Facultad de Ingeniería Eléctrica

Universidad Michoacana de San Nicolás de Hidalgo

DESARROLLO DE UN ROBOT AUTÓNOMO DE SUMO EQUIPADO CON CÁMARA

TESIS

Que para obtener el grado de

INGENIERO EN COMPUTACIÓN

presenta

Alejandro Calderón Carrillo

Leonardo Romero Muñoz

Director de Tesis

Morelia, Mich. Diciembre 2019

Este trabajo está dedicado principalmente a mi madre y a mi familia por su apoyo incondicional en todo momento.

A mis amigos y compañeros porque hicieron de mi estadía en la facultad una una época llena de recuerdos divertidos y alegres.

A mi asesor de tesis por su paciencia y el tiempo que invirtió en mi.

Resumen

En los últimos años, se han llevado a cabo competencias de robótica en Michoacán y una de las competencias en la que se tiene una gran cantidad de participantes es la de "Lucha de robots de sumo". El objetivo de la competencia es sacar al robot oponente fuera del área de combate. Se ha observado que en estas competencias los robots de sumo utilizan comúnmente sensores ultrasónicos para la detección del robot oponente y sensores reflexivos para la detección de la zona blanca y negra del área de combate.

En esta tesis se describe el desarrollo de un robot de sumo que incorpora una cámara como sensor principal y tiene la capacidad de procesamiento de las imágenes por medio de una computadora que se encuentra a bordo del robot, de manera que puede detectar el área de combate y estimar la posición del robot oponente. Para realizar la tarea de procesamiento de imágenes, se desarrolló una aplicación en el lenguaje C++ utilizando la biblioteca OpenCV.

La ventaja que se tiene al utilizar una cámara como sensor principal es que el robot conoce su posición y la de su oponente dentro del área de combate la gran mayoría del tiempo. Por otro lado, los robots comunes usualmente sólo conocen si se encuentran en la zona blanca o en la zona negra del área de combate y si el robot oponente está enfrente de ellos.

La detección del área de combate y del robot oponente se logra utilizando un modelo de distribución normal con imágenes en escala de grises para que el robot tenga la capacidad de distinguir los colores del área de combate (blanco y negro) y también pueda distinguir al robot oponente mediante los píxeles que no correspondan a este modelo de colores.

Se realizaron pruebas para valorar la rapidez del procesamiento de las imágenes y la precisión de los cálculos realizados para la detección del área de combate y la estimación de la posición del robot oponente. Los resultados obtenidos muestran una precisión muy buena y una rapidez de procesamiento de imágenes de hasta tres imágenes por segundo.

Palabras clave: Visión computacional, competencia, combate, inteligencia artificial, software.

Abstract

In recent years, robotics competitions have been held in Michoacán and one of the competitions in which there is a large number of participants is the "Sumo robot fight". The goal of this competition is to take the opposing robot out of the combat area. It has been observed that sumo robots commonly use ultrasonic sensors for the detection of the opposing robot and reflective sensors for the detection of the black and white zone of the combat area.

This thesis describes the development of a sumo robot that incorporates a camera as the main sensor and has the ability to process the images by means of a computer the is on board the robot, so that it can detect the combat area and estimate the position of the opposing robot. To perform the image processing task, an application was developed in the C++ language using the OpenCV library.

The advantage of using a camera as the main sensor is that the robot knows its position and the location of the opposing robot within the combat area most of the time. On the other hand, common robots usually only know if they are in the white or black zone of the combat area and if the opposing robot is in front of them.

The detection of the combat area and the opposing robot is achieved by using a normal distribution model with grayscale images so that the robot has the ability to distinguish the colors of the combat area (black and white) and can also distinguish the opposing robot using pixels that do not correspond to this color model.

Tests were carried out to assess the speed of image processing and the accuracy of the calculations performed for the detection of the combat area and estimation of the position of the opposing robot. The results obtained show very good accuracy and an image processing speed of up to three images per second.

Contenido

Lista de Figuras

Lista de Tablas

Lista de Publicaciones

"Desarrollo de un robot autónomo de sumo equipado con cámara."
Alejandro Calderón Carrillo, Leonardo Romero Muñoz y Moisés García Villanueva.
14to Congreso Estatal de Ciencia, Tecnología e Innovación. Realizado en Morelia Michoacán el 29 y 30 de octubre de 2019 en las instalaciones de la UVAQ Campus Sta. María.

Capítulo 1

Introducción

Nosotros, los seres humanos, captamos información de nuestro entorno mediante distintos sensores conocidos como los cinco sentidos. Entre estos sentidos, uno de los más importantes es el sentido de la vista, ya que a través de este sentido una persona es capaz de adquirir información respecto a su entorno. Por otro lado, Gary Bradski *et al.* [Kaehler16] destaca que la visión inicia con la detección de la luz en nuestros ojos, donde esa luz comienza como rayos que emanan de una fuente (por ejemplo, una lámpara o el sol), los cuales viajan a través del espacio hasta incidir en algún objeto. Cuando esa luz llega al objeto, gran parte de la luz se absorbe y lo que no se absorbe lo percibimos como el color del objeto, como se ilustra en la Figura 1.1. Gracias a este proceso, no solo los seres humanos, si no también la mayoría de los animales tienen la capacidad de reconocer su entorno.

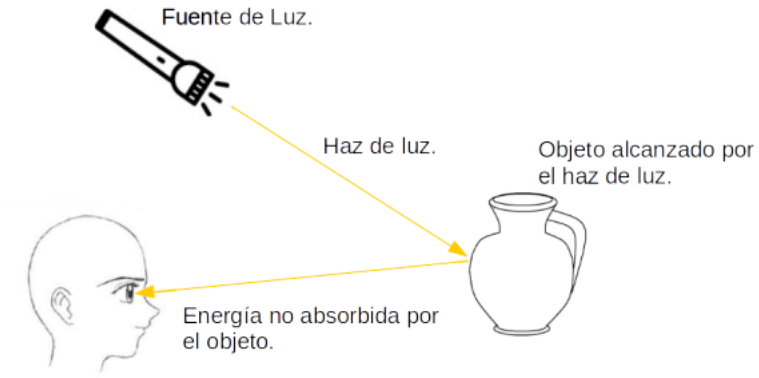


Figura 1.1: Interacción entre un rayo de luz y el ojo humano.

1.1. Planteamiento del problema

Al igual que los seres humanos tenemos vista, el propósito de esta tesis es dotar a un pequeño robot móvil de la capacidad de ver su entorno, para poder realizar su tarea. Para lograr este propósito, el robot móvil tiene capacidad de movimiento e integra una computadora a bordo y una cámara. La tarea seleccionada para el robot es participar en una competencia de robots de sumo.

La competencia de robots de sumo consiste en una lucha entre dos robots autónomos que se realiza sobre una área de combate denominada Ring o Dojo, como se muestra en la Figura 1.2. El objetivo de la lucha es empujar al robot oponente, de manera que se salga del área de combate.

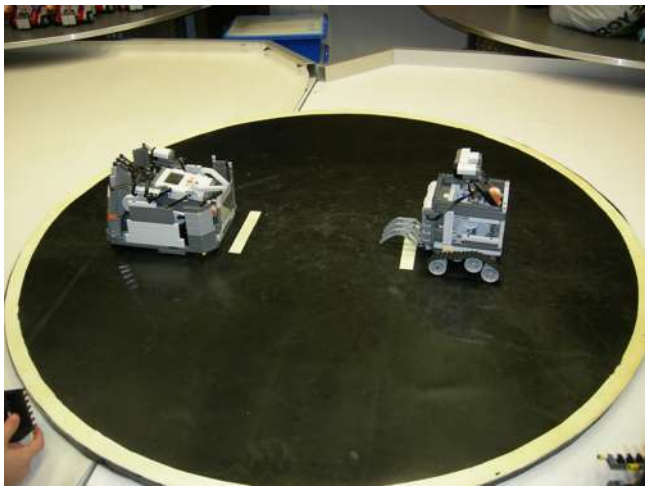


Figura 1.2: Lucha de robots de sumo.

Conforme al reglamento establecido para la competencia de robot de sumo de Robomatrix [Robomatrix18], se tienen cuatro categorías de lucha de sumos: megasumo, minisumo, microsumo y nanosumo; diferenciándose entre si por el tamaño y el peso máximo de los robots participantes. Conviene destacar que Robomatrix es la Liga Latinoamericana de Robótica en Competencia, que patrocina la Sociedad Latinoamericana de Ciencia y Tecnología.

