UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

Repositorio Digital USM **https://repositorio.usm.cl**

Tesis USM TESIS de Pregrado de acceso ABIERTO

2019-12

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

ARAVENA PACHECO, ALVARO JAVIER

https://hdl.handle.net/11673/48999

Repositorio Digital USM, UNIVERSIDAD TECNICA FEDERICO SANTA MARIA

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 coordenados 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 stablishing 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 intrínsecos 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 `size` tiene un valor de 22 milímetros y las intersecciones interiores `nx` y `ny` 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 mílimetros (color blanco) con los puntos de imagen (color rojo y verde) (u_{yx}, v_{yx}) en pixeles ([px]). 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 `inRange()`. 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 coordenado 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:

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

siendo cada transformación T_i, lo que sigue:

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

$$T_2 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.3}$$

$$T_3 = \begin{bmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.4}$$

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

luego, la ecuación 1.1 queda expresada como:

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

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 exe x_i. Es variable en articulaciones prismáticas.

- a_i: para el caso de articualciones 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:

$$T = {}^{0}A_1 \, {}^{1}A_2 \, \cdots \, {}^{n-1}A_n \tag{1.7}$$

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:

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

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:

$$X = \frac{u_p - c_u}{f} \cdot Z \tag{1.9}$$

$$Y = \frac{v_p - c_v}{f} \cdot Z \tag{1.10}$$

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]

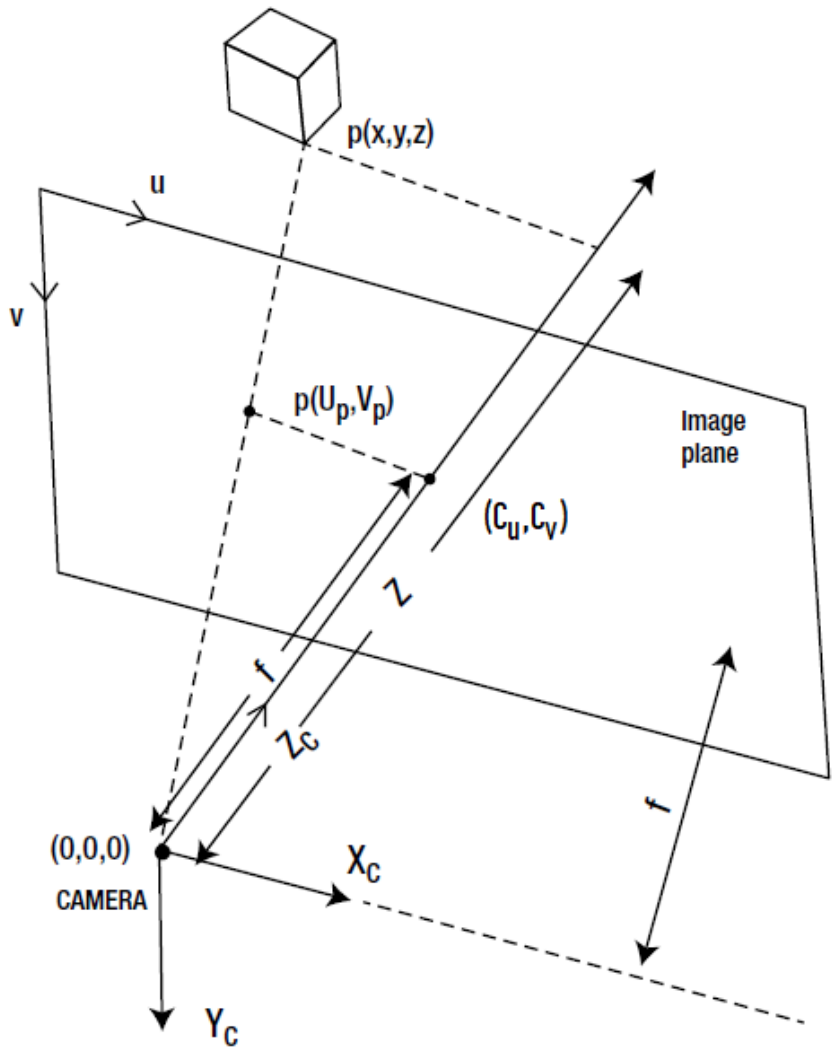


Figura 1.1: Sistema de referencia y variables relevantes del modelo de cámara estenopeica. [4]

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]

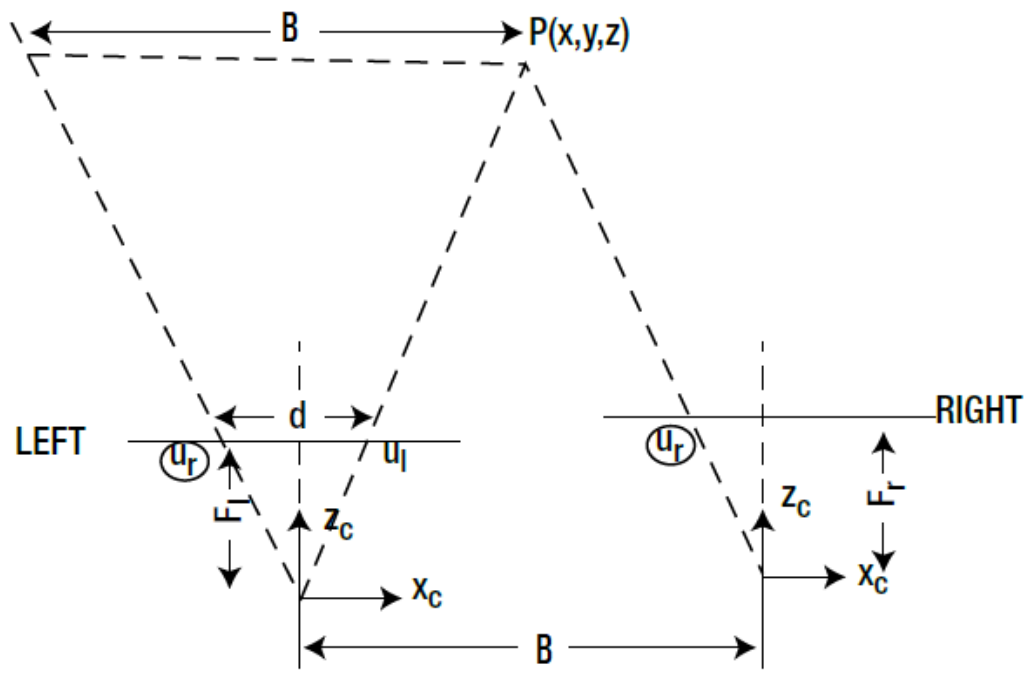


Figura 1.2: Cámara estéreo, sistemas de referencia y parámetros relevantes. [5]

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

$$d = u_l - u_r \tag{1.11}$$

$$Z = B \cdot \frac{f}{d} \tag{1.12}$$

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

OpenCV[1] 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, de 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]

[1] OpenCV, 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]

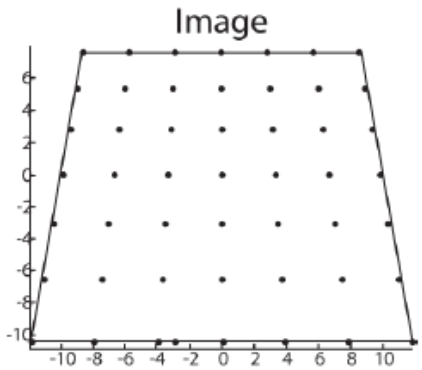


Figura 1.5: Distorsión tangencial. [9]

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.

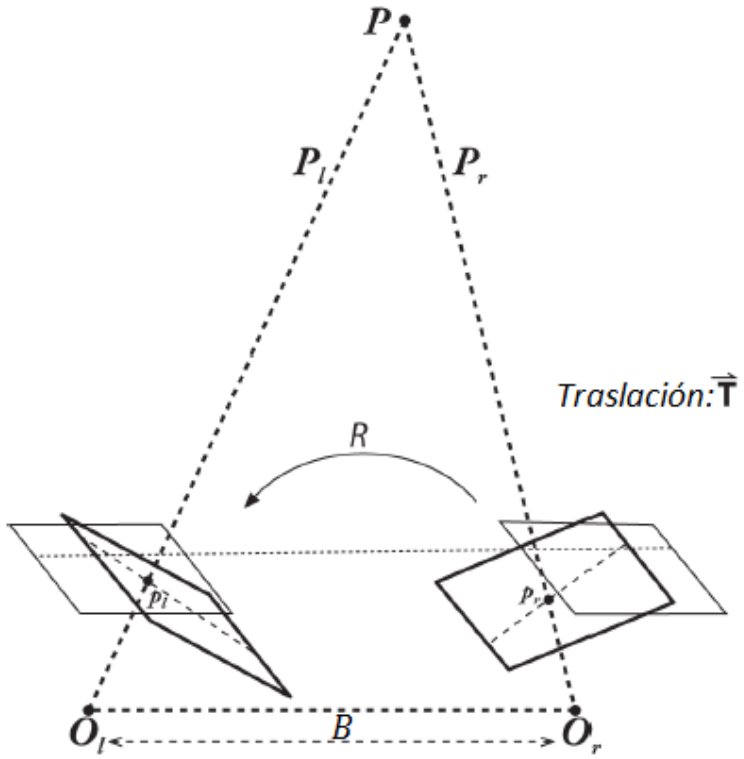


Figura 1.6: Planos de imagen de las cámaras luego de la rectificación estéreo. [15]

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 distintivas [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]

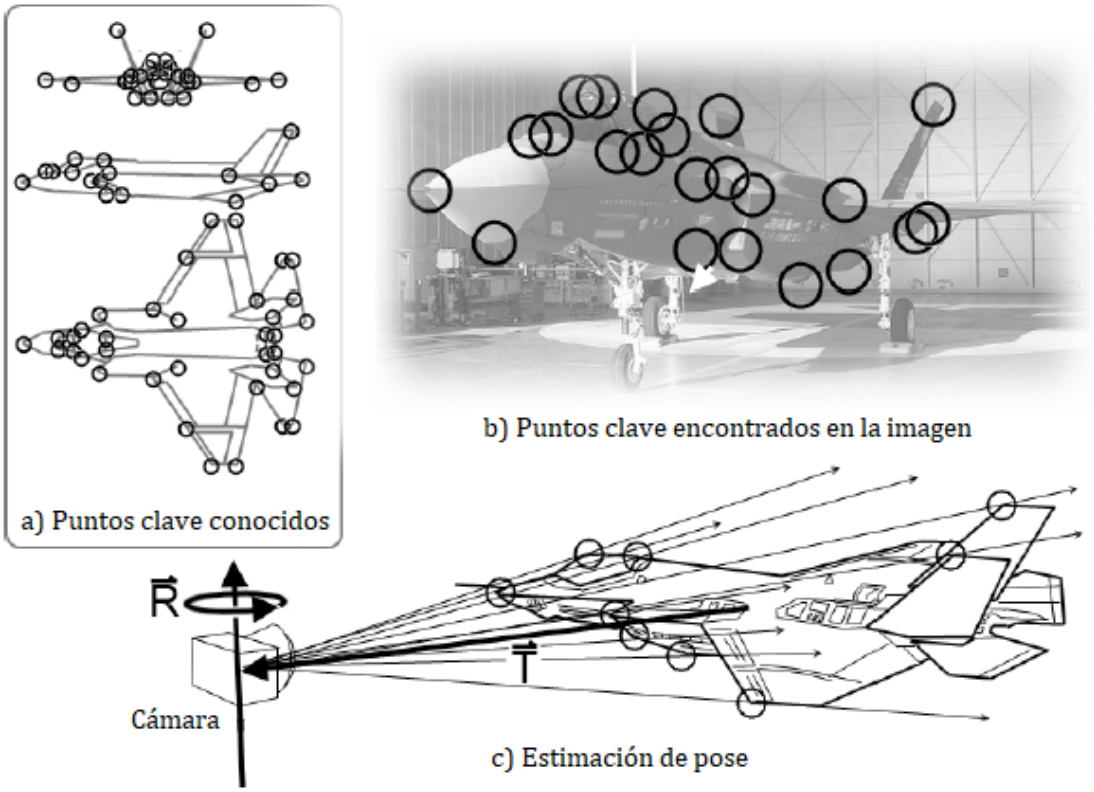


Figura 1.7: Dados los puntos clave (encerrados en los círculos) del objeto (a), se logran identificar cierta cantidad en la imagen (b) y a través de la restricción a la que están sometidos, se estima la pose del objeto relativa a la cámara (c). [20]

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.

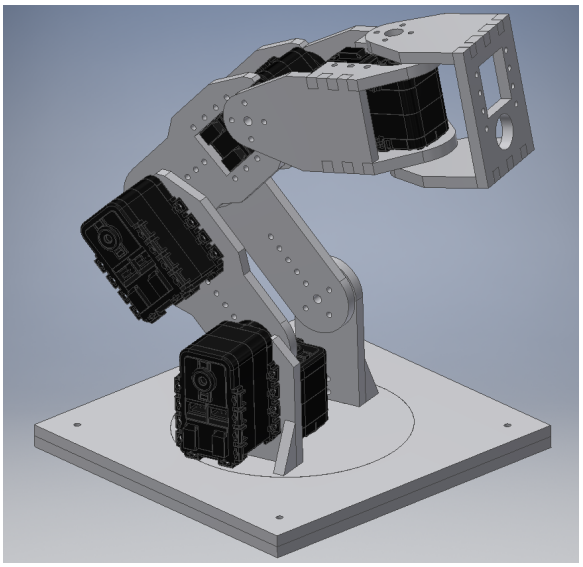


Figura 2.1: Imagen del modelo 3D de Inventor del brazo robótico. Elaboración propia.

Figura 2.2: Imagen del brazo robótico. Elaboración propia.

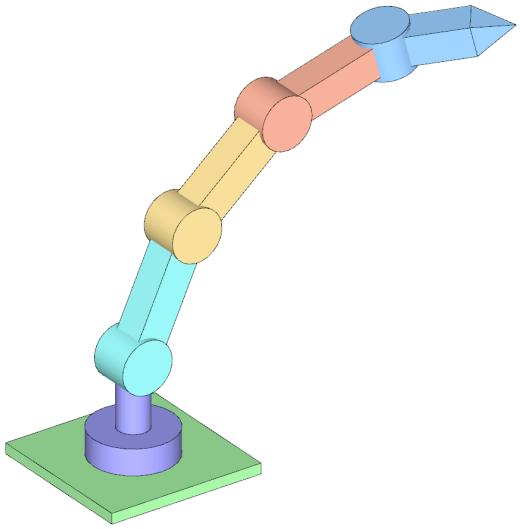


Figura 2.3: Versión simplificada del brazo robótico. Cada eslabón se encuentra destacado con un color distintivo. Elaboración propia.

