
287             corners2 = cv.cornerSubPix(gray, corners, (11,11),
288                 (-1,-1), criteria) #Refina las esquinas.
289             imgpts.append(corners2) #Agrega los puntos de imagen (2D)
290
291             #Dibujar y mostrar esquinas del tablero.
292             gray_bgr = cv.cvtColor(gray, cv.COLOR_GRAY2BGR)
293             cv.drawChessboardCorners(gray_bgr, (ny,nx), corners2, ret
294                 )
295             cv.imshow(cam_name+'_v-img', gray_bgr)
296             cv.waitKey(100)
297
298             #Guardar imagen válida.
299             vimg_name = generate_vimg_name(cam_name, i)
300             cv.imwrite(rel_dir + vimg_name, gray)
301
302         else:
303
304             print(' (Info) Válidas = '+str(i))
305
306         cv.destroyAllWindows()
307         print('\n')
308
309         #Obtener parámetros intrínsecos.
310         reproy_err_rms, imtx, distc, rvecs, tvecs = cv.calibrateCamera(objpts
311             , imgpts, gray.shape[::-1], None, None)
312
313         if reproy_err_rms <= 0.5: #Valor límite del error RMS de re-proyección
314             n para una buena calibracion.
315
316             print(' (Info) Calibración exitosa! (Error RMS de re-proyección [

```

```

315     px] = '+str(reproy_err_rms)+' E[0, 0.5])')
316     print('\n')
317     np.savez(rel_dir + 'mono_intr', imtx=imtx, distc=distc)
318
319     print('(Info) Archivo .npz con matriz de parámetros intrínsecos y
320           coeficientes de distorsión generado!')
321     print('\n')
322     npzfile = np.load(rel_dir + 'mono_intr.npz')
323
324     print('imtx = ')
325     print(npzfile['imtx'])
326     print('')
327
328     print('distc = ')
329     print(npzfile['distc'])
330     print('\n')
331
332
333     print('Probar calibración:')
334     print('')
335
336     flag = True
337
338     while flag:
339
340         print('      [1] Probar calibración')
341         print('      [0] Terminar')
342         print('')
343
344         o = input('      >> Opción: ')
345         print('\n')
346
347         if o == '1':
348
349             imgfile = input('      >> Nombre de imagen a corregir: ')
350             print('')
351
352             img = cv.imread(rel_dir + imgfile + '.jpg')
353
354             #Eliminar distorsión.
355             h, w = img.shape[:2]
356             newcameramt, roi = cv.getOptimalNewCameraMatrix(imtx,
357                                                             distc, (w,h), 1, (w,h))
358             undst_img = cv.undistort(img, imtx, distc, None,
359                                     newcameramt)
360             x, y, w, h = roi
361             undst_img = undst_img[y:y+h, x:x+w]
362
363             cv.imwrite(rel_dir + imgfile + '_calibtest.png',
364                       undst_img)

```

```

362
363         print('      (Info) Imagen de prueba generada!')
364         print('\n')
365
366
367         elif o == '0':
368
369             flag = False
370
371         else:
372
373             print('      (Error) Opción inválida!')
374             print('\n')
375
376
377     else:
378
379         print('(Info) Calibración fallida! (Error RMS de re-proyección [
          px] = ' + str(reproy_err_rms) + ' E/[0, 0.5]))')

```

B.2. Módulo de Python con código para calibración de parámetros estéreo de las cámaras

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4 import cv2 as cv
5 import os
6
7
8 def generate_vimg_name(cam_name, i): #Función para generar el nombre
   (str) de imágenes válidas (patrón encontrado).
9
10     if i >= 100:
11
12         vimg_name = cam_name+'_0'+str(i)+'.jpg'
13
14     elif i >= 10:
15
16         vimg_name = cam_name+'_00'+str(i)+'.jpg'
17
18     else:
19
20         vimg_name = cam_name+'_000'+str(i)+'.jpg'
21
22     return vimg_name
23
24
25
26 #Menú para calibración.
27 print('')
28
29 print('Determinación de números de dispositivo de cámaras:')
30 print('')
31
32 flag0 = True
33 flag1 = False
34
35 while flag0:
36
37     print('    Números de dispositivo:')
38     print('')
39     l_cam_num = int(input('                >> Cámara izq.: '))
40     r_cam_num = int(input('                >> Cámara der.: '))
41     print('\n')
42
43     print('    (Info) Intentando inicializar cámaras...')
44
45     l_cam = cv.VideoCapture(l_cam_num)
```

```

46     r_cam = cv.VideoCapture(r_cam_num)
47
48     l_ret, l_img = l_cam.read() #Lee un frame de la cámara izquierda.
49     r_ret, r_img = r_cam.read() #Lee un frame de la cámara derecha.
50
51     if l_ret or r_ret:
52
53         print('')
54         print('      (Info) Cámara(s) inicializada(s)!')
55         print('\n')
56
57         print('      Son correctos los números de dispositivo?')
58         print('')
59         print('      >> Durante cualquier ventana de captura activa
60             , presionar [key]:')
61         print('')
62         print('      [y] Sí')
63         print('      [n] No')
64         print('\n')
65         flag1 = True
66
67     else:
68
69         print('\n')
70         print('      (Error) No se han podido inicializar las cámaras!'
71             )
72         print('      Ingrese nuevamente los números de
73             dispositivo!')
74         print('\n')
75
76     while flag1:
77
78         l_ret, l_img = l_cam.read() #Lee un frame de la cámara
79             izquierda.
80         r_ret, r_img = r_cam.read() #Lee un frame de la cámara
81             derecha.
82
83         if l_ret:
84
85             cv.imshow('left_img', l_img)
86
87         if r_ret:
88
89             cv.imshow('right_img', r_img)
90
91         key = cv.waitKey(33)
92
93         if key == ord('y'):
94
95             flag1 = False
96             flag0 = False

```

```

93         l_cam.release()
94         r_cam.release()
95         cv.destroyAllWindows()
96
97     elif key == ord('n'):
98
99         flag1 = False
100
101         l_cam.release()
102         r_cam.release()
103         cv.destroyAllWindows()
104
105
106
107 print('Seleccione una opción:')
108 print('')
109
110 flag = True
111
112 while flag:
113
114     print('    [1] Continuar con calibración')
115     print('    [2] Solo modificar número de dispositivos')
116     print('')
117
118     o = input('    >> Opción: ')
119
120     print('\n')
121
122     if o == '1':
123
124         flag = False
125
126     elif o == '2':
127
128         try:
129
130             rel_dir = 'stereo/'
131             file_name = 'stereo_intr.npz'
132             path = rel_dir + file_name
133
134             npzfile = np.load(path)
135
136             l_map1 = npzfile['l_map1']
137             l_map2 = npzfile['l_map2']
138
139             r_map1 = npzfile['r_map1']
140             r_map2 = npzfile['r_map2']
141
142             l_P = npzfile['l_P']
143
144             B = npzfile['B']