La categoría seleccionada para competir fue la de megasumo. En esta categoría, el robot tiene dimensiones máximas de 20cm x 20cm y un peso máximo de 3 kilogramos, sin restricción en cuanto a altura. El diámetro del dojo es de 154 cm, de color negro, con

un borde blanco de 5 cm.

Para la competencia de sumo se deben cumplir las siguientes reglas:

- El robot luchador de sumo deberá ser de tipo autónomo y no podrá ser controlado por algún dispositivo externo.

- El robot deberá poseer un mecanismo que realice una cuenta de tiempo de seguridad igual a cinco segundos después de su activación y antes de proceder a su primer movimiento.

- Al iniciar el combate, el robot podrá desplegar elementos que se encuentren unidos físicamente a él.

- Los robots deberán contar con un pulsador o interruptor de encendido/apagado externo visible y accesible para poder iniciar las competencias o detener al robot en caso necesario.

- Está permitido el uso de un control remoto infrarrojo solo para detener el robot en caso de ser necesario y así evitar posibles daños.

- El incumplimiento de algunos de estos puntos será motivo de descalificación del robot en la competición, sin posibilidad de reintegro.

Cada partida del combate considera tres asaltos o rondas de una duración máxima de tres minutos cada uno, en las cuales los robots inician en la posiciones definidas en la Figura 1.3.

Figura 1.3: Posiciones iniciales para cada ronda en la competencia de robots de sumo de Robomatrix.

Se otorga la victoria en el asalto cuando:

- El robot contrario toque primero el área fuera del dojo.
- El robot contrario esté más de 30 segundos sin moverse.
- Por acumulación de violaciones por parte del equipo contrario en el mismo combate.

Un robot gana la lucha cuando gana en dos de los tres asaltos.

1.2. Antecedentes

Para la competencia de robots de sumo, cada equipo competidor tiene la libertad de construir su robot como desee siempre y cuando respete el reglamento.

Normalmente se utilizan robots móviles con dos o cuatro ruedas, con o sin rampa (para levantar al oponente) y algunos tienen banderas desplegables, como se muestra en la Figura 1.4.

Figura 1.4: Distintos diseños de robots de sumo.

Los sensores utilizados por los robots para detectar el piso blanco o negro del dojo son normalmente sensores reflexivos. También, se utilizan regularmente sensores infrarrojos o de ultrasonido para detectar al robot oponente y calcular la distancia en la que se encuentra. Como dispositivo de control normalmente se utiliza un micro-controlador para controlar los motores en base a las mediciones de los sensores del robot.

Hasta la fecha, en las competencias estatales no se han observado la participación de robots móviles de sumo que utilicen cámaras para ver su ambiente y poder tener un

mejor desempeño que los otros robots.

1.3. Objetivos de la tesis

El objetivo general de la tesis es:

- Desarrollar un robot móvil de sumo con la capacidad de ver su ambiente, el dojo, el robot oponente y dirigir sus acciones para empujar al otro robot fuera del dojo.

El prototipo desarrollado del robot móvil de sumo se muestra en la Figura 1.5. El robot móvil con sistema de tracción tipo oruga tiene dos bandas de tracción accionadas por motores independientes, incluye una pequeña computadora a bordo y dispone de una pequeña cámara situada en la parte superior, dirigida hacia abajo. Para darle al robot la capacidad de ver la mayor parte del dojo, se utiliza un lente de muy amplia apertura, conocido también como ojo de pez (en inglés "Fish Eye").

Figura 1.5: El prototipo de robot de sumo desarrollado.

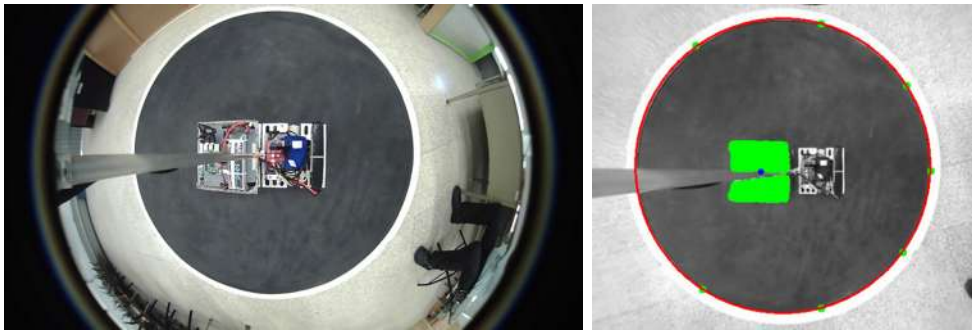
Para lograr el objetivo general se deben alcanzar los siguientes objetivos específicos:

- Calibración de la cámara, removiendo la gran distorsión introducida por el lente ojo de pez.
- Detección del dojo, en especial el círculo blanco que marca el final del tablero.
- Detección del robot oponente dentro del dojo.
- Estrategia de combate en base a la ubicación del robot y su oponente.

La Figura 1.6 ilustra el proceso que se sigue. La Figura 1.6 (a) muestra una imagen capturada por la cámara del robot, donde se puede apreciar el tablero y el robot oponente justo al lado izquierdo del robot que tiene la cámara. En la imagen se puede observar la gran distorsión que introduce el lente ojo de pez, en especial en las zonas alejadas del centro de la imagen.

La Figura 1.6 (b) muestra una imagen de los resultados finales de este trabajo de tesis donde ya se ha corregido la distorsión introducida por el lente, la circunferencia de color rojo muestra el inicio de la zona marcada por el color blanco del tablero y en verde se marca el robot oponente. Con esta información el robot está en condiciones de realizar las acciones de movimiento adecuadas para empujar al oponente.

El robot está desarrollado sobre el sistema operativo Linux, el cual esta instalado en una computadora a bordo del robot, aprovechando la biblioteca de software libre OpenCV para el manejo de imágenes. Adicionalmente utiliza un micro controlador Arduino para el control de los dos motores del robot.



(a) Imagen de la cámara

(b) Ubicación del tablero y del robot oponente

Figura 1.6: Proceso de detección del tablero y robot oponente.

1.4. Organización de la tesis

El capítulo 2 muestra el modelo matemático de la cámara y la forma de estimar los parámetros de la cámara, con el fin de remover la distorsión introducida por el lente ojo de pez. En el capítulo 3 se aborda la estrategia para detectar el círculo blanco del dojo y determinar la ubicación del robot dentro del mismo. En el capítulo 4 se aborda la construcción de un modelo del piso negro del dojo y una estrategia para detectar al robot oponente. El robot móvil desarrollado se describe en el capítulo 5, incluyendo la interfaz para darle instrucciones de movimiento y la estrategia de combate que implementa el robot, en función de la ubicación de los dos robots en el dojo. Las pruebas de desempeño del robot desarrollado se describen en el capítulo 6. Finalmente las conclusiones y sugerencias para trabajos futuros se abordan en el capítulo 7.

Capítulo 2

Calibración de la cámara

En este capítulo se revisa el modelo matemático que permite asociar un punto tridimensional del ambiente, con el correspondiente píxel de la imagen capturada por la cámara. Al proceso de estimación de los parámetros de dicho modelo matemático, a partir de imágenes tomadas por la cámara, se le conoce como calibración de la cámara.

Enseguida se revisa el modelo Pin-hole de una cámara, incluyendo modelos de distorsión, y las funciones de OpenCV utilizadas para estimar los parámetros de los modelos.

2.1. Modelo Pin-hole y parámetros intrínsecos de la cámara

El modelo de cámara pin-hole podría llamarse en español como el modelo de pequeña perforación o de perforación de aguja. Una cámara pin-hole es una caja negra que tiene una pequeña apertura en uno de los lados, por donde entra la luz a la cámara. Cuando la luz pasa a través de la perforación se forma una imagen invertida del lado opuesto de la cámara como se ilustra en la figura 2.1. A la distancia f se le conoce como distancia focal y mide la distancia entre la apertura de la cámara y el plano donde se forma la imagen.

Sin embargo, para tener un tratamiento matemático más sencillo, se traslada el plano de la imagen al frente de la cámara, entre la escena y el pin-hole, a la misma distancia f; de manera que la imagen que se forma ya no es invertida, como se ilustra en la figura 2.2. De esta manera, el punto P del ambiente, con coordenadas (x_c, y_c, z_c) con respecto al sistema de referencia de la cámara $< X_c, Y_c, Z_c >$, se proyecta en el punto p de coordenadas (x_i, y_i) del plano de la imagen, situado a una distancia f del punto pin-hole, C_o, llamado centro óptico de la cámara. A la línea de Z_c se le conoce como eje óptico de la cámara.