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

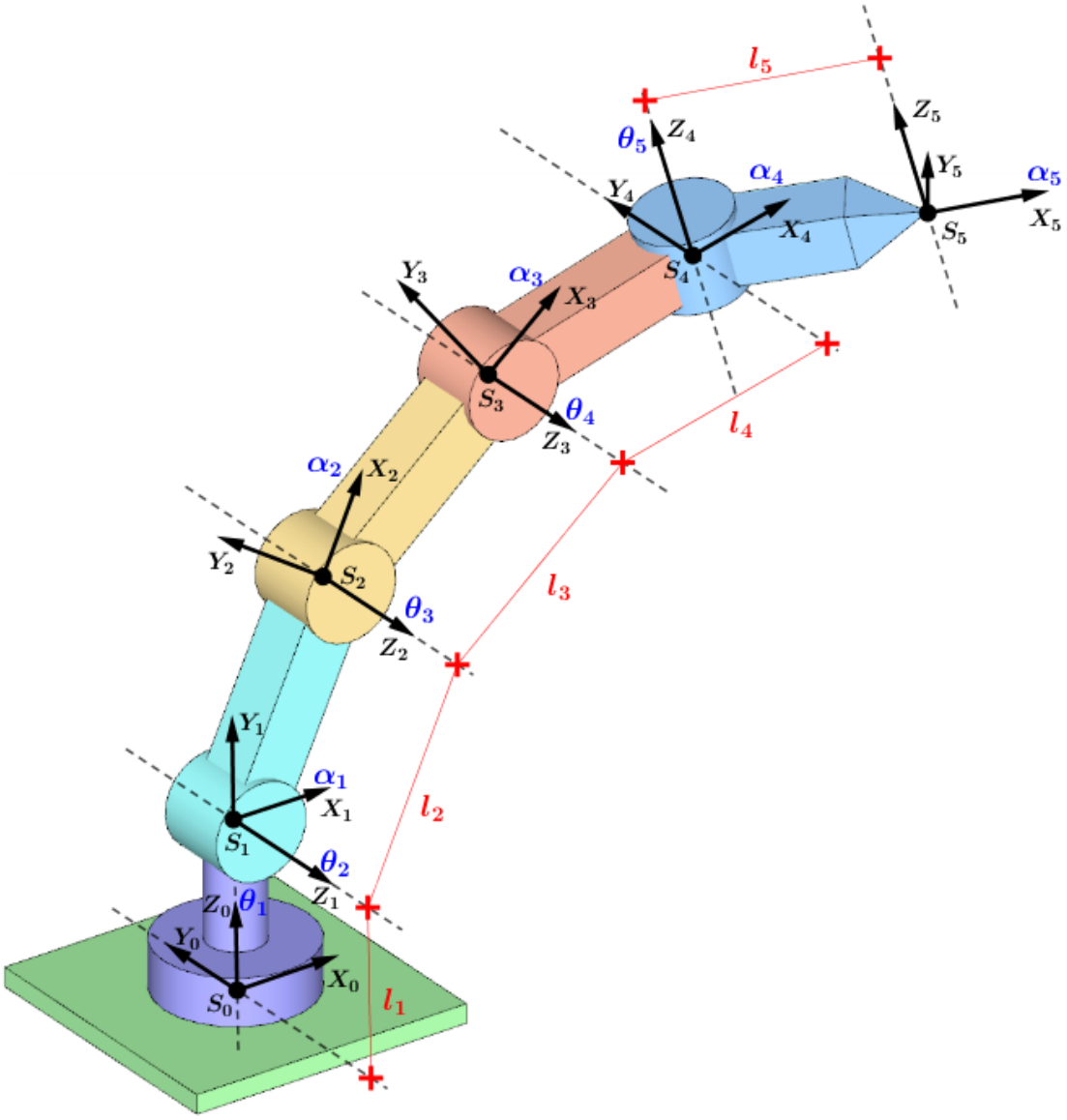


Figura 2.4: Cadena cinemática del brazo robótico según la metodología de Denavit-Hartenberg. Elaboración propia.

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 númerico. Esto es debido a que son las variables con 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.

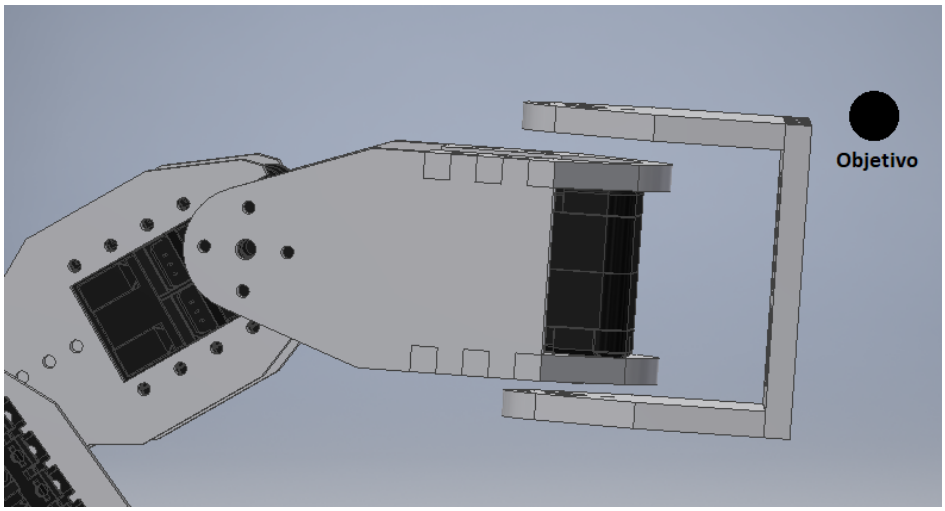


Figura 2.5: Posición alcanzada antes de la corrección del parámetro d_5. Elaboración propia.

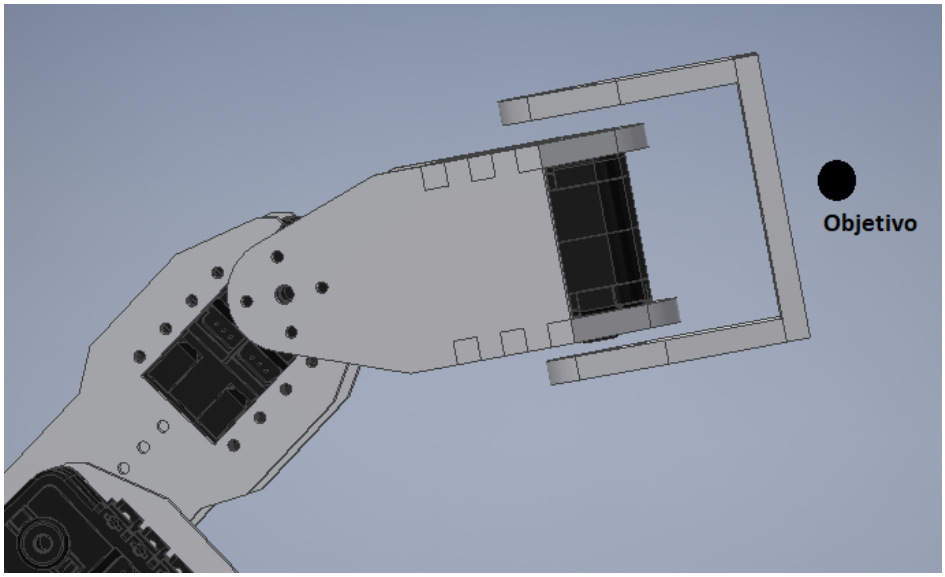


Figura 2.6: Posición alcanzada después de la corrección del parámetro d_5. Elaboración propia.

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.

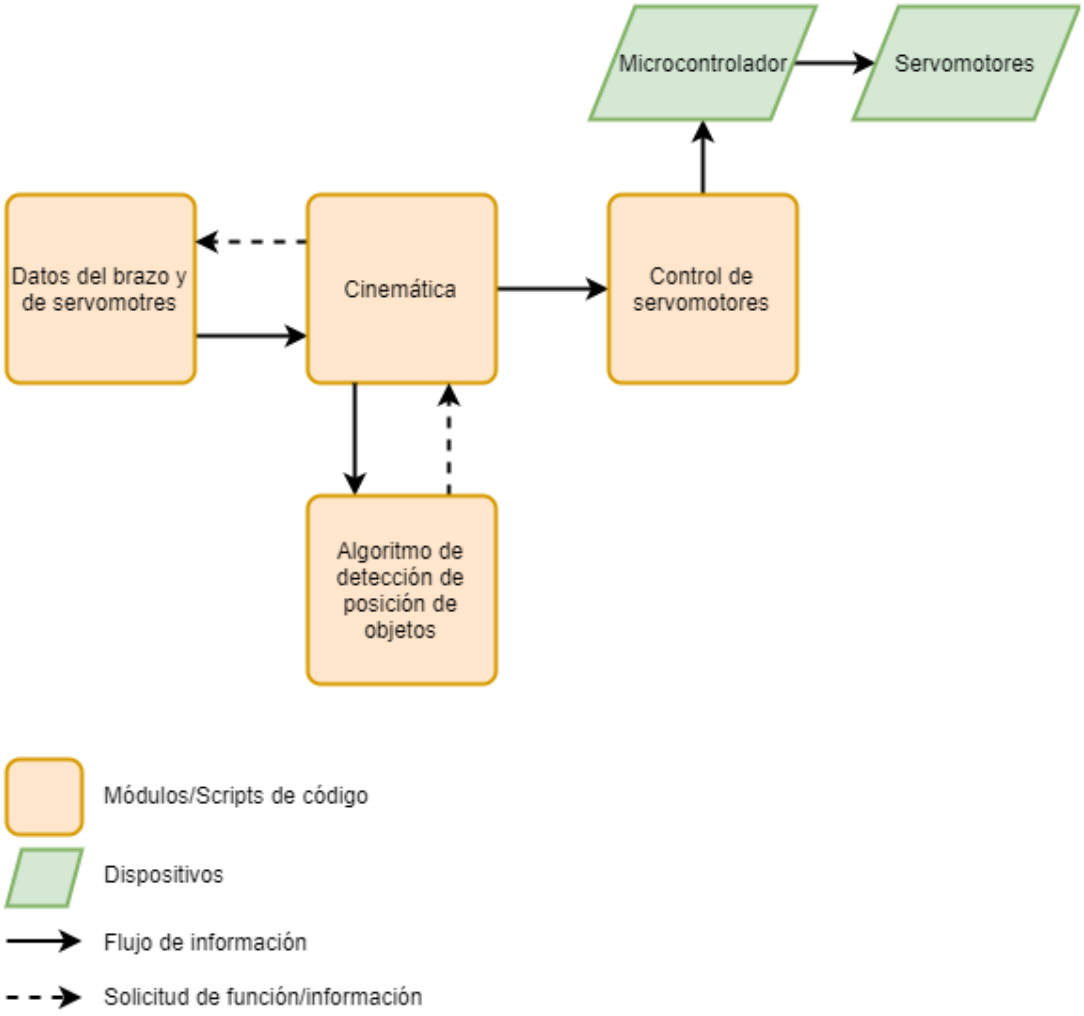


Figura 2.7: Diagrama de flujo de algoritmo de movimiento. Elaboración propia.

2.2.1. Datos del brazo robótico

Este módulo (ver Anexo A.1) escrito en Python[1], y que hace uso de la librería NumPy[2], 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 `[`θ_1`, `θ_2`, `θ_3`, `θ_4`, `θ_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 cado 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 mílimetros 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`.

[1]Python, página web - https://www.python.org

[2]NumPy, 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-12A[3]):

- $u_{i,0}$: 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:

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

Las unidades de rotación de los servomotores se denotarán por "u".

[3]Dynamixel 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, PySerial[4] (para comunicación con el puerto serial COM al que se encuentre conectado el microcontrolador) e IKPy[5] (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

[4]PySerial, página web - https://pythonhosted.org/pyserial/

[5]IKPy, 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).
- `thetai_ini`: lista con los ángulos (flotantes) de la posición actual del brazo en grados en formato `[`θ_1`, `θ_2`, `θ_3`, `θ_4`, `θ_5`]`.
- `segm`: variable tipo flotante. Magnitud de los segmentos que dividirán la trayectoria en milímietros, explicado más abajo.
- `tol`: variable tipo flotante. Límite superior de la diferencia entre el punto alcanzado `p` y el punto objetivo `p_obj` en mílimetros para el que el movmiento sea declarado como exitoso. Esta variable tiene como rol discriminar principalmente entre aquellos movimientos que se dan bajo puntos objetivos díficiles 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.
- `thetai`: 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 `thetai_ini`.

- `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.

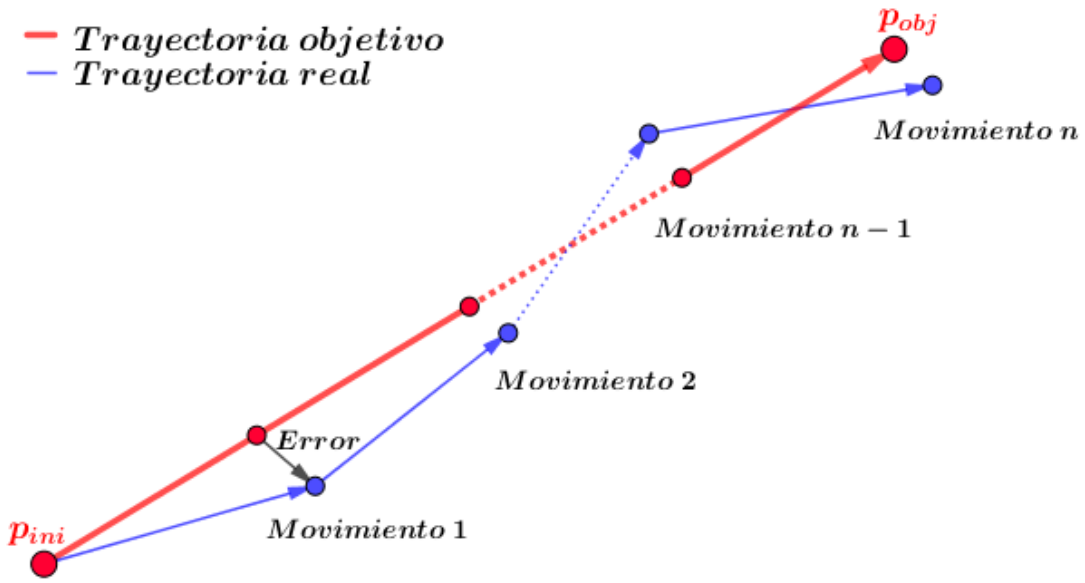


Figura 2.8: Trayectoria y movimiento del brazo generado por la función `ik()`. Elaboración propia.