```

```

145
146         #Guardar archivo .npz con los parámetros estéreo.
147         np.savez(rel_dir + file_name, l_map1=l_map1, l_map2=
148             l_map2, r_map1=r_map1, r_map2=r_map2,
149                 l_P=l_P, B=B,
150                 l_cam_num=l_cam_num,
151                 r_cam_num=r_cam_num)
152
153         print('(Info) Números de dispositivo actualizados
154             exitosamente!')
155         print('\n')
156
157         quit()
158
159     except:
160
161         print('(Error) No se ha encontrado archivo npz. con pará
162             matros intr. estéreo!')
163         print('        Se procederá a la sección de calibración
164             estéreo.')
165         print('\n')
166
167     else:
168
169         print('        (Error) Opción inválida!')
170         print('\n')
171
172 #Selección de la calibración.
173 print('Modalidad de calibración:')
174 print('')
175
176 flag = True
177
178 while flag:
179
180     print('        [1] Calibración estéreo')
181     print('        [2] Test')
182     print('')
183
184     o = input('        >> Opción: ')
185
186     print('\n')
187
188     if o == '1':
189
190         name = ''
191         rel_dir = 'stereo/'
192         flag = False
193
194     elif o == '2':

```

```

192     name = 'test_'
193     rel_dir = 'test/'
194     flag = False
195
196     else:
197
198         print('      (Error) Opción inválida!')
199         print('\n')
200
201
202 #Información del patrón de calibración.
203 print('Datos sobre el patrón de calibración:')
204 print('')
205 size = float(input('      >> Tamaño de lado de los cuadros en [mm] = '))
206     )
207 print('')
208 nx = int(input('      >> Nro. de intersecciones horizontales = '))
209 ny = int(input('      >> Nro. de intersecciones verticales   = '))
210 print('\n')
211
212
213 n = int(input('>> Número de imágenes válidas requeridas = ')) #
214     Cantidad de imágenes válidas requeridas.
215 print('\n')
216
217 i = 0 # Nro. de imágenes válidas.
218
219 criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
220     0.001) #Criterio de terminación.
221
222 #Preparar puntos de objeto en formato (0,0,0), (1*size,0,0), (2*size
223     ,0,0), ..., ((ny-1)*size, (nx-1)*size, 0) .
224 ptnpts = np.zeros((nx*ny,3), np.float32)
225 ptnpts[:, :2] = np.mgrid[0:ny*size:size, 0:nx*size:size].T.reshape
226     (-1,2)
227
228 #Arreglos para almacenar puntos de objeto y puntos de imagen desde
229     todas las imágenes válidas.
230 objpts = [] #Puntos 3D del espacio real.
231 l_imgpts = [] #Puntos 2D en el plano de la imagen de la cámara
232     izquierda.
233 r_imgpts = [] #Puntos 2D en el plano de la imagen de la cámara
234     derecha.
235
236
237 #Recolección de imágenes válidas en tiempo real y de información
238     necesaria para calibración.
239
240
241 print('>> Procesar imágenes:')
242 print('')

```

```

235 print('    Durante cualquier ventana de captura activa, presionar [
      key]:')
236 print('')
237 print('    [1] Procesar imágenes')
238 print('\n')
239
240 print('(Info) Inicializando cámaras...')
241 print('')
242
243 #Inicia la captura de video.
244 l_cam = cv.VideoCapture(l_cam_num)
245 r_cam = cv.VideoCapture(r_cam_num)
246
247 print('(Info) Cámaras inicializadas!')
248 print('\n')
249
250 print('(Info) Válidas = '+str(i))
251
252 while i < n: #Mientras la cantidad de imágenes válidas sea menor a lo
      requerido...
253
254     l_ret, l_img = l_cam.read() #Lee un frame de la cámara izquierda.
255     r_ret, r_img = r_cam.read() #Lee un frame de la cámara derecha.
256
257     if l_ret and r_ret: #Si ambas capturas fueron exitosas...
258
259         #Imágenes a escala de grises.
260         l_gray = cv.cvtColor(l_img, cv.COLOR_BGR2GRAY)
261         r_gray = cv.cvtColor(r_img, cv.COLOR_BGR2GRAY)
262
263         cv.imshow('Left img.: Gray', l_gray)
264         cv.imshow('Right img.: Gray', r_gray)
265         key = cv.waitKey(33)
266
267         if key == ord('1'):
268
269             #Encontrar las esquinas del tablero de ajedrez.
270             l_ret, l_corners = cv.findChessboardCorners(l_gray, (ny,
                nx), None)
271             r_ret, r_corners = cv.findChessboardCorners(r_gray, (ny,
                nx), None)
272
273             if l_ret and r_ret: #Si encuentra las esquinas en ambas
                capturas...
274
275                 i += 1 #Incrementa el número de imágenes válidas.
276                 print('(Info) Válidas = '+str(i))
277
278                 objpts.append(ptnpts) #Añade los puntos de objeto (3D
                )
279
280                 l_corners2 = cv.cornerSubPix(l_gray, l_corners,

```

```

        (11,11), (-1,-1), criteria)
281     l_imgpts.append(l_corners2)
282
283     r_corners2 = cv.cornerSubPix(r_gray, r_corners,
        (11,11), (-1,-1), criteria)
284     r_imgpts.append(r_corners2)
285
286     #Dibujar y mostrar esquinas del tablero.
287     l_gray_bgr = cv.cvtColor(l_gray, cv.COLOR_GRAY2BGR)
288     r_gray_bgr = cv.cvtColor(r_gray, cv.COLOR_GRAY2BGR)
289
290     cv.drawChessboardCorners(l_gray_bgr, (ny,nx),
        l_corners2, l_ret)
291     cv.imshow(name + 'left_v-img', l_gray_bgr)
292
293     cv.drawChessboardCorners(r_gray_bgr, (ny,nx),
        r_corners2, r_ret)
294     cv.imshow(name + 'right_v-img', r_gray_bgr)
295     cv.waitKey(100)
296
297     #Guardar imágenes válidas.
298     l_vimg_name = generate_vimg_name(name + 'left', i)
299     cv.imwrite(rel_dir + l_vimg_name, l_gray)
300
301     r_vimg_name = generate_vimg_name(name + 'right', i)
302     cv.imwrite(rel_dir + r_vimg_name, r_gray)
303
304
305 l_cam.release()
306 r_cam.release()
307 cv.destroyAllWindows()
308 print('\n')
309
310
311 parent_folder_dir = os.path.normpath(os.getcwd() + os.sep + os.pardir
    ) #Retrocede al directorio padre.
312
313 parent_rel_dir = '/mono_intrinsics/mono/' #Dirección relativa al
    directorio padre cam. izq..
314 file_name = 'mono_intr.npz' #Nombre del archivo .npz.
315 path = os.path.abspath(parent_folder_dir + parent_rel_dir + file_name
    )
316
317 #Cargar archivo con parámetros intrínsecos de las cámaras.
318 npzfile = np.load(path)
319
320 imtx = npzfile['imtx']
321 distc = npzfile['distc']
322
323
324 #Realizar calibración estéreo.
325 retvals = []