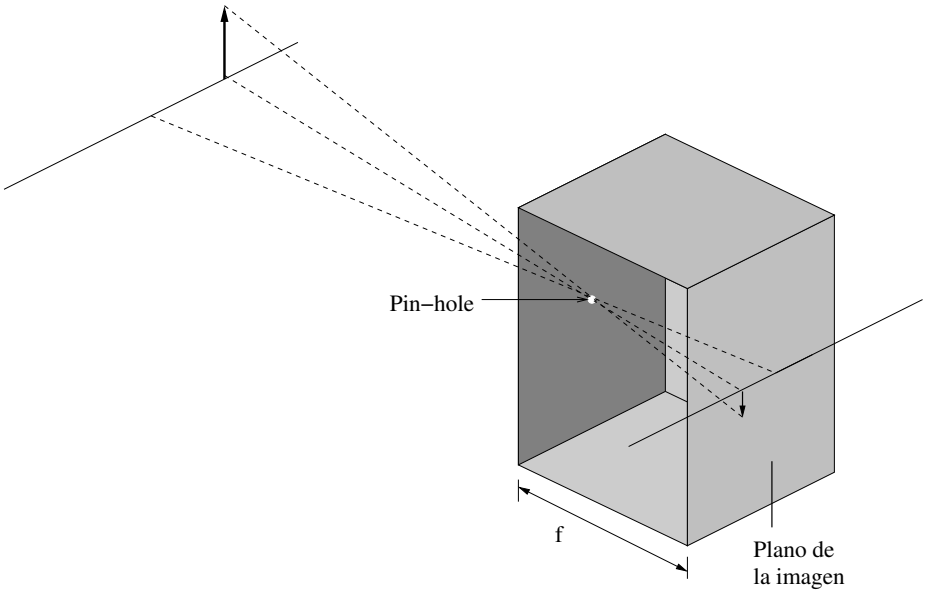


Figura 2.1: Modelo de cámara pin-hole.

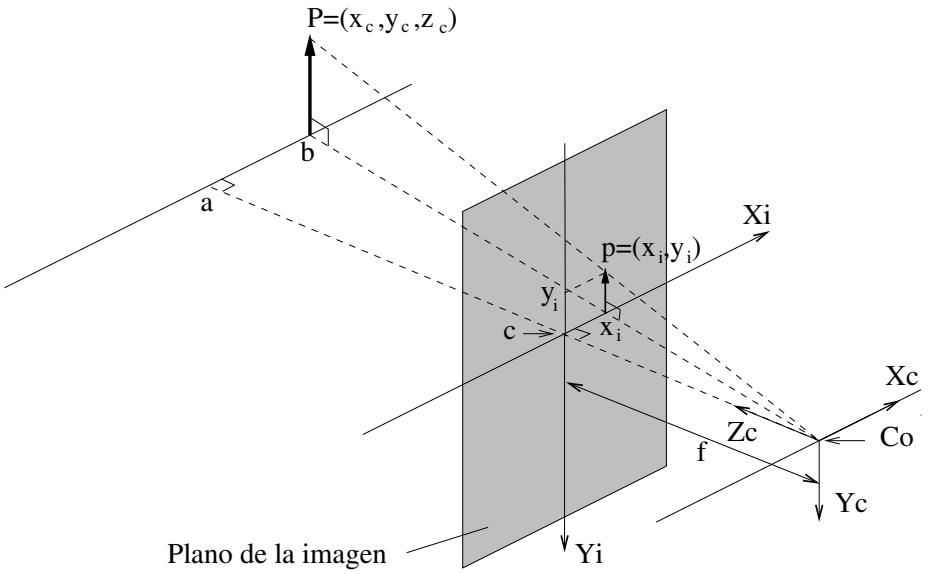


Figura 2.2: Modelo de cámara pin-hole simplificado.

De la figura podemos ver que los triángulos rectángulos $< C_o, a, b >$ y $< C_o, c, x_i >$ tienen la misma forma, es decir, son triángulos semejantes. De igual manera, los triángulos $< C_o, b, P >$ y $< C_o, x_i, p >$ son triángulos semejantes. Las figuras 2.3 y 2.4 muestran estas relaciones con mayor claridad.

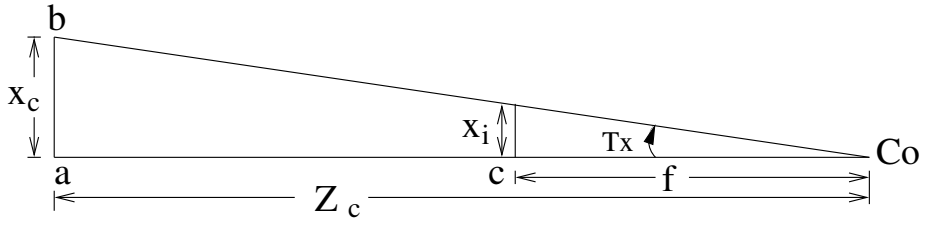


Figura 2.3: Triángulos semejantes $< C_o, a, b >$ y $< C_o, c, x_i >$.

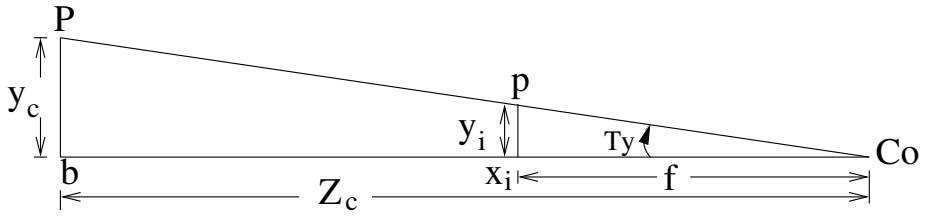


Figura 2.4: Triángulos semejantes $< C_o, b, P >$ y $< C_o, x_i, p >$.

Al ser triángulos semejantes se cumplen las siguientes relaciones:

$$\tan(T_x) = \frac{x_c}{z_c} = \frac{x_i}{f} \quad , \quad \tan(T_y) = \frac{y_c}{z_c} = \frac{y_i}{f} \tag{2.1}$$

Despejando las coordenadas x y la y de las ecuaciones anteriores, tenemos:

$$x_i = f\ \frac{x_c}{z_c} \quad , \quad y_i = f\ \frac{y_c}{z_c} \tag{2.2}$$

Ahora bien, el sensor de la cámara, situado en el plano de la imagen, contiene pequeños sensores de luz de forma rectangular o cuadrada, como se ilustra en la figura 2.5. De esta manera, cuando hablamos de una imagen tomada por una cámara, se utiliza el término de píxel para representar el valor medido por uno de estos sensores.

Sea el sistema de referencia $< U, V >$ como se indica en la figura 2.5, donde el píxel superior izquierdo tiene las coordenadas $u = 0$ y $v = 0$. El punto principal c, definido como la intersección del eje óptico de la cámara con el plano de la imagen, corresponde al píxel de la imagen con coordenadas (c_u, c_v). De esta manera, para calcular la correspondencia del punto $p = (x_i, y_i)$ en las coordenadas de la imagen $p = (u, v)$, las ecuaciones anteriores se transforman de la siguiente forma:

$$u = f_x\ \left(\frac{x_c}{z_c}\right) + c_u,\ \ v = f_y\ \left(\frac{y_c}{z_c}\right) + c_v \tag{2.3}$$

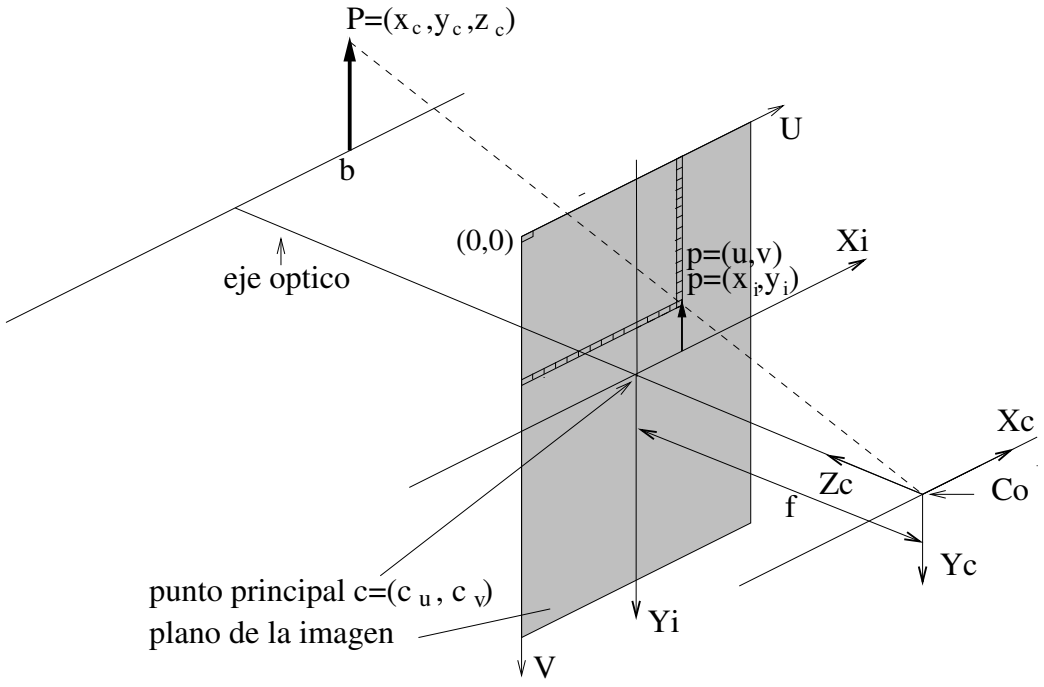


Figura 2.5: Modelo de la cámara utilizado en OpenCV [OpenCV_3D19].