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 comunicarlas 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 `[`θ_1`,` θ_2`,` θ_3`,` θ_4`,` θ_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 Arduino[6] y utiliza la librería Dynamixel Workbench[7] para el control de los servomotores Dynamixel AX-12A a través del microcontrolador OpenCM 9.04[8].

La función de este código es inicilizar 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.

[6] Arduino, página web - https://www.arduino.cc/

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

[8] OpenCM 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.

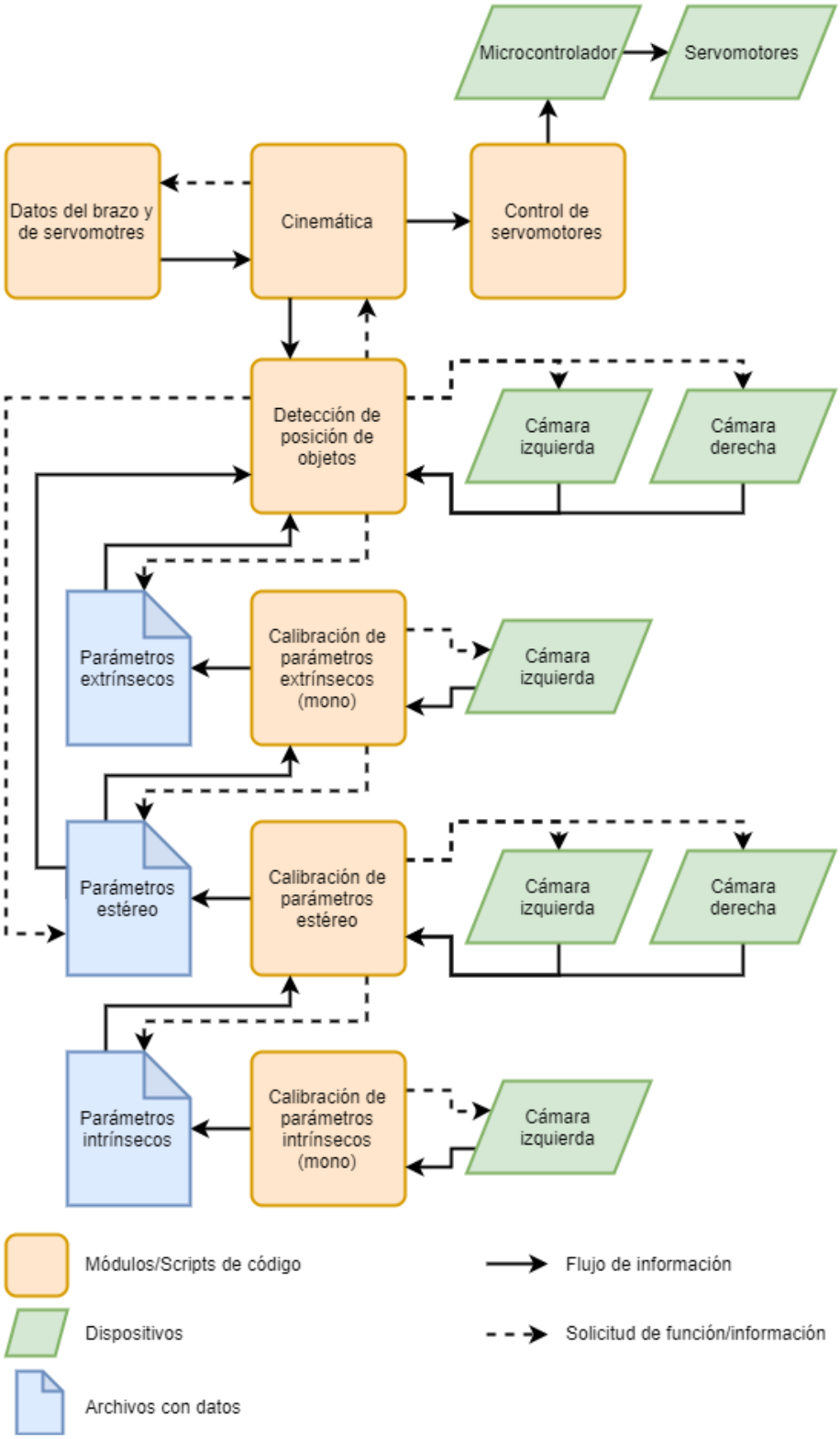


Figura 3.1: Diagrama de flujo del algoritmo completo (calibraciones, detección de posición de objetos y movimiento del brazo). Para las calibraciones *"mono"* ambas cámaras son iguales, consideración explicada más adelante. Elaboración propia.

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 3x3 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 C920[1]. 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 Capture[2]. **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 mílimetros para mantener la consistencia con los valores de los parámetros de Denavit-Hartenberg presentados en la Tabla 2.2 de la Sección 2.1.

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

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

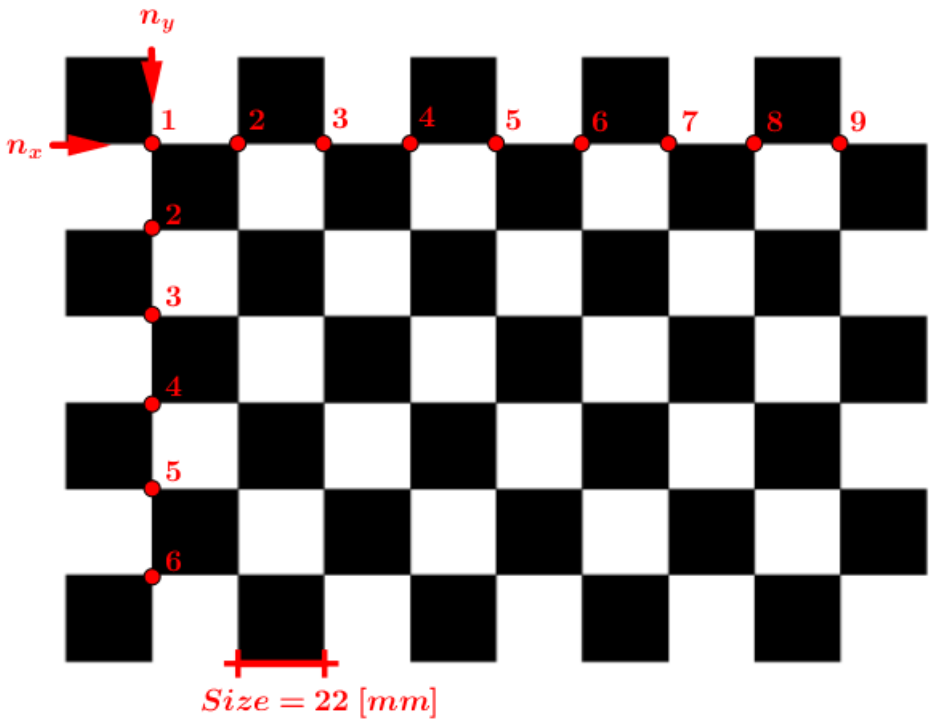


Figura 3.2: Tablero de ajedrez como patrón de calibración y sus parámetros relevantes. En este caso particular, el tamaño de los cuadros `size` tiene un valor de 22 milímetros y las intersecciones interiores `nx` y `ny` corresponden a 9 y 6 respectivamente. [29]

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).

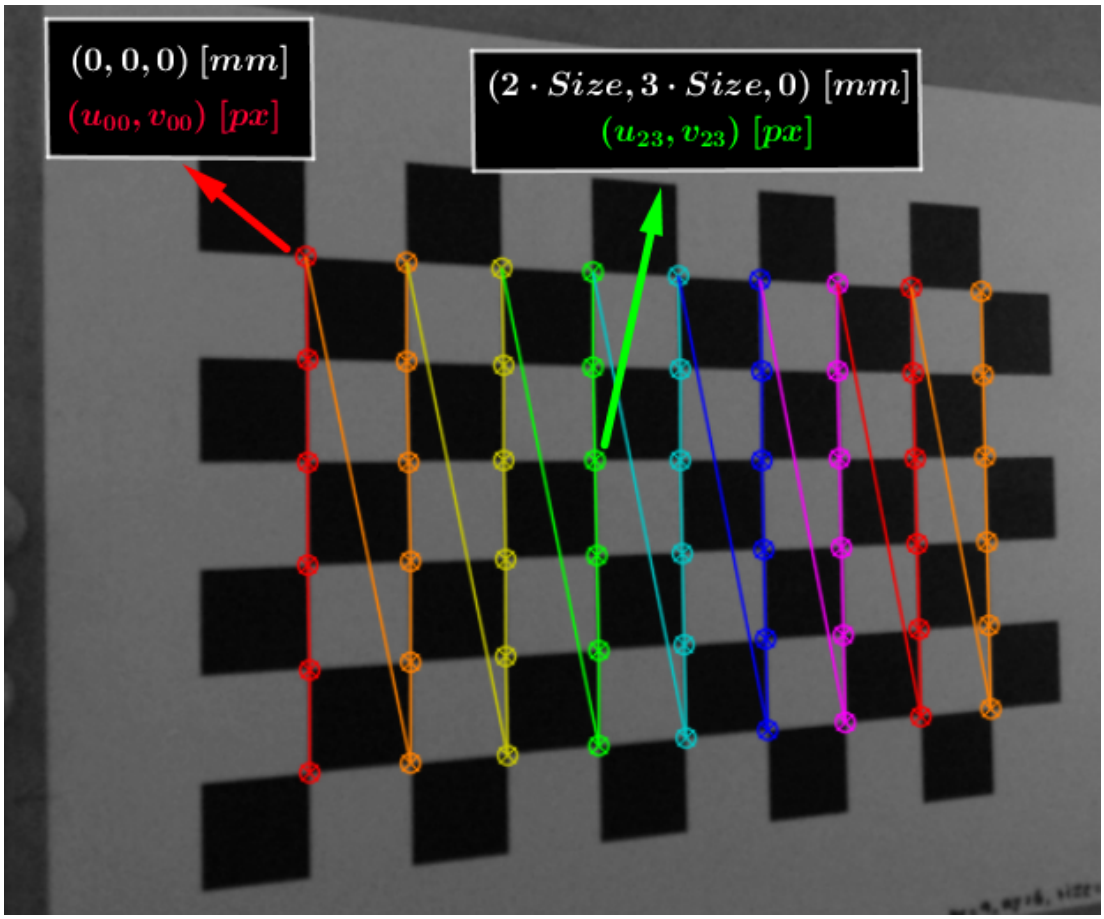


Figura 3.3: Identificación de las intersecciones interiores del patrón de calibración (puntos de colores) y asociación de los puntos de objeto en mílimetros (color blanco) con los puntos de imagen (color rojo y verde) (u_{yx}, v_{yx}) en pixeles ([px]). Elaboración propia.

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 pixeles (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, OS[3] (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).

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

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

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

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 varables `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 `initUndistorRectifyMap()` (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 `steroCalibrate()` ha sido buena, se calcula el error RMS a través de la matriz fundamental F (variable `F` de la línea 333) según la ecuación:

$$[p_r|1]^T \cdot F \cdot [p_l|1] = 0 \tag{3.2}$$

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 ArUco[4] (`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.

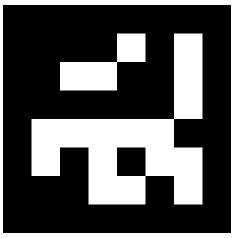


Figura 3.4: Marcador ArUco de 6x6 (cuadros internos). [29]

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:

$$^{Y}HT_{l,rect} = {}^{Y}HT_{C} \cdot {}^{C}HT_{l,rect} \tag{3.3}$$

donde $^{Y}HT_{l,rect}$ 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, $^{Y}HT_{C}$ es la que lleva las

[4]OpenCV, 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.

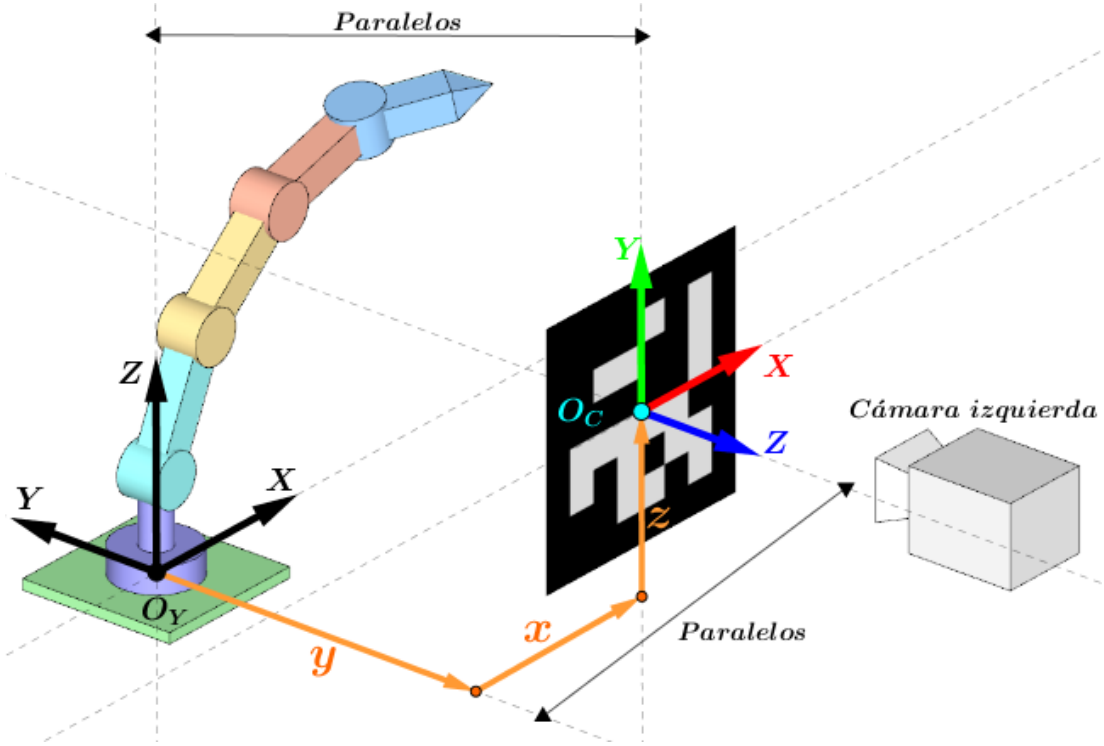


Figura 3.5: Sistemas de referencia y disposición física del brazo robótico y del marcador ArUco para la calibración. Las variables x, y y z en color naranja representan la posición, en las respectivas direcciones, del sistema O_C respecto de O_Y. Elaboración propia.