```

```

326 retvals = cv.stereoCalibrate(objpts, l_imgpts, r_imgpts, imtx, distc,
    imtx, distc,
327                                l_gray.shape[:: -1], flags=cv.
                                    CALIB_FIX_INTRINSIC)
328
329 reproy_err_rms = retvals[0] #Error de re-proyección RMS en [px].
330 #Parámetros resultantes de la calibración estéreo; describen a la cá
    mara izquierda respecto de la derecha:
331 R = retvals[5] #Matriz de rotación del sistema coordinado.
332 T = retvals[6] #Vector de traslación del sistema coordinado.
333 F = retvals[8] #Matriz fundamental.
334
335 Tx = T[0][0]
336 Ty = T[1][0]
337 Tz = T[2][0]
338 B = np.sqrt([Tx**2 + Ty**2 + Tz**2])
339 B = B[0] #Distancia base (entre centros focales) en [mm].
340
341
342 #Obtener matrices de rectificación estéreo.
343 retvals = []
344 retvals = cv.stereoRectify(imtx, distc, imtx, distc, l_gray.shape
    [:: -1], R, T)
345
346 l_R = retvals[0] #Matriz de rectificación por rotación para la cámara
    izquierda.
347 r_R = retvals[1] #Matriz de rectificación por rotación para la cámara
    derecha.
348 l_P = retvals[2] #Matriz de proyección en el sist. coord. rectificado
    para la cámara izquierda.
349 r_P = retvals[3] #Matriz de proyección en el sist. coord. rectificado
    para la cámara derecha.
350
351
352 #Obtener mapas para eliminar distorsión y rectificar las imágenes.
353 l_map1, l_map2 = cv.initUndistortRectifyMap(imtx, distc, l_R, l_P,
    l_gray.shape[:: -1], cv.CV_32FC1)
354 r_map1, r_map2 = cv.initUndistortRectifyMap(imtx, distc, r_R, r_P,
    l_gray.shape[:: -1], cv.CV_32FC1)
355
356
357 print('(Info) Error RMS de re-proyección [px] = '+str(reproy_err_rms)
    )
358 print('\n')
359
360
361 #Cálculo de error RMS de matriz fundamental (F) en base a los puntos
    de imagen encontrados y según  $[p_2|1]*F*[p_1|1]^T = 0$ .
362 quad_sum = 0
363
364 for i in range(0, len(l_imgpts)):
365

```

```

366     l_pt_i = [[l_imgpts[0][i][0][0], l_imgpts[0][i][0][1], 1]]
367     r_pt_i = [[r_imgpts[0][i][0][0], r_imgpts[0][i][0][1], 1]]
368
369     error_i = np.dot(np.dot(r_pt_i, F), np.transpose(l_pt_i))
370     error_i = error_i[0][0]
371
372     quad_sum += (error_i*error_i)
373
374 quad_sum_mean = [quad_sum/len(l_imgpts)]
375 F_RMS_error = np.sqrt(quad_sum_mean)
376 F_RMS_error = round(F_RMS_error[0], 5)
377
378 print('(Info) Error RMS F = ' + str(F_RMS_error) + ' [px]')
379 print('\n')
380
381
382 #Guardar archivo .npz con los parámetros estéreo.
383 np.savez(rel_dir + name + 'stereo_intr', l_map1=l_map1, l_map2=l_map2
384         , r_map1=r_map1, r_map2=r_map2,
385         l_P=l_P, B=B,
386         l_cam_num=l_cam_num,
387         r_cam_num=r_cam_num)
388
389 print('(Info) Archivo .npz con parámetros intrínsecos estéreo
390       generado!')
391 print('\n')

```

B.3. Módulo de Python con código para calibración de parámetros extrínsecos

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4 import cv2 as cv
5 import cv2.aruco as aruco
6 import os
7
8
9 print('')
10
11 #Información del patrón de calibración.
12 print('Datos sobre el patrón de calibración:')
13 print('')
14
15 size = float(input('    >> Tamaño de lado del ArUco [mm] = '))
16 print('\n')
17
18
19 #Cargar archivo .npz con parámetros intrínsecos estéreo.
20 parent_folder_dir = os.path.normpath(os.getcwd() + os.sep + os.pardir
    ) #Retrocede al directorio padre.
21 parent_rel_dir = '/stereo_intrinsics/stereo/'
22 file_name = 'stereo_intr.npz'
23 path = os.path.abspath(parent_folder_dir + parent_rel_dir + file_name
    )
24
25 npzfile = np.load(path)
26
27 cam_num = npzfile['l_cam_num']
28
29 map1 = npzfile['l_map1']
30 map2 = npzfile['l_map2']
31
32 imtx_rect = npzfile['l_P']
33 imtx_rect = imtx_rect[:, :-1] #Matriz aparente (tras rectificación) de
    la cámara izquierda.
34
35
36 #Instrucciones del funcionamiento del bucle de calibración extrínseca
    .
37 print('>> Procesar imágenes:')
38 print('')
39 print('    Durante cualquier ventana de captura activa, presionar [
    key]:')
40 print('')
41 print('    [1] Procesar imagen')
```

```

42 print('          [2] Ingresar coordenadas')
43 print('          [s] Generar archivo .npz con parámetros extrínsecos')
44 print('\n')
45
46 print('(Info) Inicializando cámara...')
47 print('')
48
49 cam = cv.VideoCapture(cam_num) #Inicia la captura de video de cám.
    izq.
50
51 print('(Info) Cámara inicializada!')
52 print('\n')
53
54 aruco_dict = aruco.Dictionary_get(aruco.DICT_6X6_250)
55 params = aruco.DetectorParameters_create()
56 params.adaptiveThreshConstant = 10
57 params.cornerRefinementMethod = aruco.CORNER_REFINE_SUBPIX
58
59 flag0 = True
60
61 while flag0: #Bucle calib. extr.
62
63     ret, img = cam.read() #Lee un frame de la cámara.
64
65     if ret: #Si la captura fue exitosa...
66
67         gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) #Imagen a escala
            de grises.
68         gray_rect = cv.remap(gray, map1, map2, cv.INTER_LINEAR)
69
70         cv.imshow('Left img.: Gray rect.', gray_rect)
71         k = cv.waitKey(33) #Almacena tecla presionada durante ventana
            activa.
72
73         if k == ord('l'): #Si la tecla presionada es "l"...
74
75             #Obtener intersecciones, id del aruco y puntos rechazados
                .
76             corners, ids, rejectedImgPoints = aruco.detectMarkers(
                gray_rect, aruco_dict, parameters=params)
77
78             corners = np.float32(corners)
79
80             if np.all(ids != None):
81
82                 #Estimar pose del aruco (rotación y traslación)
                    respecto de la cámara izq.
83                 rvec, tvec, _ = aruco.estimatePoseSingleMarkers(
                    corners, size, imtx_rect, None)
84
85                 tvec = tvec[0]
86                 rvec = rvec[0]

```

```

87
88         gray_rect = cv.cvtColor(gray_rect, cv.COLOR_GRAY2BGR)
89
90         #Dibujar ejes (Rojo: +X, Verde: +Y, Azul: +Z)
91         img_cordsys = aruco.drawAxis(gray_rect, imtx_rect,
92                                     None, rvec, tvec, 3*29)
93
94         img_cordsys = aruco.drawDetectedMarkers(gray_rect,
95                                                  corners)
96
97         cv.imshow('img_cordsys', img_cordsys)
98         cv.waitKey(33)
99
100        print('rvec =')
101        print(rvec)
102        print('')
103
104        print('tvec =')
105        print(tvec)
106        print('\n')
107
108    elif k == ord('2'):
109
110        #Bucle para ingresar coordenadas del punto de referencia
111        en base al sistema de YARBIZ.
112        flag1 = True
113
114        while flag1:
115
116            print('Ingresar coordenadas (x,y,z) del origen
117                  XYZ_calib en base a XYZ_YARBIZ:')
118            print('')
119
120            x_Oc_Oy = float(input('      >> x_Oc_Oy [mm] = '))
121            y_Oc_Oy = float(input('      >> y_Oc_Oy [mm] = '))
122            z_Oc_Oy = float(input('      >> z_Oc_Oy [mm] = '))
123            print('\n')
124
125            flag2 = True
126
127            while flag2:
128
129                print('      [1] Guardar coordenadas y continuar')
130                print('      [2] Volver a ingresar coordenadas')
131                print('')
132
133                o = input('      >> Opción: ')
134                print('\n')
135
136                if o == '1':
137
138                    flag1 = False

```