Donde:

- (x_c, y_c, z_c) son las coordenadas de un punto P en el sistema de referencia de la cámara.

- f_x, f_y son las distancias focales en los ejes X_i y Y_i respectivamente, las cuales convierten (x_i, y_i) a coordenadas de píxel en la imagen.

- c_u, c_v son las coordenadas del punto principal en el sistema de referencia $< U, V >$ del sensor de la cámara.

Observe que la cámara tiene dos distancias focales, una para el eje X_i y la otra para el eje Y_i. Esto ocurre debido a las características específicas de los fotoreceptores del plano de la imagen, los cuales no necesariamente son cuadrados. Si la cámara tiene sensores cuadrados, entonces $f_x = f_y$.

La Ecuación 2.3 se puede pasar a coordenadas homogéneas y de esta forma se obtiene una expresión que involucra operaciones matriciales sencillas. Dicha representación

quedaría como sigue:

$$p^h = M_i P_c, \quad p^h = s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}, \quad M_i = \begin{bmatrix} f_x & 0 & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix}, P_c = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2.4}$$

Aquí p^h representa una posición en el plano de la imagen en coordenadas homogéneas, P_c un punto de 3 dimensiones en coordenadas del sistema de referencia de la cámara (que se proyecta al punto p en la imagen) y M_i una matriz que contiene los parámetros intrínsecos de la cámara: f_x, f_y, c_u y c_v.

Es conveniente recordar que un vector v^h en coordenadas homogéneas aumenta su dimensión en una unidad; así un vector de dos dimensiones se representa por un vector de dimensión 3 y un vector de 3 dimensiones se representa por un vector de dimensión 4. En coordenadas homogéneas los vectores v^h y $k\,v^h$ ($k \neq 0$, $k \in R$) corresponden al mismo vector.

Sea $p^h = [u^h\ v^h\ w]^t$ el vector p en coordenadas homogéneas, el cual se obtiene al multiplicar la matriz M_i por el vector P_c. Es decir,

$$u^h = f_x x_c + c_u z_c \quad , \quad v^h = f_y y_c + c_v z_c \quad , \quad w = z_c$$

Para obtener el vector p^h en la forma $s\ [u\ v\ 1]^t$ basta multiplicar el vector p^h por $1/w = 1/z_c$ y hacer $s = w$. Es decir:

$$p^h = s\ [u\ ,\ v\ ,1]^t \tag{2.5}$$

$$p^h = w\ [u^h/w\ ,\ v^h/w\ ,1]^t \tag{2.6}$$

Se puede comprobar fácilmente que $u = u^h/w$ y $v = v^h/w$ corresponden a los valores correspondientes de la ecuación 2.3.

En resumen, la matriz M_i contiene los parámetros intrínsecos o inherentes a la cámara: las distancias focales, f_x y f_y; y las coordenadas del punto principal de la cámara, c_u y c_v.

Conviene mencionar que este modelo de pin-hole es un modelo ideal, donde no hay ningún tipo de distorsión. Sin embargo, en la realidad, las cámaras presentan ciertas distorsiones que se abordarán más adelante. Enseguida se revisa el efecto de la distancia focal en la cámara, como preparación para abordar los tipos de distorsión involucrados con distancias focales pequeñas y grandes.

2.1.1. Efecto de la distancia focal en la cámara

Es conveniente tener en cuenta el efecto de la distancia focal y las partes de la escena que pueden ser observadas por la cámara. Las figuras 2.6 y 2.7 muestran el efecto para una distancia focal pequeña y grande, respectivamente, manteniendo del mismo tamaño el plano de la imagen que corresponde al sensor de la cámara.

Para una distancia focal pequeña se puede observar que la cámara puede captar una gran porción de la escena que tiene enfrente, mientras que una cámara con distancia focal grande permite captar sólo una pequeña porción de la escena.

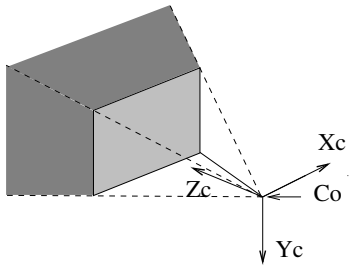


Figura 2.6: Efecto de una distancia focal pequeña.

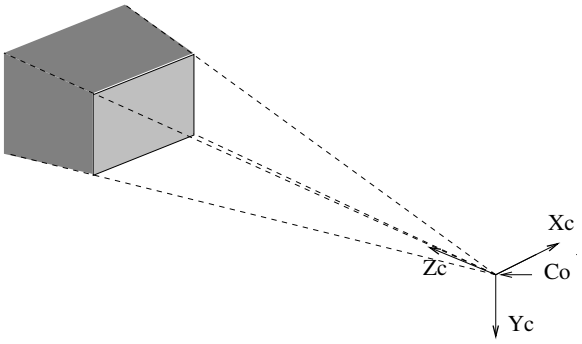


Figura 2.7: Efecto de una distancia focal grande.

Cámaras con distancias focales pequeñas, por ejemplo de 10mm, se utilizan mucho en robótica móvil para que el robot pueda observar los objetos que tiene enfrente. Por ejemplo, una cámara con distancia focal de pocos milímetros podría tener una apertura horizontal de 90 grados y una apertura vertical de 60 grados.

Por otro lado, las cámaras con distancia focales grandes, por ejemplo de 1000mm, se utilizan para captar objetos muy distantes, por ejemplo algún animal localizado a cientos

de metros de distancia de la cámara, de manera que las aperturas horizontal y vertical son de pocos grados.

2.2. Parámetros extrínsecos de la cámara

En la sección anterior vimos que M_i contiene los parámetros que tienen que ver con la cámara. Sin embargo, es común que los puntos P del ambiente estén referidos a otro sistema de referencia diferente del sistema de referencia de la cámara $< X_c, Y_c, Z_c >$, llamado sistema de referencia global $< X_g, Y_g, Z_g >$ como se ilustra en la figura 2.8. Por ejemplo, el sistema de referencia global podría corresponder a un sistema de referencia ubicado en el centro de un robot móvil y la cámara podría tener una determinada ubicación y orientación con respecto a ese sistema de referencia global.

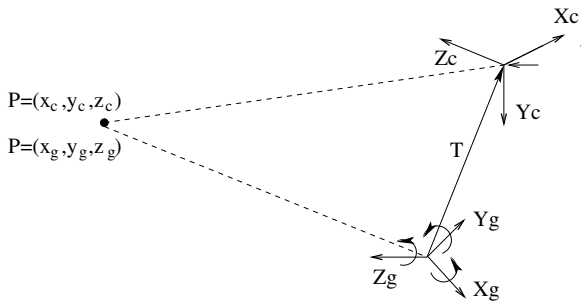


Figura 2.8: El punto P con respecto a los sistemas de referencias de la cámara y global.

Si tenemos un punto P en coordenadas (x_g, y_g, z_g) con respecto a un marco de referencia global, se le puede aplicar una transformación de rotación R, para alinear el sistema de referencia $< X_g, Y_g, Z_g >$ con el sistema de referencia de la cámara $< X_c, Y_c, Z_c >$. Esto se ilustra en la figura 2.8 mediante giros sobre los ejes X_g, Y_g y Z_g.

Una vez alineados ambos sistemas de referencia, es suficiente aplicar una operación de traslación T para que el marco de referencia global modificado coincida con el marco de referencia de la cámara. En la figura 2.8 el vector T denota esta traslación.