Para encontrar la matriz de transofmración 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 pixeles 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 pixeles, 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`.

[5]Esta 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.

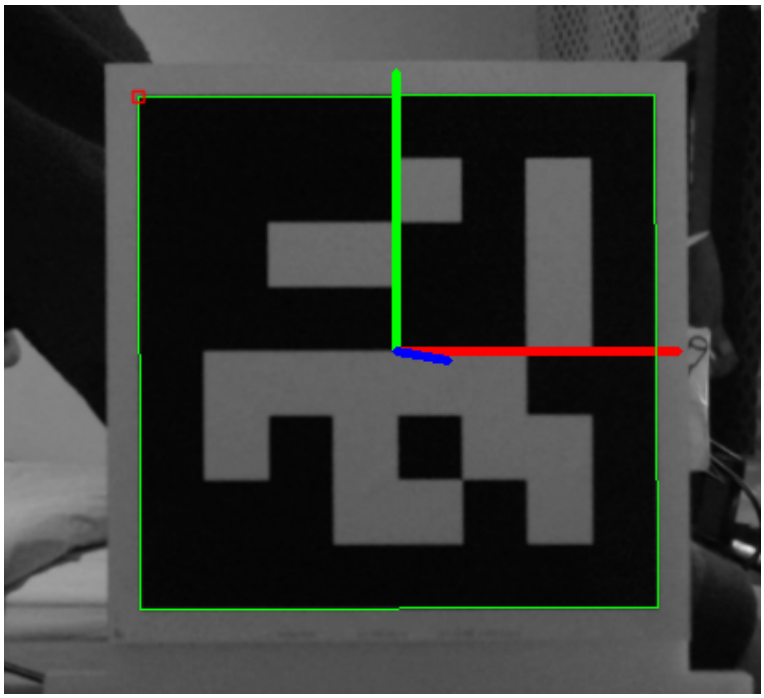


Figura 3.6: Estimación de pose de marcador ArUco 6x6. Los segmentos que se encuentran en el centro del marcador representan a sus ejes coordenados (rojo para el eje X, verde para el eje Y y azul para el eje Z) y el borde verde que encierra al marcador representa su reconocimiento en la imagen. Elaboración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

Para obtener la matriz ${}^{Y}HT_{C}$, 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 ${}^{Y}HT_{C}$ (variable `HT_Oc_Oy`) (líneas 164 a 174).

Con las matrices de transformación ${}^{Y}HT_{C}$ y ${}^{C}HT_{l,rect}$ conocidas, se calcula ${}^{Y}HT_{l,rect}$ (variable `HT_Olrect_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 Time[6] (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 modificiación 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]

[6]Time, 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]

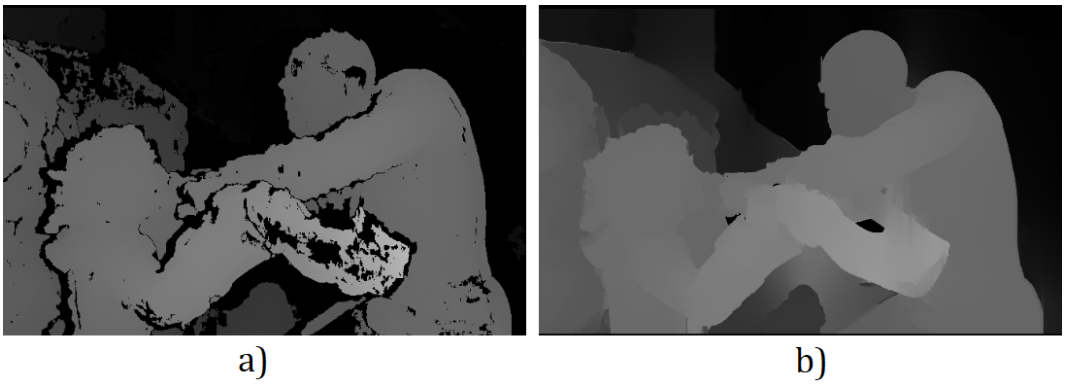


Figura 3.7: (a) Mapa de disparidad sin filtrar. Los elementos más cercanos a la cámara están representado por un color claro y los más lejanos por un color oscuro. (b) Corrección del mapa de disparidad realizada por el Filtro de disparidad WLS. [41]

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].

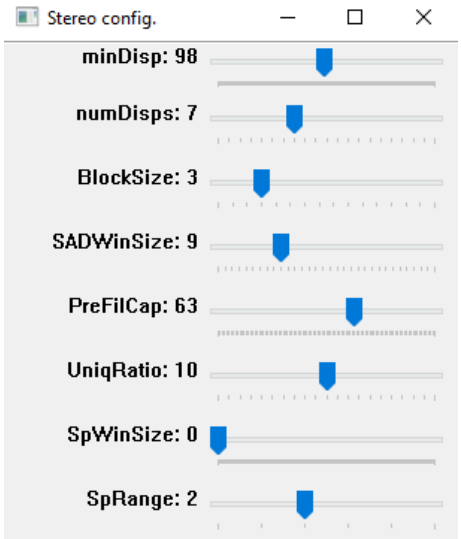


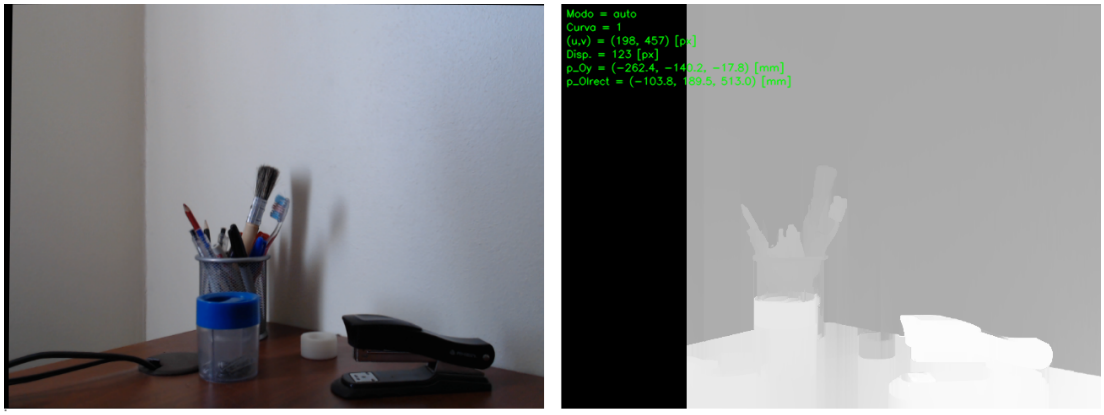
Figura 3.8: Sección del panel de deslizadores para configuración de parámetros del módulo. Elaboración propia.

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 pixeles 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 ${}^{Y}HT_{l,rect}$ (`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).



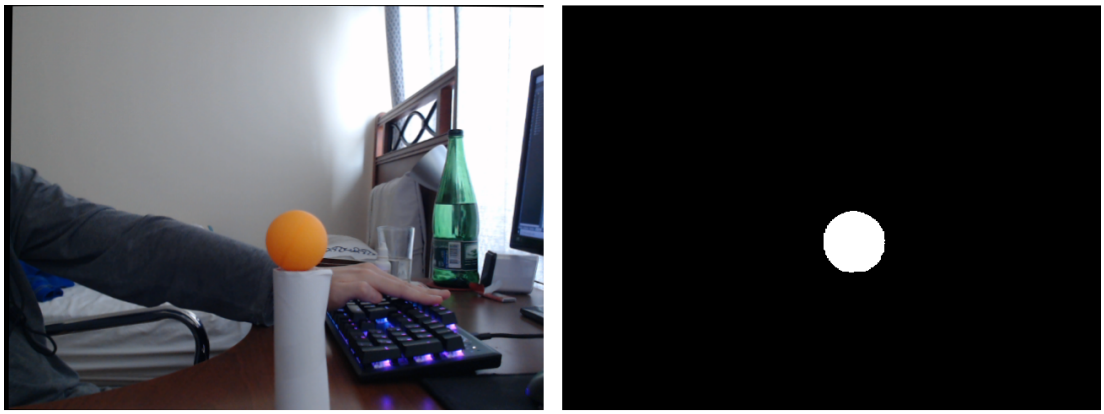
a) b)

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 trasformada, 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 ecuentran dentro de los límites HSV en color blanco y a aquellos que se encuentran fuera, en color negro (ver Figura 3.10).



a) b)

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).

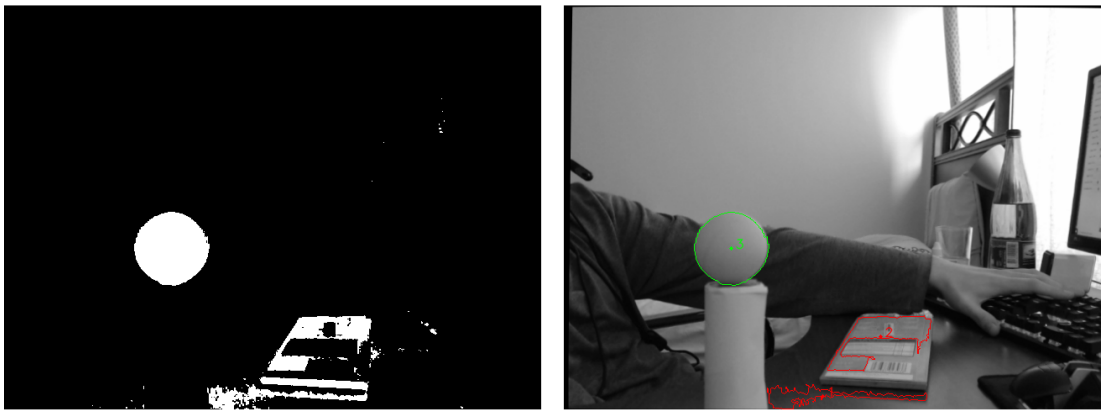
La posibilidad de ajustar los valores de los parámetros `prec_lvl` y `min_sides` tiene la finalidad de poder eliminar, en la medida de lo posible, aquellos contornos correspondientes a regiones no deseadas que hayan sido consideradas como válidas en la imagen binaria debido a los colores presentes en la escena (colores similiares 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 pixeles) 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 pixeles 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 róbotico.

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



a) b)

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:

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

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:

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

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

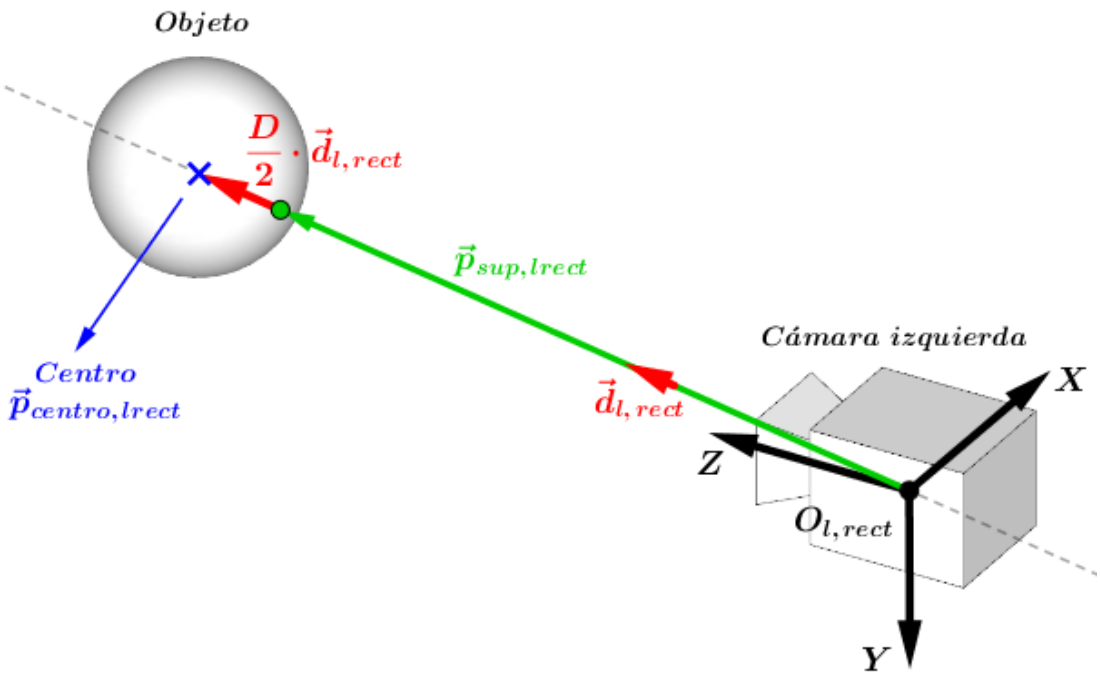


Figura 3.12: Determinación del centro del objeto esférico con respecto del sistema coordenado de la cámara izquierda. Elaboración propia.

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 ${}^{Y}HT_{l,rect}$ (`HT_Olrect_Oy`), como sigue:

$$[\vec{p}_{centro,Y}|1] = {}^{Y}HT_{l,rect} \cdot [\vec{p}_{centro,lrect}|1] \tag{3.6}$$

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):

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

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

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

En la Figura 3.13 se aprecia gráficamente el cálculo realizado.