```

135         flag2 = False
136
137     elif o == '2':
138
139         flag2 = False
140
141     else:
142
143         print('      (Error) Opción inválida!')
144         print('\n')
145
146
147 elif k == ord('s'): #Si la tecla presionada es "s"...
148
149     try:
150
151         cv.imwrite('mono/left_cordsys_calib.png', img_cordsys
152             ) #Guarda imagen con ejes dibujados.
153
154         rmtx, jacob = cv.Rodrigues(rvec[0]) #Transforma rvec
155             en una matriz de rotaciones 3x3.
156
157         #Se genera matriz de transformación homogénea de sist
158             . rect. cám. izq. respecto a .sist. calib.
159         HT_Oc_Olrect = np.zeros((4,4))
160         HT_Oc_Olrect[3,3] = 1
161         HT_Oc_Olrect[:-1, :-1] = rmtx[:, :]
162         HT_Oc_Olrect[:-1, 3] = tvec
163
164         HT_Olrect_Oc = np.linalg.inv(HT_Oc_Olrect)
165
166         #Crear matriz de transformación homogénea de
167             XYZ_calib respecto a XYZ_yarbiz.
168         thetax_Oc_Oy = 90*np.pi/180
169         Rx_Oc_Oy = np.array([[1,0,0],
170             [0,np.cos(thetax_Oc_Oy),-np.sin(
171                 thetax_Oc_Oy)],
172             [0,np.sin(thetax_Oc_Oy),np.cos(
173                 thetax_Oc_Oy)])])
174
175         T_Oc_Oy = np.array([[x_Oc_Oy,y_Oc_Oy,z_Oc_Oy]])
176
177         HT_Oc_Oy = np.zeros((4,4))
178         HT_Oc_Oy[:-1, :-1] = Rx_Oc_Oy
179         HT_Oc_Oy[:-1, 3] = T_Oc_Oy
180         HT_Oc_Oy[3,3] = 1
181
182         #Matriz de transf. hom. de sist. rect. cám. izq.
183             respecto de sist. YARBIZ.
184         HT_Olrect_Oy = np.dot(HT_Oc_Oy, HT_Olrect_Oc)

```

```

180         #Crear archivo .npz con parámetros extrínsecos.
181         np.savez('mono/mono_extr', HT_Olrect_Oy=HT_Olrect_Oy)
182
183         print('(Info) Archivo .npz con parámetros extrínsecos
184               generado!')
185         print('\n')
186
187         flag0 = False
188
189     except:
190
191         print('(Error) No se ha procesado la imagen y/o no se
192               han ingresado coordenadas!')
193
194 cam.release()
195 cv.destroyAllWindows()

```

B.4. Módulo de Python con código para detectar la posición del objeto

```
1 # -*- coding: cp1252 -*-
2
3 import numpy as np
4 from yarbiz_kine_v3 import ik
5 from yarbiz_kine_v3 import fk
6 import cv2 as cv
7 import os
8 import time
9
10
11 print('')
12
13 com = 'COM3' #Almacenar puerto COM de OpenCM9.04
14 baud = 115200 #Almacenar Baud Rate.
15
16 segm = 50 #Tamaño del segmento de movimiento de YARBIZ
17
18 tol = 3 #Tolerancia en [mm] del resultado final del movimiento.
19
20 diam = 40 #Diámetro objeto en [mm].
21
22 thetai_reset = [-90,150,-90,-45,0] #Ángulos de posición inicial de
    YARBIZ.
23
24 thetai_bring = [45,65,-60,-45,0]
25
26 thetai, p = fk(thetai_reset, com, baud) #Mover YARBIZ a posición
    inicial.
27
28
29 #Cargar archivo .npz de la calibración estéreo.
30 rel_dir = 'calibration/stereo_intrinsics/stereo/' #Dirección relativa
    .
31 file_name = 'stereo_intr.npz' #Nombre del archivo .npz.
32 path = rel_dir + file_name
33
34 npzfile = np.load(path)
35
36 l_cam_num = npzfile['l_cam_num'] #Números de dispositivo de las cá
    maras.
37 r_cam_num = npzfile['r_cam_num']
38
39 l_map1 = npzfile['l_map1'] #Mapas de rectificación para la cámara
    izquierda.
40 l_map2 = npzfile['l_map2']
41
```

```

42 r_map1 = npzfile['r_map1'] #Mapas de rectificación para la cámara
    derecha.
43 r_map2 = npzfile['r_map2']
44
45 l_P = npzfile['l_P']
46 f = l_P[0][0] #Largo focal tras rectificación.
47
48 #Coordenadas del centro de imagen de cám. izq. tras rectificación.
49 l_cx = l_P[0][2]
50 l_cy = l_P[1][2]
51
52 B = npzfile['B'] #Distancia entre centros focales.
53
54
55 #Cargar archivo .npz de la calibración mono extrínseca.
56 rel_dir = 'calibration/mono_extrinsics/mono/' #Dirección relativa.
57 file_name = 'mono_extr.npz' #Nombre del archivo .npz.
58 path = rel_dir + file_name
59
60 npzfile = np.load(path)
61
62 HT_Olrect_Oy = npzfile['HT_Olrect_Oy']
63
64
65 #Cargar archivo .npz con parámetros de SGBM, WLS, HSVMask, id. de
    curvas.
66
67 try:
68
69     npzfile = np.load('svconfig.npz')
70
71 except: #En caso de que no se encuentre el archivo .npz, se genera
    uno nuevo con valores preestablecidos.
72
73     np.savez('svconfig.npz', minDisp=0,
74               numDisps=7,
75               bsize=3,
76               SAD_win_size=5,
77               pfc=63,
78               uRatio=10,
79               sp_win_size=0,
80               sp_range=2,
81               lmbda=80000,
82               sigma=1.2,
83               lowerH=0,
84               lowerS=0,
85               lowerV=0,
86               upperH=180,
87               upperS=255,
88               upperV=255,
89               precn_lvl=100,
90               min_sides=30)

```