Esta rotación y traslación se puede expresar matemáticamente como sigue:

$$P_c = R(P_g + T) \tag{2.7}$$

Donde:

- P_g es un punto P expresado en coordenadas $(x_g, y_g, z_g)^t$ con respecto al marco de referencia global.
- P_c es el punto expresado en las coordenadas $(x_c, y_c, z_c)^t$ con respecto al marco de referencia de la cámara.
- R es una matriz ortonormal de dimensión 3×3 que modela las transformaciones de rotación. La matriz R también puede obtenerse como el producto de las rotaciones de los ejes, es decir $R = R_x R_y R_z$, donde R_i $(i = x, y, z)$ es una matriz que representa el giro alrededor del eje i.
- T es un vector de dimensión 3×1 que modela el vector de traslación que une los centros de los sistemas de referencia global y de la cámara.

Si cambiamos a coordenadas homogéneas, el punto P_g^h se expresará como $P_g^H = (x_g, y_g, z_g, 1)^t$ y el punto P_c^H como $(x_c^h, y_c^h, z_c^h, w_c)^t$. Usando la matriz R y el vector T, P_c^h se puede expresar como sigue:

$$P_c^h = \begin{bmatrix} R & T \\ 0^T & 1 \end{bmatrix} P_g^h \tag{2.8}$$

Si desglosamos la matriz R y el vector T en sus componentes, tendremos:

$$P_c^h = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_g \\ y_g \\ z_g \\ 1 \end{bmatrix} \tag{2.9}$$

Combinando esta ecuación con la Ecuación 2.4, podemos obtener una ecuación reducida que integra también los parámetros intrínsecos de la cámara.

$$\begin{bmatrix} u^h \\ v^h \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_u \\ 0 & f_y & c_v \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \end{bmatrix} \begin{bmatrix} x_g \\ y_g \\ z_g \\ 1 \end{bmatrix} \tag{2.10}$$

$$p^h = M_i \, M_e \, P_g^h \tag{2.11}$$

Conviene tener en cuenta que las coordenadas reales u y v en la imagen se pueden calcular fácilmente como: $u = u^h/w$ y $v = v^h/w$. De esta manera M_i es la matriz de parámetros

intrínsecos de la cámara, aquellos que no cambian al usar la cámara; mientras que M_e es la matriz de parámetros extrínsecos de la cámara, aquellos que definen las rotaciones y traslación del marco de referencia global con respecto al marco de referencia de la cámara. Si se mueve la cámara, la matriz M_e también cambiará.

Este modelo matemático no considera las distorsiones que se producen al utilizar cámaras con diferentes tipos de lentes. El modelado de estas distorsiones se abordarán enseguida.

2.3. Modelos de distorsión de las cámaras

Aunque existe cámaras reales que siguen el principio del modelo de pin-hole, es decir sólo tienen una pequeña apertura donde donde pasa la luz; en la práctica dichas cámaras no son utilizadas debido a que requieren una gran cantidad de luz en la escena, para lograr que se ilumine el plano de la imagen de manera adecuada.

Las cámaras utilizan lentes para poder captar una mayor cantidad de luz del ambiente. Aunque en teoría es posible definir lentes que no introducirán distorsión, en la práctica esto no es así, debido a las características propias de los lentes y al proceso de fabricación de la cámara.

Un tipo muy importante de distorsión, debido a la utilización de los lentes, se le conoce como distorsión radial. La Figura 2.9 muestra dos tipos de distorsión radial muy frecuentes. En la imagen de la izquierda no se observa distorsión, mientras que en la imagen central se observa un tipo de distorsión radial positiva que ocurre frecuentemente con cámaras que tienen lentes con distancias focales muy pequeñas. Por otro lado, en la imagen derecha se observa un tipo de distorsión radial negativa que se presenta frecuentemente con cámaras que tienen lentes con distancias focales muy grandes.

En ambos casos, el punto principal c de la imagen (ubicado en el centro de la imagen con coordenadas c_u y c_v) no sufre distorsión. Sea r el radio que une un píxel cualquiera q de la imagen sin distorsión al punto c, en el caso de distorsiones positivas, el píxel q se desplaza sobre la línea que une q con c, acercándose un poco hacia el punto c. En la imagen central de la figura 2.9 este desplazamiento se observa fácilmente en los píxeles cercanos a las esquinas de la imagen. Mientras más alejados estén del punto c, el desplazamiento hacia el punto c es mayor.

En contraste, para el caso de distorsiones negativas, el píxel q se desplaza sobre la

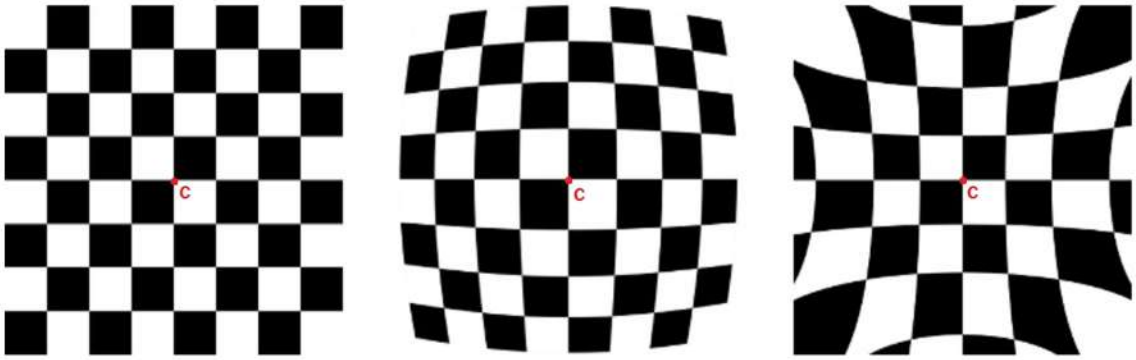
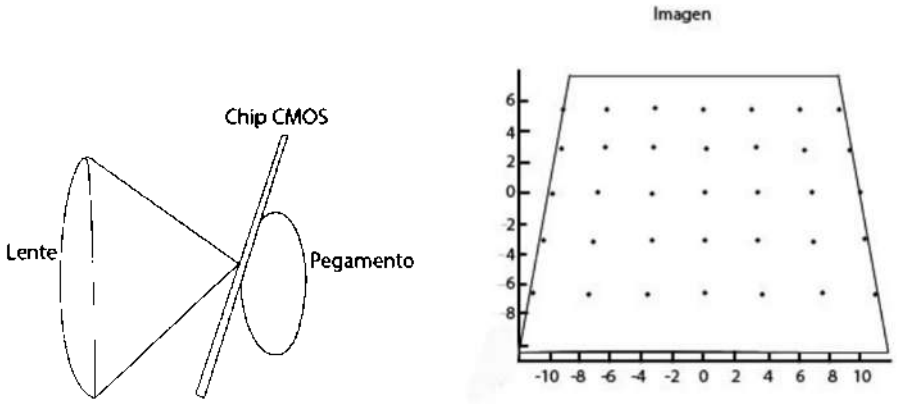


Figura 2.9: Ejemplos de distorsiones radiales [OpenCV_3D19].

línea que une a q con c, alejándose un poco del punto c. En la imagen derecha de la figura 2.9 este desplazamiento es más notorio en los píxeles de las esquinas de la imagen.

La distorsión radial es pequeña o nula en la vecindad del punto principal de la imagen y se incrementa a medida que los píxeles se alejan del punto principal.

Otro tipo de distorsión que se presenta en las cámaras es conocida como distorsión tangencial. Esta distorsión se genera cuando no existe una alineación perfecta del lente y el receptor de imagen, como se ilustra en la figura 2.10. La distorsión tangencial se debe a defectos de fabricación de la cámara, como resultado de que los lentes no están exactamente paralelos al plano de la imagen.



(a) Vista interna del ensamble de la cámara (b) Plano de la imagen con distorsión tangencial

Figura 2.10: Ejemplo de la distorsión tangencial en una cámara [Bradski08].

En resumen, la distorsión radial se crea como resultado de la forma del lente y la distorsión tangencial se crea como resultado del proceso de ensamblado de la cámara completa.

Enseguida se revisa el modelo estándar de distorsión que utiliza OpenCV y uno nuevo que se adapta mejor a distorsiones muy grandes, cuando se utilizan lentes con distancia focal muy pequeña, conocidos como lentes de ojo de pez (en inglés "Fish-eye").