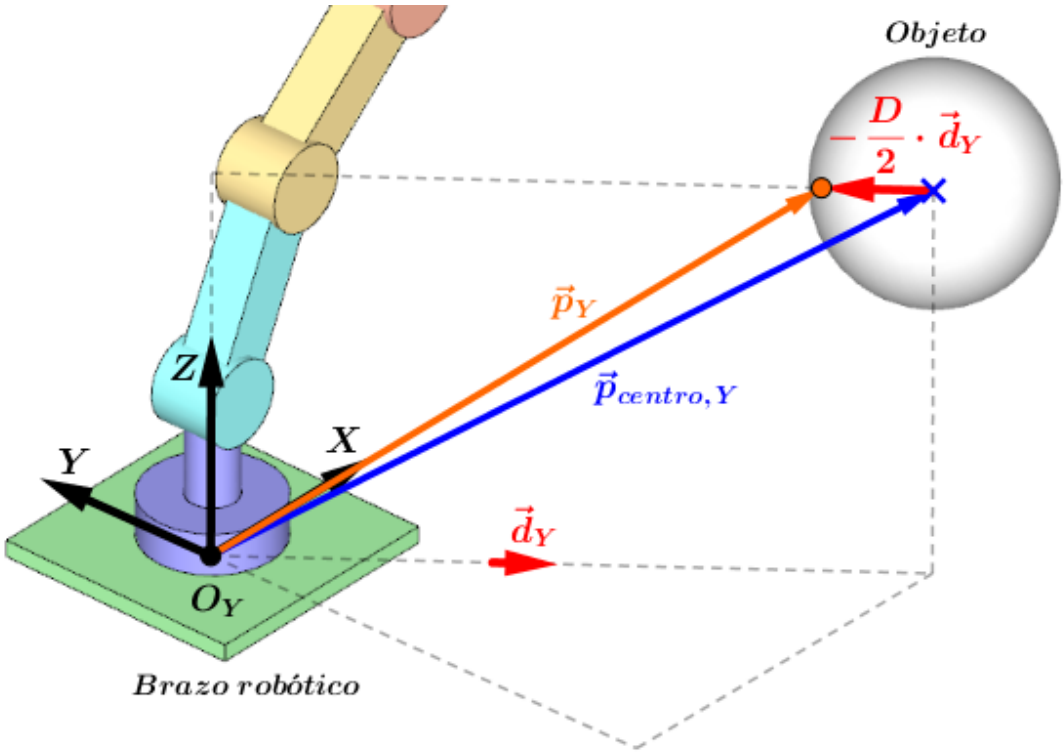


Figura 3.13: Determinación del punto de agarre del objeto esférico respecto del sistema de referencia del brazo robótico. Elaboración propia.

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 pixeles blancos y negros) originada desde la segmentación por colores de la imagen capturada, y haciendo uso de la geometría del modelo de la cámara estenopeica 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 prostética 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 pixeles 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#createstereosgbm> [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

```
#-*- coding: cp1252 -*-

import numpy as np


def par_dh(thetai): #Parámetros de Denavit-Hartenberg de YARBIZ.

    #Distancias di en [mm].
    d1 = 62 #Apoyo a eje servo 2.
    d2 = 0
    d3 = 0
    d4 = 0
    d5 = -30 #Corrección de sistema de referencia extremo.

    #Distancias ai en [mm].
    a1 = 0
    a2 = 70 #Eje servo 2 a eje servo 3.
    a3 = 70 #Eje servo 3 a eje servo 4.
    a4 = 61 #Eje servo 4 a eje servo 5.
    a5 = 43+5 #Eje servo 5 a superficie eslabón extremo.

    #Ángulos alpha en grados.
    alpha1 = 90.0
    alpha2 = 0
    alpha3 = 0
    alpha4 = -90.0
```

```
    alpha5 = 0

    #Ángulos alpha en radianes.
    alpha1 = alpha1/180*np.pi
    alpha2 = alpha2/180*np.pi
    alpha3 = alpha3/180*np.pi
    alpha4 = alpha4/180*np.pi
    alpha5 = alpha5/180*np.pi

    #Ángulos theta en radianes.
    theta1 = float(thetai[0])/180*np.pi
    theta2 = float(thetai[1])/180*np.pi
    theta3 = float(thetai[2])/180*np.pi
    theta4 = float(thetai[3])/180*np.pi
    theta5 = float(thetai[4])/180*np.pi

    dh = [[theta1,d1,a1,alpha1],
          [theta2,d2,a2,alpha2],
          [theta3,d3,a3,alpha3],
          [theta4,d4,a4,alpha4],
          [theta5,d5,a5,alpha5]]


    #Matrices de transformación.
    i = 1
    A01 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
        ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i
        -1][0])],
            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
                ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
                sin(dh[i-1][0])],
            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
            [0,0,0,1]]

    i = 2
    A12 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
        ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i
        -1][0])],
            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
                ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
                sin(dh[i-1][0])],
            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
            [0,0,0,1]]

    i = 3
    A23 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
        ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i
        -1][0])],
            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
                ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
                sin(dh[i-1][0])],
            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
```

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

    i = 4
    A34 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
        ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i
        -1][0])],
            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
                ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
                sin(dh[i-1][0])],
            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
            [0,0,0,1]]

    i = 5
    A45 = [[np.cos(dh[i-1][0]),-np.cos(dh[i-1][3])*np.sin(dh[i-1][0])
        ,np.sin(dh[i-1][3])*np.sin(dh[i-1][0]),dh[i-1][2]*np.cos(dh[i
        -1][0])],
            [np.sin(dh[i-1][0]),np.cos(dh[i-1][3])*np.cos(dh[i-1][0])
                ,-np.sin(dh[i-1][3])*np.cos(dh[i-1][0]),dh[i-1][2]*np.
                sin(dh[i-1][0])],
            [0,np.sin(dh[i-1][3]),np.cos(dh[i-1][3]),dh[i-1][1]],
            [0,0,0,1]]

    A02 = np.dot(A01, A12)
    A03 = np.dot(A02, A23)
    A04 = np.dot(A03, A34)
    A05 = np.dot(A04, A45) #Matriz de transformación final.

    #Coordenadas cartesianas del extremo de YARBIZ.
    x = A05[0,3]
    y = A05[1,3]
    z = A05[2,3]
    p = np.array([[x],
                  [y],
                  [z]])

    return dh, p


def par_fu(): #Parámetros de las funciones ui(thetai) (pfu).

    k = 1/0.29 #Constante de conversión servos en unidades/grado.

    u1_0 = 512 #Unidades u en la que thetai es cero.
    u1_min = 0 #Unidades u mín.
    u1_max = 1023 #Unidades u máx.
    theta1_min = (u1_min - u1_0)/k
    theta1_min = round(theta1_min,2) #Ángulo theta mín.
    theta1_max = (u1_max - u1_0)/k
    theta1_max = round(theta1_max,2) #Ángulo theta máx.

    u2_0 = 202
    u2_min = 133
```

```
    u2_max = 885
    theta2_min = (u2_min - u2_0)/k
    theta2_min = round(theta2_min,2)
    theta2_max = (u2_max - u2_0)/k
    theta2_max = round(theta2_max,2)

    u3_0 = 512
    u3_min = 0
    u3_max = 1023
    theta3_max = (u3_max - u3_0)/k
    theta3_max = round(theta3_max,2)
    theta3_min = (u3_min - u3_0)/k
    theta3_min = round(theta3_min,2)

    u4_0 = 512
    u4_min = 167
    u4_max = 857
    theta4_min = (u4_min - u4_0)/k
    theta4_min = round(theta4_min,2)
    theta4_max = (u4_max - u4_0)/k
    theta4_max = round(theta4_max,2)

    u5_0 = 512
    u5_min = 167
    u5_max = 857
    theta5_min = (u5_min - u5_0)/k
    theta5_min = round(theta5_min,2)
    theta5_max = (u5_max - u5_0)/k
    theta5_max = round(theta5_max,2)

    pfu = [[u1_0,u1_min,u1_max,theta1_min,theta1_max], #Arreglo de
        los pfu.
            [u2_0,u2_min,u2_max,theta2_min,theta2_max],
            [u3_0,u3_min,u3_max,theta3_min,theta3_max],
            [u4_0,u4_min,u4_max,theta4_min,theta4_max],
            [u5_0,u5_min,u5_max,theta5_min,theta5_max],
            [k]]

    return pfu
```

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

```
#-*- coding: cp1252 -*-

import numpy as np
import serial
import ikpy
from ikpy.chain import Chain
from ikpy.link import OriginLink, URDFLink
from yarbiz_data import par_dh
from yarbiz_data import par_fu


def ik(p_obj, p_ini, thetai_ini, segm, tol, com, baud): #Función de
    Cinemática Inversa.

    print('\n')

    dh,_ = par_dh(thetai_ini)

    pfu = par_fu()

    #Creación del brazo - librería ikpy.
    yarbiz = Chain(name='YARBIZ', links=[
        OriginLink(),
        URDFLink(
            name="Servo1",
            translation_vector=[0,0,0],
            orientation=[0,0,0],
            rotation=[0,0,1],
        ),
        URDFLink(
            name="Servo2",
            translation_vector=[dh[0][2],0,dh[0][1]],
            orientation=[dh[0][3],0,0],
            rotation=[0,0,1],
        ),
         URDFLink(
            name="Servo3",
            translation_vector=[dh[1][2],0,dh[1][1]],
            orientation=[dh[1][3],0,0],
            rotation=[0,0,1],
        ),
        URDFLink(
            name="Servo4",
            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
         de yarbiz.
63
64    dist = np.sqrt([v[0,0]**2 + v[1,0]**2 + v[2,0]**2])
65    dist = dist[0] #Magnitud vector p_obj - p_ini.
66
67    d = (1/dist)*v #Vector director unitario.
68
69    #Cantidad de segmentos.
70    if segm == 0:
71
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
         para los servos.
85
86    for j in range(1,n+1): #Tantos movimientos como segmentos
         calculados.
87
88        if j == n:
89
90            p_j = p_obj
91
92        else:
```

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

```
            u = pfu[i-1][0] + theta*pfu[5][0] #Calcula unidades servo u a partir de thetai.
            u = int(round(u,0)) #Aproximar a unidad más cercana.


            if u < pfu[i-1][1]: #Si u está bajo el mínimo...

                u = pfu[i-1][1]

            elif u > pfu[i-1][2]: #Si u está sobre el máximo...

                u = pfu[i-1][2]


            if i == 5:

                instr = instr + str(u)

                instrs.append(instr)

            else:

                instr = instr + str(u) + ',’


            #Actualizar posición actual para el siguiente movimiento.

            theta = (u - pfu[i-1][0])/pfu[5][0] #theta para el servo i con precisión de los servos (0.29[°/u]).
            thetai[i-1] = round(theta,2)



        print('Instrucción = ' + instr)
        print('\n')


        if j == n: #Si es el último movimiento, calcular p.

            _,p = par_dh(thetai) #Punto alcanzado con thetai.


            diff = p - p_obj #Vector diferencia entre resultado y objetivo.
            delta = np.sqrt([diff[0,0]**2 + diff[1,0]**2 + diff[2,0]**2])
            delta = delta[0] #Módulo del vector diferencia.

            if delta <= tol:

                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
                    el puerto serial ' + com + '!')
217             print('\n')
218
219
220     print('')
221     print('Theta_i = ' + str(thetai) + ' [°]')
222     print('p = (' + str(round(p[0,0],1)) + ', ' + str(round(p[1,0],1)
            ) + ', ' + str(round(p[2,0],1)) + ') [mm]')
223     print('Success: ', success)
224     print('\n')
225     print('\n')
226
227     return success, thetai, p
228
229
230
231
232 def fk(thetai_obj, com, baud): #Cinemática Directa.
233
234     print('\n')
235
```

```
    pfu = par_fu()

    range_error = False

    thetai = [] #Arreglo para ángulos según precisión de servos (0.29[°/u])

    #Generación de la instrucción para servos.
    instr = ''

    for i in range(1,6):

        u = pfu[i-1][0] + float(thetai_obj[i-1])*pfu[5][0] #Calcula unidades servo u a partir de thetai.
        u = int(round(u,0)) #Aproximar a unidad más cercana.

        theta = (u - pfu[i-1][0])/pfu[5][0] #theta para el servo i con precisión de los servos (0.29°/u).
        thetai.append(round(theta,2))


        if u >= pfu[i-1][1] and u <= pfu[i-1][2]:

            if i == 5:

                instr = instr + str(u)

            else:

                instr = instr + str(u) + ','

        else:

            range_error = True

            break



    if range_error == False:

        print('Instrucción = ' + instr)
        print('')


        _,p = par_dh(thetai) #Punto alcanzado con thetai.


        try:

            ser = serial.Serial(com,baud)

```

```
            ser.write(instr.encode())

            ser.read(1)

        except:

            print('')
            print('(Error) Ha habido problemas para comunicarse con el puerto serial ' + com + '!')
            print('\n')


        print('Theta_i = ' + str(thetai) + ' [°]')
        print('p = (' + str(round(p[0,0],1)) + ', ' + str(round(p[1,0],1)) + ', ' + str(round(p[2,0],1)) + ') [mm]')
        print('\n')
        print('\n')

    else:

        print('')
        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]) + '] [°]!')
        print('\n')

        thetai = []
        p = []


    return thetai, p
```

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

```
#include <DynamixelWorkbench.h>

#if defined(__OPENCM904__)
  #define DEVICE_NAME "1" //Dynamixel on Serial3(USART3)  <-OpenCM
      485EXP
#elif defined(__OPENCR__)
  #define DEVICE_NAME ""
#endif

#define BAUDRATE  1000000

//Se declaran las variables.
String instr;

String us;
int len;
int32_t u;

uint8_t dxl_1 = 1;
uint8_t dxl_2 = 2;
uint8_t dxl_3 = 3;
uint8_t dxl_4 = 4;
uint8_t dxl_5 = 5;

uint8_t dxl_id;

int32_t dxl_vel = 40;

DynamixelWorkbench dxl_wb;

void setup() {

  Serial.begin(115200); //Incializar puerto serial.

  const char *log;
  bool result = false;

  //Se inicializan los servomotores.
  dxl_wb.init(DEVICE_NAME, BAUDRATE, &log);

  dxl_wb.ping(dxl_1, &log);
  dxl_wb.ping(dxl_2, &log);
  dxl_wb.ping(dxl_3, &log);
  dxl_wb.ping(dxl_4, &log);
  dxl_wb.ping(dxl_5, &log);

```