```

91
92     npzfile = np.load('svconfig.npz')
93
94 #Parámetros HSVMask.
95 lowerH = npzfile['lowerH']
96 lowerS = npzfile['lowerS']
97 lowerV = npzfile['lowerV']
98 upperH = npzfile['upperH']
99 upperS = npzfile['upperS']
100 upperV = npzfile['upperV']
101
102 #Parámetros Identificación de curvas.
103 precn_lvl = npzfile['precn_lvl']
104 min_sides = npzfile['min_sides']
105
106 #Parámetros SGBM:
107 img_chans = 1 #Image Channels
108 minDisp = npzfile['minDisp'] #MinDisparity.
109 numDisps = npzfile['numDisps'] #NumDisparities.
110 bsize = npzfile['bsize'] #BlockSize.
111 SAD_win_size = npzfile['SAD_win_size']
112 pfc = npzfile['pfc'] #PreFilterCap.
113 uRatio = npzfile['uRatio'] #UniquenessRatio.
114 sp_win_size = npzfile['sp_win_size'] #SpeckleWindowSize.
115 sp_range = npzfile['sp_range'] #SpeckleRange.
116
117 #Definir parámetros filtro WLS.
118 lmbda = npzfile['lmbda']
119 sigma = npzfile['sigma']
120 visual_multiplier = 1.0
121
122 #Almacenar par evaluar cambios.
123 minDisp0 = minDisp #MinDisparity.
124 numDisps0 = numDisps #NumDisparities.
125 bsize0 = bsize #BlockSize.
126 SAD_win_size0 = SAD_win_size
127 pfc0 = pfc #PreFilterCap.
128 uRatio0 = uRatio #UniquenessRatio.
129 sp_win_size0 = sp_win_size #SpeckleWindowSize.
130 sp_range0 = sp_range #SpeckleRange.
131 lmbda0 = lmbda
132 sigma0 = sigma
133
134
135 #Crear objeto StereoSGBM para el cálculo de disparidad.
136 #Crear matcher izquierdo.
137 l_matcher = cv.StereoSGBM_create(minDisparity=minDisp,
138                                   numDisparities=numDisps*16,
139                                   blockSize=bsize,
140                                   P1=8*img_chans*SAD_win_size**2,
141                                   P2=32*img_chans*SAD_win_size**2,
142                                   disp12MaxDiff=-1,

```

```

143                                     preFilterCap=pfc,
144                                     uniquenessRatio=uRatio,
145                                     speckleWindowSize=sp_win_size,
146                                     speckleRange=sp_range,
147                                     mode=cv.STEREO_SGBM_MODE_SGBM_3WAY)
148
149 #Inicializar filtro WLS.
150 r_matcher = cv.ximgproc.createRightMatcher(l_matcher) #Crear matcher
    derecho.
151
152 wls_filter = cv.ximgproc.createDisparityWLSFilter(matcher_left=
    l_matcher)
153 wls_filter.setLambda(lmbda)
154 wls_filter.setSigmaColor(sigma)
155
156
157 #Crear sliders:
158 def nothing(x):
159     pass
160
161
162 print('>> Configurar:')
163 print('')
164 print('    Durante cualquier ventana de captura activa, presionar [
    key]:')
165 print('')
166 print('    [s] Guardar cambios')
167 print('\n')
168
169 print('(Info) Inicializando panel de sliders...')
170 print('')
171
172 time.sleep(4)
173
174 cfg_w_name = 'Stereo config.' #Nombre de la ventana de sliders.
175 cv.namedWindow(cfg_w_name, cv.WINDOW_NORMAL) #Crear ventana para
    sliders.
176
177 slrs = {'minDisp'      : ['minDisp', 0, 200],
178         'numDisps'     : ['numDisps', 1, 20],
179         'bsize'        : ['BlockSize', 1, 15],
180         'SAD_win_size' : ['SADWinSize', 3, 31],
181         'pfc'          : ['PreFilCap', 0, 100],
182         'uRatio'       : ['UniqRatio', 0, 20],
183         'sp_win_size'  : ['SpWinSize', 0, 220],
184         'sp_range'     : ['SpRange', 0, 5],
185         'lmbda'        : ['Lambda', 0, 200000],
186         'sigma'        : ['Sigma', 0, 200],
187         'lowerH'       : ['Lower H', 0, 180],
188         'lowerS'       : ['Lower S', 0, 255],
189         'lowerV'       : ['Lower V', 0, 255],
190         'upperH'       : ['Upper H', 0, 180],

```

```

191         'upperS'      : ['Upper S', 0, 255],
192         'upperV'      : ['Upper V', 0, 255],
193         'precn_lvl'    : ['CurvePrLvl', 1, 200],
194         'min_sides'    : ['CurveMinSi', 3, 100]
195     }
196
197 cv.createTrackbar(slrs['minDisp'][0], cfg_w_name, minDisp, slrs['
    minDisp'][2], nothing)
198 cv.createTrackbar(slrs['numDisps'][0], cfg_w_name, numDisps, slrs['
    numDisps'][2], nothing)
199 cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize, slrs['bsize'
    ][2], nothing)
200 cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name, SAD_win_size,
    slrs['SAD_win_size'][2], nothing)
201 cv.createTrackbar(slrs['pfc'][0], cfg_w_name, pfc, slrs['pfc'][2],
    nothing)
202 cv.createTrackbar(slrs['uRatio'][0], cfg_w_name, uRatio, slrs['uRatio
    '][2], nothing)
203 cv.createTrackbar(slrs['sp_win_size'][0], cfg_w_name, sp_win_size,
    slrs['sp_win_size'][2], nothing)
204 cv.createTrackbar(slrs['sp_range'][0], cfg_w_name, sp_range, slrs['
    sp_range'][2], nothing)
205 cv.createTrackbar(slrs['lmbda'][0], cfg_w_name, lmbda, slrs['lmbda'
    ][2], nothing)
206 cv.createTrackbar(slrs['sigma'][0], cfg_w_name, int(sigma*100), slrs[
    'sigma'][2], nothing)
207 cv.createTrackbar(slrs['lowerH'][0], cfg_w_name, lowerH, slrs['lowerH
    '][2], nothing)
208 cv.createTrackbar(slrs['upperH'][0], cfg_w_name, upperH, slrs['upperH
    '][2], nothing)
209 cv.createTrackbar(slrs['lowerS'][0], cfg_w_name, lowerS, slrs['lowerS
    '][2], nothing)
210 cv.createTrackbar(slrs['upperS'][0], cfg_w_name, upperS, slrs['upperS
    '][2], nothing)
211 cv.createTrackbar(slrs['lowerV'][0], cfg_w_name, lowerV, slrs['lowerV
    '][2], nothing)
212 cv.createTrackbar(slrs['upperV'][0], cfg_w_name, upperV, slrs['upperV
    '][2], nothing)
213 cv.createTrackbar(slrs['precn_lvl'][0], cfg_w_name, precn_lvl, slrs['
    precn_lvl'][2], nothing)
214 cv.createTrackbar(slrs['min_sides'][0], cfg_w_name, min_sides, slrs['
    min_sides'][2], nothing)
215
216 change = 0 #Evaluador de cambio en config.
217
218 print('(Info) Panel de sliders inicializado!')
219 print('\n')
220
221 #Valor inicial de coord. en pixeles.
222 u = 0
223 v = 0
224

```

```

225 #Funcion para obtener coordenadas u,v en pixeles mediante lclick.
226 def get_px(event, x, y, flags, param):
227
228     global u,v
229
230     if event == cv.EVENT_LBUTTONDOWN:
231
232         [u,v] = [x,y]
233
234
235
236 font = cv.FONT_HERSHEY_SIMPLEX #Fuente de texto.
237 fsize1 = 0.5 #Tamaño de fuente 1 (Curvas).
238 fsize2 = 0.4 #Tamaño de fuente 2 (Info.).
239 linespace = 16 #Interlineado.
240
241 mode = 'auto' #Modo de cálculo de coordenadas (u,v).
242 curvenum = 1 #Número de curva seleccionada.
243
244 print('>> MODO AUTO. Durante ventana activa, presionar [key]:')
245 print('')
246 print('    [1-5] Seleccionar curva')
247 print('\n')
248
249 print('>> MODO MANUAL. En ventana de imagen gris izq. rectificada,
        presionar [key] ')
250 print('')
251 print('    [Click Izq.] Seleccionar un punto')
252 print('\n')
253
254 print('>> MOVIMIENTO. Durante ventana activa, presionar [key]:')
255 print('')
256 print('    [Enter] Generar movimiento')
257 print('\n')
258
259 print('>> AGARRE. Durante ventana activa, presionar [key]:')
260 print('')
261 print('    [Espacio] Generar agarre')
262 print('\n')
263
264
265 print('(Info) Inicializando cámaras...')
266 print('')
267
268 #Inicia la captura de video de las cámaras.
269 l_cam = cv.VideoCapture(l_cam_num)
270 r_cam = cv.VideoCapture(r_cam_num)
271
272 #Nombre de las ventanas de captura de las cámaras.
273 l_win_name = 'Left img.: '
274 r_win_name = 'Right img.: '
275

```