2.3.1. Modelo estándar de distorsión de los lentes en OpenCV

Sea $[x_c, y_c, z_c]^t = M_e P_g^h$ las coordenadas del punto P_c en el marco de referencia de la cámara. Las nuevas coordenadas $< u, v >$ en la imagen se obtienen como sigue [OpenCV_3D19]:

$$x' = x_c/z_c \tag{2.12}$$

$$y' = y_c/z_c \tag{2.13}$$

$$r^2 = (x')^2 + (y')^2 \tag{2.14}$$

$$x'' = x' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + 2p_1 x' y' + p_2(r^2 + 2(x')^2) \tag{2.15}$$

$$y'' = y' \frac{1 + k_1 r^2 + k_2 r^4 + k_3 r^6}{1 + k_4 r^2 + k_5 r^4 + k_6 r^6} + p_1(r^2 + 2(y')^2) + 2p_2 x' y' \tag{2.16}$$

$$u = f_x x'' + c_u \tag{2.17}$$

$$v = f_y y'' + c_v \tag{2.18}$$

Donde: k_i(i=1,...,6) son los coeficientes de la función polinomial que modelan la distorsión radial y p_1 y p_2 modelan la distorsión tangencial.

En varios textos se describen muchos métodos para calcular los parámetros intrínsecos y extrínsecos de una cámara, sea por separado o combinando los parámetros en una sola matriz. Sin embargo, para desarrollar rápidamente software de visión computacional una buena estrategia es utilizar los algoritmos de calibración implementados en OpenCV.

OpenCV [OpenCV_3D19] utiliza un algoritmo basado en el método de Zhang [Zhong99] para detectar las distancias focales y los centros ópticos, pero utiliza el método basado en el método de Bouguet [Bouguet15] (en OpenCV 2) para obtener los parámetros de los coeficientes de distorsión radial y tangencial.

Para calibrar la cámara OpenCV utiliza una función llamada cv::calibrateCamera, la cual estima los parámetros intrínsecos y extrínsecos de la cámara y los coeficientes de distorsión, a partir de imágenes de un patrón conocido, como un tablero de ajedrez. Esta función se describe brevemente enseguida.

OpenCV proporciona la siguiente función en C++ para calibrar una cámara:

```
double calibrateCamera (InputArrayOfArrays objectPoints, InputArrayOfArrays
imagePoints, Size imageSize, InputOutputArray cameraMatrix, InputOutputArray
distCoeffs, OutputArrayOfArrays rvecs, OutputArrayOfArrays tvecs, int flags=0,
TermCriteria criteria = TermCriteria( TermCriteria::COUNT+TermCriteria::EPS,
30, DBL\_EPSILON) )
```

Esta función regresa el error de re-proyección y en un inicio los parámetros de entrada/salida pueden parecer muchos pero al describirlos pueden ya no parecer tan abrumadores:

- objetctPoints: Este parámetro es de entrada y se refiere a los puntos 3D del objeto que se presenta a la cámara, con respecto al sistema de referencia global. Es una matriz de $(N * K) \times 3$ números, donde N es el número de vistas o imágenes del objeto y K el número de puntos del patrón utilizado. El 3 se refiere a que trata de puntos 3D, con coordenadas: (x_g, y_g, z_g).

- imagePoints: Este parámetro es de entrada y se refiere a las esquinas encontradas en las imágenes en coordenadas (u, v) y es una matriz de $(N*K)\times 2$, donde cada punto 2D en la imagen tiene su correspondencia con un punto 3D del parámetto objectPoints.

- imageSize: Es el tamaño de la imagen en píxeles.

- cameraMatrix: Es la matriz de los parámetros intrínsecos M_i de la cámara, puede ser una matriz de entrada/salida, si se proporciona como entrada se utiliza para calcular los coeficientes de distorsión y si no, se calcula y se regresa. Es una matriz de 3×3.

- distCoeffs: Es un vector de coeficientes de distorsión de la cámara. Es una vector de 4, 5 u 8 elementos: $(k_1, k_2, p_1, p_2, [k_3, [k_4, k5, k6]])$ en ese orden.

- rvecs: Es una matriz de salida que contiene los vectores de rotación para cada objeto 3D. El tamaño de esta matriz depende del número de vista o imágenes del objeto.

- tvecs: Es una matriz de salida que contiene los vectores de traslación para cada objeto 3D. Al igual que *rvecs*, el tamaño de la matriz depende del número de tomas. Con rvecs y tvecs se pueden obtener las N matrices extrínsecas M_e, una por cada vista o imagen.

- flags: Son las banderas que se pueden utilizar para calibrar la cámara. Por defecto se inicializa en 0.

- criteria: Es una variable que indica cuando debe detenerse el algoritmo, por defecto se inicializa con un número de iteraciones de 30 como máximo y un umbral de épsilon de DBL_EPSILON.

2.3.2. Modelo Fisheye de OpenCV

En cámaras que tienen una lente ojo de pez (con aperturas mayores de 160 grados), el modelo clásico de OpenCV expuesto anteriormente, puede fallar. Por esta razón, desde la versión 3.0.0 de OpenCV se ha incluido el paquete cv2::fisheye, especialmente dedicado a modelar la distorsión de cámaras con lentes de ojo de pez [Jiang17].

Sea $[x_c, y_c, z_c]^t = M_e P_g^h$ las coordenadas del punto P en el marco de referencia de la cámara. Las nuevas coordenadas $< u, v >$ en la imagen se obtienen como sigue [OpenCV19c]:

$$x' = x_c/z_c \tag{2.19}$$

$$y' = y_c/z_c \tag{2.20}$$

$$r^2 = (x')^2 + (y')^2 \tag{2.21}$$

$$r' = atan(r) \tag{2.22}$$

$$r'_d = r'(1 + k_1(r')^2 + k_2(r')^4 + k_3(r')^6 + k_4(r')^8) \tag{2.23}$$

$$x'' = \frac{r'_d}{r}\, x' \tag{2.24}$$

$$y'' = \frac{r'_d}{r}\, y' \tag{2.25}$$

$$u = f_x(x'' + \alpha y'') + c_u \tag{2.26}$$

$$v = f_y y'' + c_v \tag{2.27}$$

Donde: k_i $(i = 1, \ldots, 4)$ son los coeficientes que modelan la distorsión radial y α

modela la distorsión tangencial. Conviene mencionar que en las ecuaciones mostradas en [OpenCV19c], en lugar de r' se utiliza θ, en lugar de r'_d se utiliza θ_d, en lugar de x' se utiliza a y en lugar de y' se utiliza b. Sin embargo se hicieron estos cambios por continuidad de los símbolos utilizados en el modelo de la sección anterior.

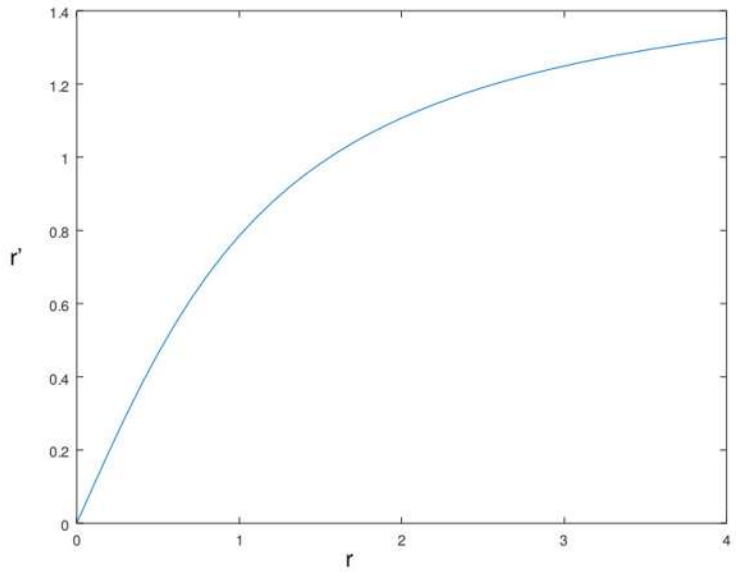


Figura 2.11: Gráfica de la función $r' = atan(r)$.

Conviene observar que debido a que $r \geq 0$, tenemos que $0 \leq r' < \pi/2$. Por otro lado, la derivada de r' con respecto a r está dada por:

$$\frac{d(r')}{r} = \frac{1}{1+r^2}$$

Esto indica que, para valores pequeños de r, tenemos que esta derivada es cercana a la unidad y por lo tanto $r' \approx r$, de manera que $x'' \approx x'$ y $y'' \approx y'$; es decir, se tiene una distorsión pequeña (asumiendo que $k_i = 0$, $i = 1, \ldots, 4$). La figura 2.11 muestra a r' como función de r, donde se puede apreciar la relación entre r y r'.