```
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); //Envíar 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

```
#-*- coding: cp1252 -*-

import numpy as np
import cv2 as cv
import glob


def generate_vimg_name(cam_name, i): #Función para generar el nombre
    (str) de imágenes válidas (patrón encontrado).

    if i >= 100:

        vimg_name = cam_name+'_0'+str(i)+'.jpg'

    elif i >= 10:

        vimg_name = cam_name+'_00'+str(i)+'.jpg'

    else:

        vimg_name = cam_name+'_000'+str(i)+'.jpg'

    return vimg_name


#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,
               presionar [key]:')
94          print('')
95          print('            [y] Sí')
96          print('            [n] No')
97          print('\n')
98
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
               dispositivo!')
106         print('\n')
107
108     while flag1:
109
110         ret, img = cam.read() #Lee frame.
111
112         if ret:
113
114             cv.imshow(win_name + 'BGR', img)
115
116         key = cv.waitKey(33)
117
118         if key == ord('y'):
119
120             flag1 = False
121             flag0 = False
122
123             cam.release()
124             cam.release()
125             cv.destroyAllWindows()
126
127         elif key == ord('n'):
```

```
            flag1 = False

            cam.release()
            cam.release()
            cv.destroyAllWindows()



#Información del patrón de calibración.

print('Datos sobre el patrón de calibración:')
print('')

size = float(input('    >> Tamaño de lado de los cuadros en [mm] = ')
    )
print('')

nx = int(input('    >> Nro. de intersecciones horizontales = '))
ny = int(input('    >> Nro. de intersecciones verticales   = '))

print('\n')


#Selección de fuente de datos para la calibración.
print('Modalidad de calibración:')
print('')

flag = True

while flag:

    print('    [1] Calibración desde imagenes en tiempo real')
    print('    [2] Calibración desde imágenes pre-tomadas')
    print('')

    o = input('    >> Opción: ')

    print('\n')

    if o == '1' or o == '2':

        flag = False

    else:

        print('    (Error) Opción inválida!')
        print('\n')


cont = 0 #Nro. de iteraciones.
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)
                        , None) #Encontrar las esquinas del tablero de
                        ajedrez.
228
229                 if ret == True: #Si encuentra las esquinas...
230
231                     i += 1 #Incrementa el número de imágenes válidas.
232                     print('(Info) Válidas = '+str(i))
233
234                     objpts.append(ptnpts) #Agrega puntos de objeto (3
                            D).
235
236                     corners2 = cv.cornerSubPix(gray, corners, (11,11)
                            , (-1,-1), criteria) #Refina esquinas.
237                     imgpts.append(corners2) #Agrega puntos refinados
                            de imagen (2D, pixeles)
238
239                     #Dibujar y mostrar esquinas del tablero.
240                     gray_bgr = cv.cvtColor(gray, cv.COLOR_GRAY2BGR)
241                     cv.drawChessboardCorners(gray_bgr, (ny,nx),
                            corners2, ret)
242                     cv.imshow(cam_name+'_v-img', gray_bgr)
243                     cv.waitKey(33)
244
245                     #Guardar imagen válida.
246                     vimg_name = generate_vimg_name(cam_name, i)
247                     cv.imwrite(rel_dir + vimg_name, gray)
248
249
250
251     cam.release()
252     cv.destroyAllWindows()
253     print('\n')
254
255
256 elif o == '2': #Validación y recolección de información necesaria
        para fase de calibración desde imágenes pre-tomadas.
257
258     print('Procesar imágenes:')
259     print('\n')
260
261     images = glob.glob(rel_dir+'*.jpg')
262
263     for fname in images:
264
265         cont += 1
266
267         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)
            #Encontrar las esquinas del tablero de ajedrez.
278
279         if ret == True: #Si encuentra las esquinas...
280
281             i += 1
282             print('(Info) Válidas = '+str(i))
283
284             objpts.append(ptnpts) #Agrega los puntos de objeto (3D)
285
286             corners2 = cv.cornerSubPix(gray, corners, (11,11),
                (-1,-1), criteria) #Refina las esquinas.
287             imgpts.append(corners2) #Agrega los puntos de imagen (2D)
288
289             #Dibujar y mostrar esquinas del tablero.
290             gray_bgr = cv.cvtColor(gray, cv.COLOR_GRAY2BGR)
291             cv.drawChessboardCorners(gray_bgr, (ny,nx), corners2, ret
                )
292             cv.imshow(cam_name+'_v-img', gray_bgr)
293             cv.waitKey(100)
294
295             #Guardar imagen válida.
296             vimg_name = generate_vimg_name(cam_name, i)
297             cv.imwrite(rel_dir + vimg_name, gray)
298
299
300         else:
301
302             print('(Info) Válidas = '+str(i))
303
304
305     cv.destroyAllWindows()
306     print('\n')
307
308
309 #Obtener parámetros intrínsecos.
310 reproy_err_rms, imtx, distc, rvecs, tvecs = cv.calibrateCamera(objpts
    , imgpts, gray.shape[::-1], None, None)
311
312 if reproy_err_rms <= 0.5: #Valor límite del error RMS de re-proyecció
    n para una buena calibracion.
313
314     print('(Info) Calibración exitosa! (Error RMS de re-proyección [
```

```
        px] = '+str(reproy_err_rms)+' E[0, 0.5])')
    print('\n')

    np.savez(rel_dir + 'mono_intr', imtx=imtx, distc=distc)

    print('(Info) Archivo .npz con matriz de parámetros intrínsecos y coeficientes de distorsión generado!')
    print('\n')

    npzfile = np.load(rel_dir + 'mono_intr.npz')

    print('imtx = ')
    print(npzfile['imtx'])
    print('')

    print('distc = ')
    print(npzfile['distc'])
    print('\n')


    print('Probar calibración:')
    print('')

    flag = True

    while flag:

        print('    [1] Probar calibración')
        print('    [0] Terminar')
        print('')

        o = input('    >> Opción: ')
        print('\n')

        if o == '1':

            imgfile = input('    >> Nombre de imagen a corregir: ')
            print('')

            img = cv.imread(rel_dir + imgfile + '.jpg')

            #Eliminar distorsión.
            h,  w = img.shape[:2]
            newcameramtx, roi = cv.getOptimalNewCameraMatrix(imtx, distc, (w,h), 1, (w,h))
            undst_img = cv.undistort(img, imtx, distc, None, newcameramtx)
            x, y, w, h = roi
            undst_img = undst_img[y:y+h, x:x+w]

            cv.imwrite(rel_dir + imgfile + '_calibtest.png', 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

```
#-*- coding: cp1252 -*-

import numpy as np
import cv2 as cv
import os


def generate_vimg_name(cam_name, i): #Función para generar el nombre
    (str) de imágenes válidas (patrón encontrado).

    if i >= 100:

        vimg_name = cam_name+'_0'+str(i)+'.jpg'

    elif i >= 10:

        vimg_name = cam_name+'_00'+str(i)+'.jpg'

    else:

        vimg_name = cam_name+'_000'+str(i)+'.jpg'

    return vimg_name



#Menú para calibración.
print('')

print('Determinación de números de dispositivo de cámaras:')
print('')

flag0 = True
flag1 = False

while flag0:

    print('    Números de dispositivo:')
    print('')
    l_cam_num = int(input('         >> Cámara izq.: '))
    r_cam_num = int(input('         >> Cámara der.: '))
    print('\n')

    print('    (Info) Intentando inicializar cámaras...')

    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 cárama 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
            , presionar [key]:')
60          print('')
61          print('             [y] Sí')
62          print('             [n] No')
63          print('\n')
64
65          flag1 = True
66
67      else:
68
69          print('\n')
70          print('     (Error) No se han podido inicializar las cámaras!'
            )
71          print('             Ingrese nuevamente los números de
            dispositivo!')
72          print('\n')
73
74      while flag1:
75
76          l_ret, l_img = l_cam.read() #Lee un frame de la cámara
            izquierda.
77          r_ret, r_img = r_cam.read() #Lee un frame de la cárama
            derecha.
78
79          if l_ret:
80
81              cv.imshow('left_img', l_img)
82
83          if r_ret:
84
85              cv.imshow('right_img', r_img)
86
87          key = cv.waitKey(33)
88
89          if key == ord('y'):
90
91              flag1 = False
92              flag0 = False
```

```
93
94              l_cam.release()
95              r_cam.release()
96              cv.destroyAllWindows()
97
98          elif key == ord('n'):
99
100             flag1 = False
101
102             l_cam.release()
103             r_cam.release()
104             cv.destroyAllWindows()
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=
                    l_map2, r_map1=r_map1, r_map2=r_map2,
148                                             l_P=l_P, B=B,
149                                             l_cam_num=l_cam_num,
                                                    r_cam_num=r_cam_num)
150
151             print('(Info) Números de dispositivo actualizados
                    exitosamente!')
152             print('\n')
153
154             quit()
155
156         except:
157
158             print('(Error) No se ha encontrado archivo npz. con pará
                    matros intr. estéreo!')
159             print('        Se procederá a la sección de calibración
                    estéreo.')
160             print('\n')
161
162     else:
163
164         print('    (Error) Opción inválida!')
165         print('\n')
166
167
168 #Selección de la calibración.
169 print('Modalidad de calibración:')
170 print('')
171
172 flag = True
173
174 while flag:
175
176     print('    [1] Calibración estéreo')
177     print('    [2] Test')
178     print('')
179
180     o = input('    >> Opción: ')
181
182     print('\n')
183
184     if o == '1':
185
186         name = ''
187         rel_dir = 'stereo/'
188         flag = False
189
190     elif o == '2':
191
```

```
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 print('')
207 nx = int(input('    >> Nro. de intersecciones horizontales = '))
208 ny = int(input('    >> Nro. de intersecciones verticales   = '))
209
210 print('\n')
211
212
213 n = int(input('>> Número de imágenes válidas requeridas = ')) #
    Cantidad de imágenes válidas requeridas.
214 print('\n')
215
216
217 i = 0 # Nro. de imágenes válidas.
218
219 criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30,
    0.001) #Criterio de terminación.
220
221 #Preparar puntos de objeto en formato (0,0,0), (1*size,0,0), (2*size
    ,0,0), ...,((ny-1)*size,(nx-1)*size,0).
222 ptnpts = np.zeros((nx*ny,3), np.float32)
223 ptnpts[:,:2] = np.mgrid[0:ny*size:size, 0:nx*size:size].T.reshape
    (-1,2)
224
225 #Arreglos para almacenar puntos de objeto y puntos de imagen desde
    todas las imágenes válidas.
226 objpts = [] #Puntos 3D del espacio real.
227 l_imgpts = [] #Puntos 2D en el plano de la imagen de la cámara
    izquierda.
228 r_imgpts = [] #Puntos 2D en el plano de la imagen de la cámara
    derecha.
229
230
231 #Recolección de imágenes válidas en tiempo real y de información
    necesaria para calibración.
232
233 print('>> Procesar imágenes:')
234 print('')
```

```
print('    Durante cualquier ventana de captura activa, presionar [key]:')
print('')
print('        [1] Procesar imágenes')
print('\n')

print('(Info) Inicializando cámaras...')
print('')

#Inicia la captura de video.
l_cam = cv.VideoCapture(l_cam_num)
r_cam = cv.VideoCapture(r_cam_num)

print('(Info) Cámaras inicializadas!')
print('\n')

print('(Info) Válidas = '+str(i))

while i < n: #Mientras la cantidad de imágenes válidas sea menor a lo requerido...

    l_ret, l_img = l_cam.read() #Lee un frame de la cámara izquierda.
    r_ret, r_img = r_cam.read() #Lee un frame de la cárama derecha.

    if l_ret and r_ret: #Si ambas capturas fueron exitosas...

        #Imágenes a escala de grises.
        l_gray = cv.cvtColor(l_img, cv.COLOR_BGR2GRAY)
        r_gray = cv.cvtColor(r_img, cv.COLOR_BGR2GRAY)

        cv.imshow('Left img.: Gray', l_gray)
        cv.imshow('Right img.: Gray', r_gray)
        key = cv.waitKey(33)

        if key == ord('1'):

            #Encontrar las esquinas del tablero de ajedrez.
            l_ret, l_corners = cv.findChessboardCorners(l_gray, (ny, nx), None)
            r_ret, r_corners = cv.findChessboardCorners(r_gray, (ny, nx), None)

            if l_ret and r_ret: #Si encuentra las esquinas en ambas capturas...

                i += 1 #Incrementa el número de imágenes válidas.
                print('(Info) Válidas = '+str(i))

                objpts.append(ptnpts) #Añade los puntos de objeto (3D)

                l_corners2 = cv.cornerSubPix(l_gray, l_corners,
```

```
                    (11,11), (-1,-1), criteria)
                l_imgpts.append(l_corners2)

                r_corners2 = cv.cornerSubPix(r_gray, r_corners,
                    (11,11), (-1,-1), criteria)
                r_imgpts.append(r_corners2)

                #Dibujar y mostrar esquinas del tablero.
                l_gray_bgr = cv.cvtColor(l_gray, cv.COLOR_GRAY2BGR)
                r_gray_bgr = cv.cvtColor(r_gray, cv.COLOR_GRAY2BGR)

                cv.drawChessboardCorners(l_gray_bgr, (ny,nx),
                    l_corners2, l_ret)
                cv.imshow(name + 'left_v-img', l_gray_bgr)

                cv.drawChessboardCorners(r_gray_bgr, (ny,nx),
                    r_corners2, r_ret)
                cv.imshow(name + 'right_v-img', r_gray_bgr)
                cv.waitKey(100)

                #Guardar imágenes válidas.
                l_vimg_name = generate_vimg_name(name + 'left', i)
                cv.imwrite(rel_dir + l_vimg_name, l_gray)

                r_vimg_name = generate_vimg_name(name + 'right', i)
                cv.imwrite(rel_dir + r_vimg_name, r_gray)


l_cam.release()
r_cam.release()
cv.destroyAllWindows()
print('\n')


parent_folder_dir = os.path.normpath(os.getcwd() + os.sep + os.pardir
    ) #Retrocede al directorio padre.

parent_rel_dir = '/mono_intrinsics/mono/' #Dirección relativa al
    directorio padre cam. izq..
file_name = 'mono_intr.npz' #Nombre del archivo .npz.
path = os.path.abspath(parent_folder_dir + parent_rel_dir + file_name
    )

#Cargar archivo con parámetros intrínsecos de las cámaras.
npzfile = np.load(path)

imtx = npzfile['imtx']
distc = npzfile['distc']


#Realizar calibración estéreo.
retvals = []
```

```
retvals = cv.stereoCalibrate(objpts, l_imgpts, r_imgpts, imtx, distc, imtx, distc,
                                        l_gray.shape[::-1], flags=cv.CALIB_FIX_INTRINSIC)

reproy_err_rms = retvals[0] #Error de re-proyección RMS en [px].
#Parámetros resultantes de la calibración estéreo; describen a la cámara izquierda respecto de la derecha:
R = retvals[5] #Matriz de rotación del sistema coordenado.
T = retvals[6] #Vector de traslación del sistema coordenado.
F = retvals[8] #Matriz fundamental.