```

276 time.sleep(4)
277
278 print('(Info) Cámaras inicializadas!')
279 print('\n')
280
281 #Bucle de estéreovisión.
282 flag = True
283
284 while flag:
285
286     #Leer las capturas.
287     l_ret, l_img = l_cam.read()
288     r_ret, r_img = r_cam.read()
289
290     if l_ret and r_ret: #Si las capturas fueron leídas con éxito...
291
292         #Rectificar imágenes según mapas de rectificación obtenidos
293         #en calibración estéreo.
294         l_img_rect = cv.remap(l_img, l_map1, l_map2, cv.INTER_LINEAR)
295         r_img_rect = cv.remap(r_img, r_map1, r_map2, cv.INTER_LINEAR)
296
297         #Cálculo del mapa de disparidad.
298
299         #Imágenes a escala de grises.
300         l_gray_rect = cv.cvtColor(l_img_rect, cv.COLOR_BGR2GRAY)
301         r_gray_rect = cv.cvtColor(r_img_rect, cv.COLOR_BGR2GRAY)
302
303         #Mapas de disparidad izquierdo y derecho.
304         l_disp = l_matcher.compute(l_gray_rect, r_gray_rect)
305         r_disp = r_matcher.compute(r_gray_rect, l_gray_rect)
306
307         #Transformación de valores de mapas de disparidad a tipo
308         #int16.
309         l_disp = np.int16(l_disp)
310         r_disp = np.int16(r_disp)
311
312         #Generar mapa de disparidad basado en imagen izq. con filtro
313         #WLS aplicado.
314         disp = wls_filter.filter(l_disp, l_gray_rect, None, r_disp)
315
316         #Normalizar mapa de disparidad y transformar datos a valores
317         #de saturación en escala de grises SAT[0,255].
318         disp_norm = wls_filter.filter(l_disp, l_gray_rect, None,
319                                     r_disp)
320         disp_norm = cv.normalize(src=disp_norm, dst=disp_norm,
321                                beta=0, alpha=255,
322                                norm_type=cv.NORM_MINMAX)
323
324         disp_norm = np.uint8(disp_norm) #Transformar valores a tipo
325         #uint8.

```

```

322
323     #Escala de grises a BGR.
324     l_gray_rect = cv.cvtColor(l_gray_rect, cv.COLOR_GRAY2BGR)
325     r_gray_rect = cv.cvtColor(r_gray_rect, cv.COLOR_GRAY2BGR)
326     disp_norm = cv.cvtColor(disp_norm, cv.COLOR_GRAY2BGR)
327
328
329     #Modo automático (basado en detección de figuras).
330     if mode == 'auto':
331
332         #Detección de figuras geométricas y cálculo de centroides
333         .
334         l_hsv_rect = cv.cvtColor(l_img_rect, cv.COLOR_BGR2HSV) #
335             Canales BGR a HSV.
336
337         #Definir límites superior e inferior de canal HSV.
338         lowerHSV = np.array([lowerH, lowerS, lowerV])
339         upperHSV = np.array([upperH, upperS, upperV])
340
341         #Generar imagen binaria (máscara): - blanco para lo que
342             se encuentre dentro de los límites HSV.
343             # - negro para lo que se
344             encuentre fuera.
345         l_hsv_mask = cv.inRange(l_hsv_rect, lowerHSV, upperHSV)
346
347         contours, hie = cv.findContours(l_hsv_mask, 0, 2) #
348             Encontrars contornos.
349
350         try: #Manejar errores en caso de no encontrar curvas.
351
352             validCurves = [] #Lista para almacenar curvas válidas
353             .
354             for c in contours:
355
356                 arclen = cv.arcLength(c, True) #Perímetro de la
357                     figura. True = figura cerrada.
358                 epsilon = arclen*(1/(precn_lvl*10)) #Precisión de
359                     la aproximación de las curvas.
360                 appCurve = cv.approxPolyDP(c, epsilon, True) #
361                     Aproximar curva.
362
363                 if len(appCurve) >= min_sides: #Número de lados
364                     requerido luego de aproximación.
365
366                     validCurves.append(appCurve)
367
368             centroids = [] #Lista para almacenar centroides de
369                 las curvas válidas.

```

```

363
364     for vc in validCurves: #Calcular centroides de las
365                               curvas válidas.
366
367         moments = cv.moments(vc) #Cálculo de los momentos
368                                   de la curva.
369
370         #Coordenada u [px] del centroide.
371         centr_u = moments['m10']/moments['m00']
372         centr_u = round(centr_u, 0)
373         centr_u = int(centr_u)
374
375         #Coordenada v [px] del centroide.
376         centr_v = moments['m01']/moments['m00']
377         centr_v = round(centr_v, 0)
378         centr_v = int(centr_v)
379
380         centr = (centr_u, centr_v) #Tupla con coordenadas
381                                   (u,v) [px] del centroide.
382
383         centroids.append(centr) #Añade centroide a la
384                                   lista.
385
386     centdisps = []
387
388     for i in range(0, len(validCurves)): #Dibujar curvas y
389                                           centroides de las curvas.
390
391         u = centroids[i][0]
392         v = centroids[i][1]
393
394         if i+1 == curvenum:
395
396             curvecolor = (0,255,0)
397
398         else:
399
400             curvecolor = (0,0,255)
401
402         cv.drawContours(l_gray_rect, [validCurves[i]],
403                         -1, curvecolor, 1) #Dibujar contorno.
404
405         cv.circle(l_gray_rect, (u,v), 2, curvecolor, -1)
406               #Dibujar centroide.
407
408         cv.putText(l_gray_rect, str(i+1), (u+5,v), font,
409                   fsize1, curvecolor, 1, cv.LINE_AA)
410
411         disp_uv = disp[v][u]/16 #Encontrar disparidad del
412                                   centroide.
413

```