Para calibrar la cámara OpenCV utiliza una función llamada cv::fisheye::calibrate, la cual estima los parámetros intrínsecos y extrínsecos de la cámara y los coeficientes de distorsión, a partir de imágenes de un patrón conocido, como un tablero de ajedrez. Esta función se describe enseguida.

OpenCV proporciona la siguiente función en C++ para calibrar una cámara usando este modelo:

```
double cv::fisheye::calibrate (InputArrayOfArrays objectPoints,
InputArrayOfArrays imagePoints, Size \& imageSize\_size,
InputOutputArray K, InputOutputArray D, OutputArrayOfArrays rvecs,
OutputArrayOfArrays  tvecs,        int flags = 0, TermCriteria criteria =
TermCriteria(TermCriteria::COUNT+TermCriteria::EPS, 100, DBL\_EPSILON))
```

Los parámetros objetcsPoints, imagePoints, imageSize, rvecs, tvecs, flags y criteria, son los mismos que para la función calibrateCamera descrita anteriormente. Los parámetros nuevos son:

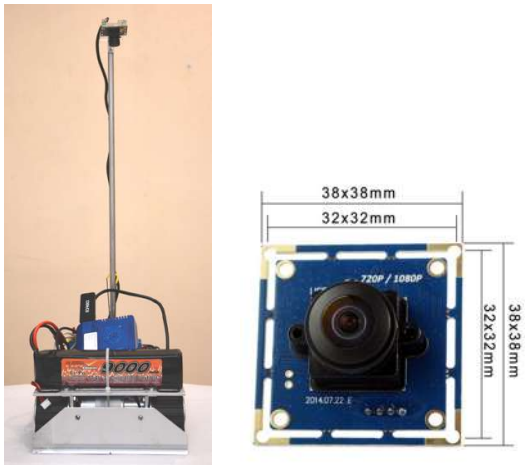
- K: Es la matriz de los parámetros intrínsecos M_i de la cámara, puede ser una matriz de entrada/salida, si se proporciona como entrada se utiliza para calcular los coeficientes de distorsión y si no, se calcula y se regresa. Es una matriz de 3×3.
- D: Es un vector de salida que contiene los 4 coeficientes de distorsión de la cámara: (k_1, k_2, k_3, k_4), en ese orden.

2.4. Ejemplo de calibración de una cámara con lente ojo de pez

En la figura 2.12 se muestra el robot de sumo que utiliza una cámara USB modelo OV2710 [ELP19] de la marca ELP, con una lente ojo de pez (apertura de 180 grados). La cámara tiene un sensor de 2 Megapíxeles y es capaz de adquirir 30 imágenes por segundo con resolución de 1920×1080. La cámara se coloca en la parte superior del robot, orientada hacia abajo, con la propósito de que pueda ver la mayor parte del tablero donde se realiza el combate.

El tablero de calibración se construyó uniendo hojas tamaño carta de color blanco y negro, con la finalidad de ubicar la cámara a una distancia similar a la distancia que tendría en el robot, para ver el tablero de combate. La figura 2.13(a) muestra una imagen del tablero de calibración, tomada por la cámara.

Para detectar los puntos esquina en la imagen se utiliza la función de la biblioteca de OpenCV cv::findChessboardCorners y la función cv::cornerSubPix para mejorar la detección de las esquinas a nivel subpíxel. La figura 2.14(a) muestra el resultado de la función



(a) El robot. (b) La cámara OV2710.

Figura 2.12: El robot de sumo con su cámara en la parte superior.

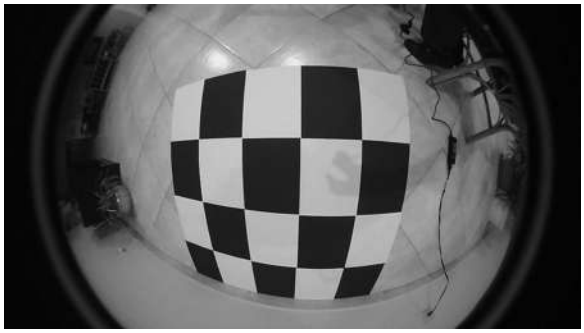
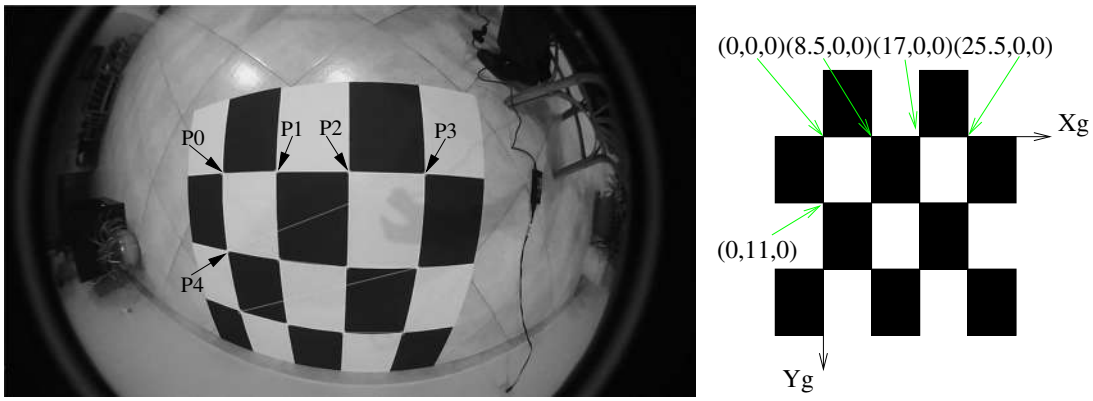


Figura 2.13: Imagen del tablero de calibración.

cv::drawChessboardCorners, que sirve para dibujar la secuencia de puntos detectados. A dicha imagen se le añadieron las indicaciones de las primeras esquinas detectadas: $P0$, $P1$, $P2$, $P3$ y $P4$. En total son 12 esquinas detectadas en el tablero.

Por otro lado, la figura 2.14(b) muestra los correspondientes puntos 3D en el sistema de referencia global. Las unidades están en pulgadas, asumiendo que se utilizaron hojas tamaño carta de 8.5" $\times$ 11" y la coordenada $z = 0$ en todos los puntos 3D.

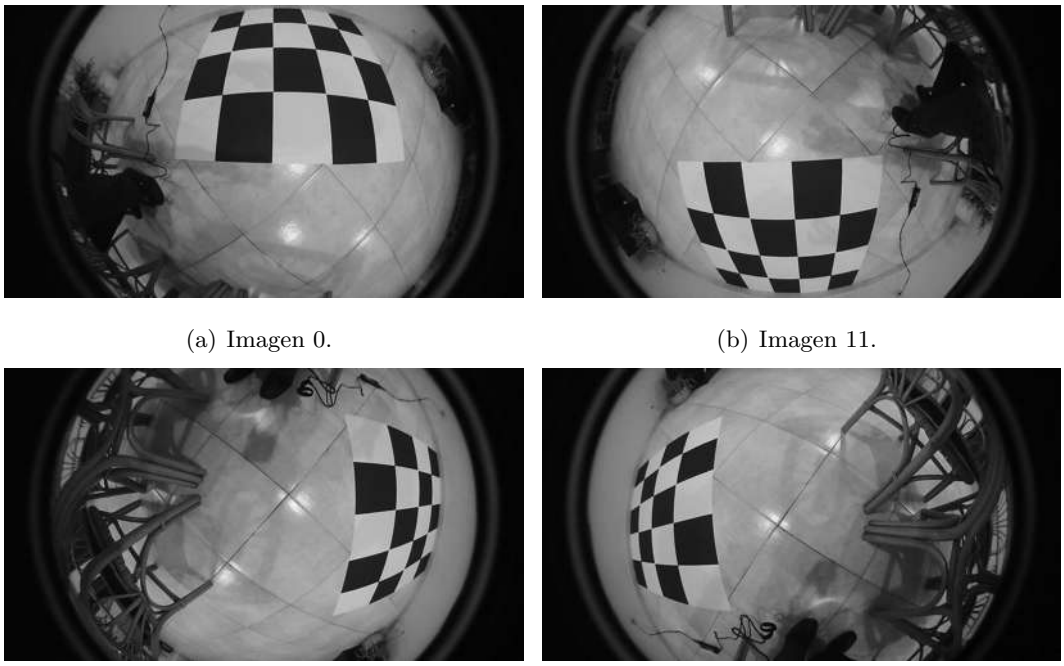
Para realizar la calibración se utilizaron 29 imágenes del tablero en total, cambiando la ubicación y y orientación de la cámara, de modo que el tablero de calibración apareciera en distintas zonas de la imagen. En la figura 2.15 se muestran algunas de estas imágenes.