Tx = T[0][0]
Ty = T[1][0]
Tz = T[2][0]
B = np.sqrt([Tx**2 + Ty**2 + Tz**2])
B = B[0] #Distancia base (entre centros focales) en [mm].


#Obtener matrices de rectificación estéreo.
retvals = []
retvals = cv.stereoRectify(imtx, distc, imtx, distc, l_gray.shape[::-1], R, T)

l_R = retvals[0] #Matriz de rectificación por rotación para la cámara izquierda.
r_R = retvals[1] #Matriz de rectificación por rotación para la cámara derecha.
l_P = retvals[2] #Matriz de proyección en el sist. coord. rectificado para la cámara izquierda.
r_P = retvals[3] #Matriz de proyección en el sist. coord. rectificado para la cámara derecha.


#Obtener mapas para eliminar distorsión y rectificar las imágenes.
l_map1, l_map2 = cv.initUndistortRectifyMap(imtx, distc, l_R, l_P, l_gray.shape[::-1], cv.CV_32FC1)
r_map1, r_map2 = cv.initUndistortRectifyMap(imtx, distc, r_R, r_P, l_gray.shape[::-1], cv.CV_32FC1)


print('(Info) Error RMS de re-proyección [px] = '+str(reproy_err_rms))
print('\n')


#Cálculo de error RMS de matriz fundamental (F) en base a los puntos de imagen encontrados y según [p2|1]*F*[p1|1]' = 0.
quad_sum = 0

for i in range(0, len(l_imgpts)):

```

```
    l_pt_i = [[l_imgpts[0][i][0][0], l_imgpts[0][i][0][1], 1]]
    r_pt_i = [[r_imgpts[0][i][0][0], r_imgpts[0][i][0][1], 1]]

    error_i = np.dot(np.dot(r_pt_i, F), np.transpose(l_pt_i))
    error_i = error_i[0][0]

    quad_sum += (error_i*error_i)

quad_sum_mean = [quad_sum/len(l_imgpts)]
F_RMS_error = np.sqrt(quad_sum_mean)
F_RMS_error = round(F_RMS_error[0], 5)

print('(Info) Error RMS F = '+ str(F_RMS_error) + ' [px]')
print('\n')


#Guardar archivo .npz con los parámetros estéreo.
np.savez(rel_dir + name + 'stereo_intr', l_map1=l_map1, l_map2=l_map2, r_map1=r_map1, r_map2=r_map2,
                                          l_P=l_P, B=B,
                                          l_cam_num=l_cam_num, r_cam_num=r_cam_num)

print('(Info) Archivo .npz con parámetros intrínsecos estéreo generado!')
print('\n')
```

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

```
#-*- coding: cp1252 -*-

import numpy as np
import cv2 as cv
import cv2.aruco as aruco
import os


print('')

#Información del patrón de calibración.
print('Datos sobre el patrón de calibración:')
print('')

size = float(input('    >> Tamaño de lado del ArUco [mm] = '))
print('\n')


#Cargar archivo .npz con parámetros intrínsecos estéreo.
parent_folder_dir = os.path.normpath(os.getcwd() + os.sep + os.pardir) #Retrocede al directorio padre.
parent_rel_dir = '/stereo_intrinsics/stereo/'
file_name = 'stereo_intr.npz'
path = os.path.abspath(parent_folder_dir + parent_rel_dir + file_name)

npzfile = np.load(path)

cam_num = npzfile['l_cam_num']

map1 = npzfile['l_map1']
map2 = npzfile['l_map2']

imtx_rect = npzfile['l_P']
imtx_rect = imtx_rect[:,:-1] #Matriz aparente (tras rectificación) de la cámara izquierda.


#Instrucciones del funcionamiento del bucle de calibración extrínseca.
print('>> Procesar imágenes:')
print('')
print('    Durante cualquier ventana de captura activa, presionar [key]:')
print('')
print('         [1] Procesar imagen')
```

```
print('         [2] Ingresar coordenadas')
print('         [s] Generar archivo .npz con parámetros extrínsecos')
print('\n')

print('(Info) Inicializando cámara...')
print('')

cam = cv.VideoCapture(cam_num) #Inicia la captura de video de cám. izq.

print('(Info) Cámara inicializada!')
print('\n')

aruco_dict = aruco.Dictionary_get(aruco.DICT_6X6_250)
params = aruco.DetectorParameters_create()
params.adaptiveThreshConstant = 10
params.cornerRefinementMethod = aruco.CORNER_REFINE_SUBPIX

flag0 = True

while flag0: #Bucle calib. extr.

    ret, img = cam.read() #Lee un frame de la cámara.

    if ret: #Si la captura fue exitosa...

        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY) #Imagen a escala de grises.
        gray_rect = cv.remap(gray, map1, map2, cv.INTER_LINEAR)

        cv.imshow('Left img.: Gray rect.', gray_rect)
        k = cv.waitKey(33) #Almacena tecla presionada durante ventana activa.

        if k == ord('1'): #Si la tecla presionada es "1"...

            #Obtener intersecciones, id del aruco y puntos rechazados.
            corners, ids, rejectedImgPoints = aruco.detectMarkers(gray_rect, aruco_dict, parameters=params)

            corners = np.float32(corners)

            if np.all(ids != None):

                #Estimar pose del aruco (rotación y traslación) respecto de la cámara izq.
                rvec, tvec,_ = aruco.estimatePoseSingleMarkers(corners, size, imtx_rect, None)

                tvec = tvec[0]
                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,
                        None, rvec, tvec, 3*29)
92
93                  img_cordsys = aruco.drawDetectedMarkers(gray_rect,
                        corners)
94
95                  cv.imshow('img_cordsys', img_cordsys)
96                  cv.waitKey(33)
97
98                  print('rvec =')
99                  print(rvec)
100                 print('')
101
102                 print('tvec =')
103                 print(tvec)
104                 print('\n')
105
106         elif k == ord('2'):
107
108             #Bucle para ingresar coordenadas del punto de referencia
                    en base al sistema de YARBIZ.
109             flag1 = True
110
111             while flag1:
112
113                 print('Ingresar coordenadas (x,y,z) del origen
                        XYZ_calib en base a XYZ_YARBIZ:')
114                 print('')
115
116                 x_Oc_Oy = float(input('     >> x_Oc_Oy [mm] = '))
117                 y_Oc_Oy = float(input('     >> y_Oc_Oy [mm] = '))
118                 z_Oc_Oy = float(input('     >> z_Oc_Oy [mm] = '))
119                 print('\n')
120
121                 flag2 = True
122
123                 while flag2:
124
125                     print('     [1] Guardar coordenadas y continuar')
126                     print('     [2] Volver a ingresar coordenadas')
127                     print('')
128
129                     o = input('     >> Opción: ')
130                     print('\n')
131
132                     if o == '1':
133
134                         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
                        ) #Guarda imagen con ejes dibujados.
152
153                 rmtx, jacob = cv.Rodrigues(rvec[0]) #Transforma rvec
                        en una matriz de rotaciones 3x3.
154
155                 #Se genera matriz de transformación homogénea de sist
                        . rect. cám. izq. respecto a .sist. calib.
156                 HT_Oc_Olrect = np.zeros((4,4))
157                 HT_Oc_Olrect[3,3] = 1
158                 HT_Oc_Olrect[:-1,:-1] = rmtx[:,:]
159                 HT_Oc_Olrect[:-1,3] = tvec
160
161                 HT_Olrect_Oc = np.linalg.inv(HT_Oc_Olrect)
162
163                 #Crear matriz de transformación homogénea de
                        XYZ_calib respecto a XYZ_yarbiz.
164                 thetax_Oc_Oy = 90*np.pi/180
165                 Rx_Oc_Oy = np.array([[1,0,0],
166                                     [0,np.cos(thetax_Oc_Oy),-np.sin(
                                            thetax_Oc_Oy)],
167                                     [0,np.sin(thetax_Oc_Oy),np.cos(
                                            thetax_Oc_Oy)]])
168
169                 T_Oc_Oy = np.array([[x_Oc_Oy,y_Oc_Oy,z_Oc_Oy]])
170
171                 HT_Oc_Oy = np.zeros((4,4))
172                 HT_Oc_Oy[:-1,:-1] = Rx_Oc_Oy
173                 HT_Oc_Oy[:-1,3] = T_Oc_Oy
174                 HT_Oc_Oy[3,3] = 1
175
176                 #Matriz de transf. hom. de sist. rect. cám. izq.
                        respecto de sist. YARBIZ.
177                 HT_Olrect_Oy = np.dot(HT_Oc_Oy, HT_Olrect_Oc)
178
179
```

```
                #Crear archivo .npz con parámetros extrínsecos.
                np.savez('mono/mono_extr', HT_Olrect_Oy=HT_Olrect_Oy)

                print('(Info) Archivo .npz con parámetros extrínsecos generado!')
                print('\n')

                flag0 = False

            except:

                print('(Error) No se ha procesado la imagen y/o no se han ingresado coordenadas!')


cam.release()
cv.destroyAllWindows()
```

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

```
#-*- coding: cp1252 -*-

import numpy as np
from yarbiz_kine_v3 import ik
from yarbiz_kine_v3 import fk
import cv2 as cv
import os
import time


print('')

com = 'COM3' #Almacenar puerto COM de OpenCM9.04
baud = 115200 #Almacenar Baud Rate.

segm = 50 #Tamaño del segmento de movimiento de YARBIZ

tol = 3 #Tolerancia en [mm] del resultado final del movimiento.

diam = 40 #Diámetro objeto en [mm].

thetai_reset = [-90,150,-90,-45,0] #Ángulos de posición inicial de
    YARBIZ.

thetai_bring = [45,65,-60,-45,0]

thetai, p = fk(thetai_reset, com, baud) #Mover YARBIZ a posición
    inicial.


#Cargar archivo .npz de la calibración estéreo.
rel_dir = 'calibration/stereo_intrinsics/stereo/' #Dirección relativa
    .
file_name = 'stereo_intr.npz' #Nombre del archivo .npz.
path = rel_dir + file_name

npzfile = np.load(path)

l_cam_num = npzfile['l_cam_num'] #Números de dispositivo de las cá
    maras.
r_cam_num = npzfile['r_cam_num']

l_map1 = npzfile['l_map1'] #Mapas de rectificación para la cámara
    izquierda.
l_map2 = npzfile['l_map2']

```

```
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,
```

```
                                    preFilterCap=pfc,
                                    uniquenessRatio=uRatio,
                                    speckleWindowSize=sp_win_size,
                                    speckleRange=sp_range,
                                    mode=cv.STEREO_SGBM_MODE_SGBM_3WAY)

#Inicializar filtro WLS.
r_matcher = cv.ximgproc.createRightMatcher(l_matcher) #Crear matcher derecho.

wls_filter = cv.ximgproc.createDisparityWLSFilter(matcher_left=l_matcher)
wls_filter.setLambda(lmbda)
wls_filter.setSigmaColor(sigma)


#Crear sliders:
def nothing(x):
    pass


print('>> Configurar:')
print('')
print('    Durante cualquier ventana de captura activa, presionar [key]:')
print('')
print('        [s] Guardar cambios')
print('\n')

print('(Info) Inicializando panel de sliders...')
print('')

time.sleep(4)

cfg_w_name = 'Stereo config.' #Nombre de la ventana de sliders.
cv.namedWindow(cfg_w_name, cv.WINDOW_NORMAL) #Crear ventana para sliders.

slrs = {'minDisp'      : ['minDisp', 0, 200],
        'numDisps'     : ['numDisps', 1, 20],
        'bsize'        : ['BlockSize', 1, 15],
        'SAD_win_size' : ['SADWinSize', 3, 31],
        'pfc'          : ['PreFilCap', 0, 100],
        'uRatio'       : ['UniqRatio', 0, 20],
        'sp_win_size'  : ['SpWinSize', 0, 220],
        'sp_range'     : ['SpRange', 0, 5],
        'lmbda'        : ['Lambda', 0, 200000],
        'sigma'        : ['Sigma', 0, 200],
        'lowerH'       : ['Lower H', 0, 180],
        'lowerS'       : ['Lower S', 0, 255],
        'lowerV'       : ['Lower V', 0, 255],
        'upperH'       : ['Upper H', 0, 180],
```

```
        'upperS'        : ['Upper S', 0, 255],
        'upperV'        : ['Upper V', 0, 255],
        'precn_lvl'     : ['CurvePrLvl', 1, 200],
        'min_sides'     : ['CurveMinSi', 3, 100]
           }

cv.createTrackbar(slrs['minDisp'][0], cfg_w_name, minDisp, slrs['minDisp'][2], nothing)
cv.createTrackbar(slrs['numDisps'][0], cfg_w_name, numDisps, slrs['numDisps'][2], nothing)
cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize, slrs['bsize'][2], nothing)
cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name, SAD_win_size, slrs['SAD_win_size'][2], nothing)
cv.createTrackbar(slrs['pfc'][0], cfg_w_name, pfc, slrs['pfc'][2], nothing)
cv.createTrackbar(slrs['uRatio'][0], cfg_w_name, uRatio, slrs['uRatio'][2], nothing)
cv.createTrackbar(slrs['sp_win_size'][0], cfg_w_name, sp_win_size, slrs['sp_win_size'][2], nothing)
cv.createTrackbar(slrs['sp_range'][0], cfg_w_name, sp_range, slrs['sp_range'][2], nothing)
cv.createTrackbar(slrs['lmbda'][0], cfg_w_name, lmbda, slrs['lmbda'][2], nothing)
cv.createTrackbar(slrs['sigma'][0], cfg_w_name, int(sigma*100), slrs['sigma'][2], nothing)
cv.createTrackbar(slrs['lowerH'][0], cfg_w_name, lowerH, slrs['lowerH'][2], nothing)
cv.createTrackbar(slrs['upperH'][0], cfg_w_name, upperH, slrs['upperH'][2], nothing)
cv.createTrackbar(slrs['lowerS'][0], cfg_w_name, lowerS, slrs['lowerS'][2], nothing)
cv.createTrackbar(slrs['upperS'][0], cfg_w_name, upperS, slrs['upperS'][2], nothing)
cv.createTrackbar(slrs['lowerV'][0], cfg_w_name, lowerV, slrs['lowerV'][2], nothing)
cv.createTrackbar(slrs['upperV'][0], cfg_w_name, upperV, slrs['upperV'][2], nothing)
cv.createTrackbar(slrs['precn_lvl'][0], cfg_w_name, precn_lvl, slrs['precn_lvl'][2], nothing)
cv.createTrackbar(slrs['min_sides'][0], cfg_w_name, min_sides, slrs['min_sides'][2], nothing)

change = 0 #Evaluador de cambio en config.