```

406         cv.circle(r_gray_rect, (u-int(round(disg_uv,0)),v
           ), 2, curvecolor, -1) #Dibujar centroide en
           imagen der.
407
408         cv.putText(r_gray_rect, str(i+1), (u-int(round(
           disg_uv,0))+5,v), font, fsize1, curvecolor,
           1, cv.LINE_AA)
409
410         centdisps.append(disg_uv) #Añade disparidad a la
           lista.
411
412
413         disg_uv = centdisps[curvenum-1]
414         u = centroids[curvenum-1][0]
415         v = centroids[curvenum-1][1]
416
417
418     except:
419
420         disg_uv = '-'
421
422         print('(Info) Curva(s) inexistente(s)!')
423         print('\n')
424
425
426         cv.imshow(l_win_name + 'HSV rect.', l_img_rect)
427         cv.imshow(l_win_name + 'HSV mask ', l_hsv_mask)
428
429
430     #Modo manual (basado en coordenada dada por el usuario).
431     elif mode == 'manual':
432
433         #Dibujar círculo en color en imagen izquierda gris
           rectificada.
434         cv.setMouseCallback(l_win_name + 'Gray rect.', get_px) #
           Llamar función get_px.
435         cv.circle(l_gray_rect, (u,v), 2, (0,255,0), -1) #Dibujar
           un círculo en las coordenadas u,v obtenidas con
           funcion get_px.
436
437         disg_uv = disp[v][u]/16 #Calcular valor de disparidad en
           [px].
438
439         #Dibujar círculo en color en imagen derecha según
           disparidad calculada.
440         cv.circle(r_gray_rect, (u-int(round(disg_uv,0)),v), 2,
           (0,255,0), -1)
441
442
443
444     #Calcular triangulación de coordenadas (respecto a sist.
           coord. rectificado de cámara izquierda).

```

```

445     try: #Intentar calcular las coordenadas a partir de la
446           disparidad.
447
448           z_Olrect = B*f/disp_uv
449           x_Olrect = z_Olrect*(u-l_cx)/f
450           y_Olrect = z_Olrect*(v-l_cy)/f
451
452           if mode == 'auto':
453
454               norm = np.sqrt([x_Olrect**2 + y_Olrect**2 + z_Olrect
455                               **2])
456               norm = norm[0] #Módulo de p_Olrect.
457
458               dx_Olrect = x_Olrect/norm
459               dy_Olrect = y_Olrect/norm
460               dz_Olrect = z_Olrect/norm
461
462               x_Olrect = x_Olrect + dx_Olrect*(diam/2)
463               y_Olrect = y_Olrect + dy_Olrect*(diam/2)
464               z_Olrect = z_Olrect + dz_Olrect*(diam/2)
465
466           p_Olrect = np.array([x_Olrect], #Vector objetivo en base
467                                a Olrect.
468                                [y_Olrect],
469                                [z_Olrect],
470                                [1]))
471
472           #Calcular coordenadas en base al sist. coord. de YARBIZ.
473           p_Oy = np.dot(HT_Olrect_Oy, p_Olrect) #Vector objetivo
474           respecto a YARBIZ.
475           p_Oy = p_Oy[:-1,:]
476
477           if mode == 'auto':
478
479               norm = np.sqrt([p_Oy[0,0]**2 + p_Oy[1,0]**2])
480               norm = norm[0]
481
482               d_Oy = np.array([p_Oy[0,0]],
483                                [p_Oy[1,0]],
484                                [0]))
485
486               d_Oy = d_Oy*(1/norm)
487
488               p_Oy = p_Oy - d_Oy*(diam/2)
489
490           #Aproximación de coordenadas a un decimal (más cercano)
491           para prints.
492           x_Oy = round(p_Oy[0,0],1)
493           y_Oy = round(p_Oy[1,0],1)
494           z_Oy = round(p_Oy[2,0],1)

```

```

492     x_Olrect = round(x_Olrect,1)
493     y_Olrect = round(y_Olrect,1)
494     z_Olrect = round(z_Olrect,1)
495
496     #Display de texto con información de coordenadas y
         disparidad.
497     cv.putText(displ_norm, 'Modo = ' + mode, (5,1*linespace),
         font, fsize2, (0,255,0), 1, cv.LINE_AA)
498     cv.putText(displ_norm, 'Curva = ' + str(curvenum), (5,2*
         linespace), font, fsize2, (0,255,0), 1, cv.LINE_AA)
499     cv.putText(displ_norm, '(u,v) = (' + str(u) + ', ' + str(v)
         ) + ') [px]', (5,3*linespace), font, fsize2,
         (0,255,0), 1, cv.LINE_AA)
500     cv.putText(displ_norm, 'Disp. = ' + str(int(round(displ_uv
         ,0))) + ' [px]', (5,4*linespace), font, fsize2,
         (0,255,0), 1, cv.LINE_AA)
501     cv.putText(displ_norm, 'p_Oy = (' + str(x_Oy) + ', ' + str
         (y_Oy) + ', ' + str(z_Oy) + ') [mm]', (5,5*linespace)
         , font, fsize2, (0,255,0), 1, cv.LINE_AA)
502     cv.putText(displ_norm, 'p_Olrect = (' + str(x_Olrect) + ',
         ' + str(y_Olrect) + ', ' + str(z_Olrect) + ') [mm]',
         (5,6*linespace), font, fsize2, (0,255,0), 1, cv.
         LINE_AA)
503
504     except:
505
506         print('(Error) No se ha podido calcular la disparidad!')
507         print('\n')
508
509         cv.putText(displ_norm, 'Modo = ' + mode, (5,1*linespace),
         font, fsize2, (0,255,0), 1, cv.LINE_AA)
510         cv.putText(displ_norm, 'Curva = ' + str(curvenum), (5,2*
         linespace), font, fsize2, (0,255,0), 1, cv.LINE_AA)
511         cv.putText(displ_norm, '(u,v) = - [px]', (5,3*linespace),
         font, fsize2, (0,255,0), 1, cv.LINE_AA)
512         cv.putText(displ_norm, 'Disp. = - [px]', (5,4*linespace),
         font, fsize2, (0,255,0), 1, cv.LINE_AA)
513         cv.putText(displ_norm, 'p_Oy = - [mm]', (5,5*linespace),
         font, fsize2, (0,255,0), 1, cv.LINE_AA)
514         cv.putText(displ_norm, 'p_Olrect = - [mm]', (5,6*linespace)
         ), font, fsize2, (0,255,0), 1, cv.LINE_AA)
515
516
517     #Mostrar imágenes rectificadas de ambas cámaras y mapa de
         disparidad.
518     cv.imshow(l_win_name + 'Gray rect.', l_gray_rect)
519     cv.imshow(r_win_name + 'Gray rect.', r_gray_rect)
520     cv.imshow(l_win_name + 'Disparity map', displ_norm)
521
522
523     #Obtención y corrección de variables de sliders.
524     minDisp = cv.getTrackbarPos(slrs['minDisp'][0], cfg_w_name) #

```

```

525         MinDisparity.
526         if minDisp != minDisp0:
527             change = 1
528
529         numDisps = cv.getTrackbarPos(slrs['numDisps'][0], cfg_w_name)
530         #NumDisparities.
531         if numDisps != numDisps0:
532             if numDisps < slrs['numDisps'][1]:
533                 numDisps = slrs['numDisps'][1]
534                 cv.createTrackbar(slrs['numDisps'][0], cfg_w_name,
535                                 numDisps, slrs['numDisps'][2], nothing)
536             change = 1
537
538         bsize = cv.getTrackbarPos(slrs['bsize'][0], cfg_w_name) #
539         BlockSize.
540         if bsize != bsize0:
541             if bsize > slrs['bsize'][1] and bsize%2 == 0: #Si es par,
542                 se le resta 1 unidad.
543                 if bsize > bsize0:
544                     bsize += 1
545                 elif bsize < bsize0:
546                     bsize -= 1
547                 cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize
548                                 , slrs['bsize'][2], nothing)
549             elif bsize < slrs['bsize'][1]:
550                 bsize = slrs['bsize'][1]
551                 cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize
552                                 , slrs['bsize'][2], nothing)
553             change = 1
554
555         SAD_win_size = cv.getTrackbarPos(slrs['SAD_win_size'][0],
556                                         cfg_w_name)
557         if SAD_win_size != SAD_win_size0:
558             if SAD_win_size > slrs['SAD_win_size'][1] and
559                 SAD_win_size%2 == 0:
560                 if SAD_win_size > SAD_win_size0:
561                     SAD_win_size += 1
562                 elif SAD_win_size < SAD_win_size0:
563                     SAD_win_size -= 1
564                 cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name
565                                 , SAD_win_size, slrs['SAD_win_size'][2], nothing)
566             elif SAD_win_size < slrs['SAD_win_size'][1]:
567                 SAD_win_size = slrs['SAD_win_size'][1]
568                 cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name
569                                 , SAD_win_size, slrs['SAD_win_size'][2], nothing)
570             change = 1
571
572         pfc = cv.getTrackbarPos(slrs['pfc'][0], cfg_w_name) #
573         PreFilterCap.
574         if pfc != pfc0:
575             change = 1
576

```

```

565     uRatio = cv.getTrackbarPos(slrs['uRatio'][0], cfg_w_name) #
566         UniquenessRatio.
567     if uRatio != uRatio0:
568         change = 1
569
570     sp_win_size = cv.getTrackbarPos(slrs['sp_win_size'][0],
571         cfg_w_name) #SpeckleWindowSize.
572     if sp_win_size != sp_win_size0:
573         change = 1
574
575     sp_range = cv.getTrackbarPos(slrs['sp_range'][0], cfg_w_name)
576         #SpeckleRange.
577     if sp_range != sp_range0:
578         change = 1
579
580     lambda = cv.getTrackbarPos(slrs['lambda'][0], cfg_w_name)
581     if lambda != lambda0:
582         change = 1
583
584     sigma = cv.getTrackbarPos(slrs['sigma'][0], cfg_w_name)
585     sigma = round(sigma/100,2)
586     if sigma != sigma0:
587         change = 1
588
589     lowerH = cv.getTrackbarPos(slrs['lowerH'][0], cfg_w_name)
590     upperH = cv.getTrackbarPos(slrs['upperH'][0], cfg_w_name)
591     lowerS = cv.getTrackbarPos(slrs['lowerS'][0], cfg_w_name)
592     upperS = cv.getTrackbarPos(slrs['upperS'][0], cfg_w_name)
593     lowerV = cv.getTrackbarPos(slrs['lowerV'][0], cfg_w_name)
594     upperV = cv.getTrackbarPos(slrs['upperV'][0], cfg_w_name)
595
596     precn_lvl = cv.getTrackbarPos(slrs['precn_lvl'][0],
597         cfg_w_name)
598     if precn_lvl < slrs['precn_lvl'][1]:
599         precn_lvl = slrs['precn_lvl'][1]
600         cv.createTrackbar(slrs['precn_lvl'][0], cfg_w_name,
601             precn_lvl, slrs['precn_lvl'][2], nothing)
602
603     min_sides = cv.getTrackbarPos(slrs['min_sides'][0],
604         cfg_w_name)
605     if min_sides < slrs['min_sides'][1]:
606         min_sides = slrs['min_sides'][1]
607         cv.createTrackbar(slrs['min_sides'][0], cfg_w_name,
608             min_sides, slrs['min_sides'][2], nothing)
609
610     #Almacenar para evaluar cambios.
611     minDisp0 = minDisp #MinDisparity.
612     numDisps0 = numDisps #NumDisparities.
613     bsize0 = bsize #BlockSize.
614     SAD_win_size0 = SAD_win_size
615     pfc0 = pfc #PreFilterCap.

```

```

610     uRatio0 = uRatio #UniquenessRatio.
611     sp_win_size0 = sp_win_size #SpeckleWindowSize.
612     sp_range0 = sp_range #SpeckleRange.
613     lambda0 = lambda
614
615     if change == 1: #Si algún valor cambió...
616
617         #Reconfigurar SGBM y WLS:
618         l_matcher = cv.StereoSGBM_create(minDisparity=minDisp,
619                                         numDisparities=numDisps
620                                         *16,
621                                         blockSize=bsize,
622                                         P1=8*img_chans*
623                                         SAD_win_size**2,
624                                         P2=32*img_chans*
625                                         SAD_win_size**2,
626                                         disp12MaxDiff=-1,
627                                         preFilterCap=pfc,
628                                         uniquenessRatio=uRatio,
629                                         speckleWindowSize=
630                                         sp_win_size,
631                                         speckleRange=sp_range,
632                                         mode=cv.
633                                         STEREO_SGBM_MODE_SGBM_3WAY
634                                         )
635
636         r_matcher = cv.ximgproc.createRightMatcher(l_matcher) #
637         # Crear matcher derecho.
638
639         wls_filter = cv.ximgproc.createDisparityWLSFilter(
640             matcher_left=l_matcher)
641         wls_filter.setLambda(lambda)
642         wls_filter.setSigmaColor(sigma)
643
644         change = 0
645
646     key = cv.waitKey(33)
647
648     #Bloque de control según [key] presionada.
649
650     if key == ord('s'): #Guardar configuración.
651
652         np.savez('svconfig.npz', minDisp=minDisp,
653                 numDisps=numDisps,
654                 bsize=bsize,
655                 SAD_win_size=SAD_win_size,
656                 pfc=pfc,
657                 uRatio=uRatio,
658                 sp_win_size=sp_win_size,
659                 sp_range=sp_range,
660                 lambda=lambda,

```

```

654         sigma=sigma,
655         lowerH = lowerH,
656         lowerS = lowerS,
657         lowerV = lowerV,
658         upperH = upperH,
659         upperS = upperS,
660         upperV = upperV,
661         precn_lvl = precn_lvl,
662         min_sides = min_sides)
663
664     print('\n')
665     print('(Info) Archivo svconfig generado!')
666     print('\n')
667     print('')
668
669     if key == ord('n'):
670
671         if mode == 'manual':
672             curvenum = 1
673         else:
674             pass
675
676         mode = 'auto'
677
678     if key == ord('m'):
679
680         mode = 'manual'
681
682         curvenum = '-'
683
684     if mode == 'auto':
685
686         if key == ord('1'):
687
688             curvenum = 1
689
690         elif key == ord('2'):
691
692             curvenum = 2
693
694         elif key == ord('3'):
695
696             curvenum = 3
697
698         elif key == ord('4'):
699
700             curvenum = 4
701
702         elif key == ord('5'):
703
704             curvenum = 5
705

```

```

706         if key == ord('\r'):
707
708             success, thetai, p = ik(p_Oy, p, thetai, segm, tol, com,
                                     baud)
709
710         if key == 32 and success == True:
711
712             print('(Info) Generando agarre...')
713             print('')
714             time.sleep(2)
715             print('(Info) Objeto sujeto!')
716             print('')
717
718             print('(Info) Llevando objeto a ubicación deseada...')
719             print('\n')
720             thetai, p = fk(thetai_bring, com, baud)
721
722             time.sleep(6)
723
724             print('(Info) Liberando objeto...')
725             print('')
726             time.sleep(2)
727             print('(Info) Objeto liberado en posición deseada!')
728             print('\n')
729
730             print('(Info) Volviendo a posición de reposo...')
731             print('\n')
732             thetai, p = fk(thetai_reset, com, baud)
733             time.sleep(6)
734             print('(Info) En posición de reposo!')
735
736             success = False

```