print('(Info) Panel de sliders inicializado!')
print('\n')

#Valor inicial de coord. en pixeles.
u = 0
v = 0

```

```
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 imágen 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
                en calibración estéreo.
293         l_img_rect = cv.remap(l_img, l_map1, l_map2, cv.INTER_LINEAR)
294         r_img_rect = cv.remap(r_img, r_map1, r_map2, cv.INTER_LINEAR)
295
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
                int16.
308         l_disp = np.int16(l_disp)
309         r_disp = np.int16(r_disp)
310
311         #Generar mapa de disparidad basado en imagen izq. con filtro
                WLS aplicado.
312         disp = wls_filter.filter(l_disp, l_gray_rect, None, r_disp)
313
314         #Normalizar mapa de disparidad y transformar datos a valores
                de saturación en escala de grises SAT[0,255].
315         disp_norm = wls_filter.filter(l_disp, l_gray_rect, None,
                r_disp)
316         disp_norm = cv.normalize(src=disp_norm, dst=disp_norm,
317                                  beta=0, alpha=255,
318                                  norm_type=cv.NORM_MINMAX)
319
320         disp_norm = np.uint8(disp_norm) #Transformar valores a tipo
                uint8.
321
```

```
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             l_hsv_rect = cv.cvtColor(l_img_rect, cv.COLOR_BGR2HSV) #
                Canales BGR a HSV.
334
335             #Definir límites superior e inferior de canal HSV.
336             lowerHSV = np.array([lowerH, lowerS, lowerV])
337             upperHSV = np.array([upperH, upperS, upperV])
338
339             #Generar imagen binaria (máscara): - blanco para lo que
                se encuentre dentro de los límites HSV.
340                                 #              - negro para lo que se
                                        encuentre fuera.
341             l_hsv_mask = cv.inRange(l_hsv_rect, lowerHSV, upperHSV)
342
343
344             contours, hie = cv.findContours(l_hsv_mask, 0, 2) #
                Encontrars contornos.
345
346
347             try: #Manejar errores en caso de no encontrar curvas.
348
349                 validCurves = [] #Lista para almacenar curvas válidas
                    .
350
351                 for c in contours:
352
353                     arclen = cv.arcLength(c, True) #Perímetro de la
                        figura. True = figura cerrada.
354                     epsilon = arclen*(1/(precn_lvl*10)) #Precisión de
                         la aproximación de las curvas.
355                     appCurve = cv.approxPolyDP(c, epsilon, True) #
                        Aproximar curva.
356
357                     if len(appCurve) >= min_sides: #Número de lados
                        requerido luego de aproximación.
358
359                         validCurves.append(appCurve)
360
361
362                 centroids = [] #Lista para almacenar centroides de
                    las curvas válidas.
```

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

```
406                     cv.circle(r_gray_rect, (u-int(round(disp_uv,0)),v
                        ), 2, curvecolor, -1) #Dibujar centroide en
                        imagen der.
407
408                     cv.putText(r_gray_rect, str(i+1), (u-int(round(
                        disp_uv,0))+5,v), font, fsize1, curvecolor,
                        1, cv.LINE_AA)
409
410                     centdisps.append(disp_uv) #Añade disparidad a la
                        lista.
411
412
413                 disp_uv = centdisps[curvenum-1]
414                 u = centroids[curvenum-1][0]
415                 v = centroids[curvenum-1][1]
416
417
418             except:
419
420                 disp_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             disp_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(disp_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
                disparidad.
446
447             z_Olrect = B*f/disp_uv
448             x_Olrect = z_Olrect*(u-l_cx)/f
449             y_Olrect = z_Olrect*(v-l_cy)/f
450
451             if mode == 'auto':
452
453                 norm = np.sqrt([x_Olrect**2 + y_Olrect**2 + z_Olrect
                        **2])
454                 norm = norm[0] #Módulo de p_Olrect.
455
456                 dx_Olrect = x_Olrect/norm
457                 dy_Olrect = y_Olrect/norm
458                 dz_Olrect = z_Olrect/norm
459
460                 x_Olrect = x_Olrect + dx_Olrect*(diam/2)
461                 y_Olrect = y_Olrect + dy_Olrect*(diam/2)
462                 z_Olrect = z_Olrect + dz_Olrect*(diam/2)
463
464             p_Olrect = np.array([[x_Olrect], #Vector objetivo en base
                    a Olrect.
465                                 [y_Olrect],
466                                 [z_Olrect],
467                                 [1]])
468
469             #Calcular coordenadas en base al sist. coord. de YARBIZ.
470             p_Oy = np.dot(HT_Olrect_Oy, p_Olrect) #Vector objetivo
                    respecto a YARBIZ.
471             p_Oy = p_Oy[:-1,:]
472
473             if mode == 'auto':
474
475                 norm = np.sqrt([p_Oy[0,0]**2 + p_Oy[1,0]**2])
476                 norm = norm[0]
477
478                 d_Oy = np.array([[p_Oy[0,0]],
479                                 [p_Oy[1,0]],
480                                 [0]])
481
482                 d_Oy = d_Oy*(1/norm)
483
484                 p_Oy = p_Oy - d_Oy*(diam/2)
485
486
487             #Aproximación de coordenadas a un decimal (más cercano)
                    para prints.
488             x_Oy = round(p_Oy[0,0],1)
489             y_Oy = round(p_Oy[1,0],1)
490             z_Oy = round(p_Oy[2,0],1)
491
```

```
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(disp_norm, 'Modo = ' + mode, (5,1*linespace),
                    font, fsize2, (0,255,0), 1, cv.LINE_AA)
498             cv.putText(disp_norm, 'Curva = ' + str(curvenum), (5,2*
                    linespace), font, fsize2, (0,255,0), 1, cv.LINE_AA)
499             cv.putText(disp_norm, '(u,v) = (' + str(u) + ', ' + str(v
                    ) + ') [px]', (5,3*linespace), font, fsize2,
                    (0,255,0), 1, cv.LINE_AA)
500             cv.putText(disp_norm, 'Disp. = ' + str(int(round(disp_uv
                    ,0))) + ' [px]', (5,4*linespace), font, fsize2,
                    (0,255,0), 1, cv.LINE_AA)
501             cv.putText(disp_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(disp_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(disp_norm, 'Modo = ' + mode, (5,1*linespace),
                    font, fsize2, (0,255,0), 1, cv.LINE_AA)
510             cv.putText(disp_norm, 'Curva = ' + str(curvenum), (5,2*
                    linespace), font, fsize2, (0,255,0), 1, cv.LINE_AA)
511             cv.putText(disp_norm, '(u,v) = - [px]', (5,3*linespace),
                    font, fsize2, (0,255,0), 1, cv.LINE_AA)
512             cv.putText(disp_norm, 'Disp. = - [px]', (5,4*linespace),
                    font, fsize2, (0,255,0), 1, cv.LINE_AA)
513             cv.putText(disp_norm, 'p_Oy = - [mm]', (5,5*linespace),
                    font, fsize2, (0,255,0), 1, cv.LINE_AA)
514             cv.putText(disp_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', disp_norm)
521
522
523         #Obtención y corrección de variables de sliders.
524         minDisp = cv.getTrackbarPos(slrs['minDisp'][0], cfg_w_name) #
```

```
            MinDisparity.
        if minDisp != minDisp0:
            change = 1

        numDisps = cv.getTrackbarPos(slrs['numDisps'][0], cfg_w_name) #NumDisparities.
        if numDisps != numDisps0:
            if numDisps < slrs['numDisps'][1]:
                numDisps = slrs['numDisps'][1]
                cv.createTrackbar(slrs['numDisps'][0], cfg_w_name, numDisps, slrs['numDisps'][2], nothing)
            change = 1

        bsize = cv.getTrackbarPos(slrs['bsize'][0], cfg_w_name) # BlockSize.
        if bsize != bsize0:
            if bsize > slrs['bsize'][1] and bsize%2 == 0: #Si es par, se le resta 1 unidad.
                if bsize > bsize0:
                    bsize += 1
                elif bsize < bsize0:
                    bsize -= 1
                cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize, slrs['bsize'][2], nothing)
            elif bsize < slrs['bsize'][1]:
                bsize = slrs['bsize'][1]
                cv.createTrackbar(slrs['bsize'][0], cfg_w_name, bsize, slrs['bsize'][2], nothing)
            change = 1

        SAD_win_size = cv.getTrackbarPos(slrs['SAD_win_size'][0], cfg_w_name)
        if SAD_win_size != SAD_win_size0:
            if SAD_win_size > slrs['SAD_win_size'][1] and SAD_win_size%2 == 0:
                if SAD_win_size > SAD_win_size0:
                    SAD_win_size += 1
                elif SAD_win_size < SAD_win_size0:
                    SAD_win_size -= 1
                cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name, SAD_win_size, slrs['SAD_win_size'][2], nothing)
            elif SAD_win_size < slrs['SAD_win_size'][1]:
                SAD_win_size = slrs['SAD_win_size'][1]
                cv.createTrackbar(slrs['SAD_win_size'][0], cfg_w_name, SAD_win_size, slrs['SAD_win_size'][2], nothing)
            change = 1

        pfc = cv.getTrackbarPos(slrs['pfc'][0], cfg_w_name) # PreFilterCap.
        if pfc != pfc0:
            change = 1

```

```
565         uRatio = cv.getTrackbarPos(slrs['uRatio'][0], cfg_w_name) #
                UniquenessRatio.
566         if uRatio != uRatio0:
567             change = 1
568
569         sp_win_size = cv.getTrackbarPos(slrs['sp_win_size'][0],
                cfg_w_name) #SpeckleWindowSize.
570         if sp_win_size != sp_win_size0:
571             change = 1
572
573         sp_range = cv.getTrackbarPos(slrs['sp_range'][0], cfg_w_name)
                 #SpeckleRange.
574         if sp_range != sp_range0:
575             change = 1
576
577         lmbda = cv.getTrackbarPos(slrs['lmbda'][0], cfg_w_name)
578         if lmbda != lmbda0:
579             change = 1
580
581         sigma = cv.getTrackbarPos(slrs['sigma'][0], cfg_w_name)
582         sigma = round(sigma/100,2)
583         if sigma != sigma0:
584             change = 1
585
586         lowerH = cv.getTrackbarPos(slrs['lowerH'][0], cfg_w_name)
587         upperH = cv.getTrackbarPos(slrs['upperH'][0], cfg_w_name)
588         lowerS = cv.getTrackbarPos(slrs['lowerS'][0], cfg_w_name)
589         upperS = cv.getTrackbarPos(slrs['upperS'][0], cfg_w_name)
590         lowerV = cv.getTrackbarPos(slrs['lowerV'][0], cfg_w_name)
591         upperV = cv.getTrackbarPos(slrs['upperV'][0], cfg_w_name)
592
593         precn_lvl = cv.getTrackbarPos(slrs['precn_lvl'][0],
                cfg_w_name)
594         if precn_lvl < slrs['precn_lvl'][1]:
595             precn_lvl = slrs['precn_lvl'][1]
596             cv.createTrackbar(slrs['precn_lvl'][0], cfg_w_name,
                    precn_lvl, slrs['precn_lvl'][2], nothing)
597
598         min_sides = cv.getTrackbarPos(slrs['min_sides'][0],
                cfg_w_name)
599         if min_sides < slrs['min_sides'][1]:
600             min_sides = slrs['min_sides'][1]
601             cv.createTrackbar(slrs['min_sides'][0], cfg_w_name,
                    min_sides, slrs['min_sides'][2], nothing)
602
603
604         #Almacenar para evaluar cambios.
605         minDisp0 = minDisp #MinDisparity.
606         numDisps0 = numDisps #NumDisparities.
607         bsize0 = bsize #BlockSize.
608         SAD_win_size0 = SAD_win_size
609         pfc0 = pfc #PreFilterCap.
```

```
610         uRatio0 = uRatio #UniquenessRatio.
611         sp_win_size0 = sp_win_size #SpeckleWindowSize.
612         sp_range0 = sp_range #SpeckleRange.
613         lmbda0 = lmbda
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
                                                     *16,
620                                              blockSize=bsize,
621                                              P1=8*img_chans*
                                                     SAD_win_size**2,
622                                              P2=32*img_chans*
                                                     SAD_win_size**2,
623                                              disp12MaxDiff=-1,
624                                              preFilterCap=pfc,
625                                              uniquenessRatio=uRatio,
626                                              speckleWindowSize=
                                                     sp_win_size,
627                                              speckleRange=sp_range,
628                                              mode=cv.
                                                     STEREO_SGBM_MODE_SGBM_3WAY
                                                     )
629 
630             r_matcher = cv.ximgproc.createRightMatcher(l_matcher) #
                    Crear matcher derecho.
631 
632             wls_filter = cv.ximgproc.createDisparityWLSFilter(
                    matcher_left=l_matcher)
633             wls_filter.setLambda(lmbda)
634             wls_filter.setSigmaColor(sigma)
635 
636             change = 0
637 
638 
639         key = cv.waitKey(33)
640 
641         #Bloque de control según [key] presionada.
642 
643         if key == ord('s'): #Guardar configuración.
644 
645             np.savez('svconfig.npz', minDisp=minDisp,
646                                      numDisps=numDisps,
647                                      bsize=bsize,
648                                      SAD_win_size=SAD_win_size,
649                                      pfc=pfc,
650                                      uRatio=uRatio,
651                                      sp_win_size=sp_win_size,
652                                      sp_range=sp_range,
653                                      lmbda=lmbda,
```

```
                                        sigma=sigma,
                                        lowerH = lowerH,
                                        lowerS = lowerS,
                                        lowerV = lowerV,
                                        upperH = upperH,
                                        upperS = upperS,
                                        upperV = upperV,
                                        precn_lvl = precn_lvl,
                                        min_sides = min_sides)

            print('\n')
            print('(Info) Archivo svconfig generado!')
            print('\n')
            print('')

        if key == ord('n'):

            if mode == 'manual':
                curvenum = 1
            else:
                pass

            mode = 'auto'

        if key == ord('m'):

            mode = 'manual'

            curvenum = '-'

        if mode == 'auto':

            if key == ord('1'):

                curvenum = 1

            elif key == ord('2'):

                curvenum = 2

            elif key == ord('3'):

                curvenum = 3

            elif key == ord('4'):

                curvenum = 4

            elif key == ord('5'):

                curvenum = 5

```

```
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 ubación 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
```