(a) Esquinas detectadas en la imagen.

(b) Puntos 3D en coordenadas globales.

Figura 2.14: Puntos 2D en la imagen y correspondientes puntos 3D globales.



(a) Imagen 0.

(b) Imagen 11.

(c) Imagen 21.

(d) Imagen 28.

Figura 2.15: Ejemplos de imágenes utilizadas para la calibración.

2.4.1. Calibración usando el modelo estándar de distorsión

Se utilizaron las siguientes macros en el parámetro *flags* de la función de la biblioteca de OpenCV cv::calibrateCamera:

```
CV_CALIB_ZERO_TANGENT_DIST | CV_CALIB_FIX_ASPECT_RATIO | CV_CALIB_RATIONAL_MODEL
```

La macro CV_CALIB_ZERO_TANGENT_DIST tiene el efecto de fijar y mantener $p1 = 0$ y $p2 = 0$. La macro CV_CALIB_FIX_ASPECT_RATIO tiene el efecto de mantener $f_x = f_y$. La macro CV_CALIB_RATIONAL_MODEL tiene el efecto de habilitar la búsqueda de los parámetros k_4, k_5 y k_6, de manera que el vector de distorsión regresa 8 coeficientes, en lugar de los 5 coeficientes que de otra forma se regresan $(k_1, k_2, p_1, p_2, k_3)$.

También se cambio el parámetro *criteria*, para aumentar a 100 el número máximo de iteraciones:

```
cv::TermCriteria(cv::TermCriteria::MAX_ITER + cv::TermCriteria::EPS, 100,0.00001)
```

De esta forma se obtuvo una buena calibración con un error de reproyección de 0.699667 (este valor lo devuelve la función cv::calibrateCamera). La matriz de parámetros intrínsecos y los coeficientes de distorsión obtenidos se muestran en las tablas 2.1 y 2.2, respectivamente.

455.879	0.	959.37
0.	455.879	517.757
0.	0.	1.

Tabla 2.1: Parámetros intrínsecos de la cámara.

k_1	0.318183954501045
k_2	0.01316290018049696
k_3	3.676999202673193e-05
k_4	0.5792580156543465
k_5	0.05738608593136076
k_6	0.0006978893561464275

Tabla 2.2: Coeficientes de distorsión de la cámara.

Corrigiendo la distorsión

Para corregir la distorsión en las imágenes se utiliza la función de la biblioteca de OpenCV cv::initUndistortRectifyMap, la cual calcula un par de matrices auxiliares map1 y map2, que son utilizadas en la función cv::remap. La matriz de la cámara y los coeficientes de distorsión calculadas en la función cv::calibrateCamera son los elementos indispensables

para llevar a cabo la remoción de la distorsión en las imágenes capturadas por la cámara. Enseguida se presentan los argumentos de las llamadas a cv::initUndistortRectifyMap y cv::remap:

```
cv::initUndistortRectifyMap(
      cameraMatrix,        // matriz de la cámara (paramétros intrínsecos)
      distCoeffs,          // coeficientes de distorsión
      cv::Mat(),           // rectificación opcional (ninguna)
      cv::Mat(),           // nueva matriz de la cámara (ninguna)
      cv::Size(1920,1080), //tamaño de la imagen sin distorsión
      CV_32FC1,            // type of output map
      map1, map2);         // las matrices de mapeo en X y en Y
cv::remap(
      imagen_entrada,      // imagen de entrada distorsionada
      imagen_salida,       // imagen de salida sin distorsión
      map1, map2,          // matrices de mapeo
      cv::INTER_LINEAR);   // tipo de interpolación
```

Es conveniente resaltar que a la función cv::initUndistortRectifyMap sólo se requiere llamar una única vez, para calcular las matrices map1 y map2.

En la figura 2.16 se presenta la imagen de salida de la función cv::remap, correspondiente a la imagen mostrada a la figura 2.13.

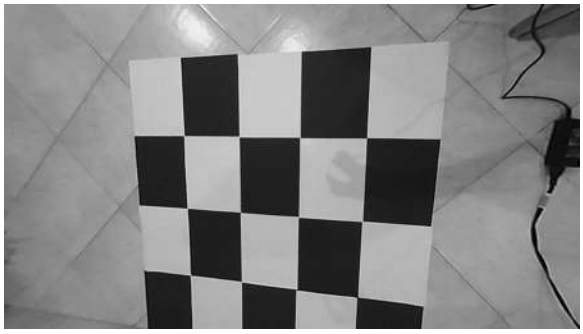


Figura 2.16: Imagen con la distorsión removida.

Se puede notar que al expandirse la imagen, hay una buena porción de la imagen de entrada que queda fuera de la imagen de salida. Una solución es ampliar el tamaño de la imagen de salida, por ejemplo de cv::Size(1920,1080) a cv::Size(1920*3,1080*3). En la

figura 2.17 se presenta la imagen de salida, con el tamaño incrementado por un factor de 3 en ambas direcciones.

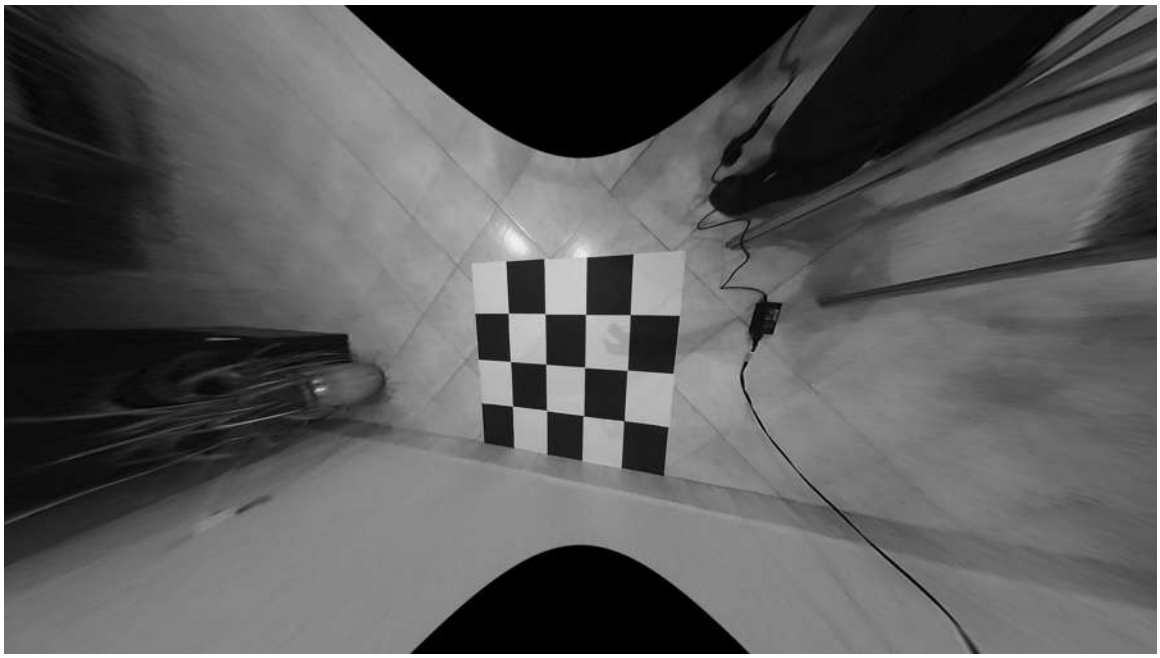


Figura 2.17: Imagen con la distorsión removida de tamaño $(1920 * 3 \times 1080 * 3)$.

Si bien esta nueva imagen contiene mayor información, su tamaño es muy grande para un procesamiento rápido de dicha imagen. Una mejor solución es conservar el tamaño de la imagen de salida en la misma resolución 1920×1080 e introducir una matriz de calibración con una distancia focal reducida, como quinto parámetro de la función cv::initUndistortRectifyMap. El siguiente fragmento de código fija las nuevas distancias focales en 1/3 de las originales. La imagen de salida es muy similar a la figura 2.17, pero con un tamaño de 1920×1080.

```
cv::Mat cm2;
cm2 = cameraMatrix.clone();
cm2.at<double>(0,0) /= 3.0;
cm2.at<double>(1,1) /= 3.0;
cv::initUndistortRectifyMap(cameraMatrix,distCoeffs,cv::Mat(), cm2,
  cv::Size(1920,1080), CV_32FC1,map1, map2);
```

2.4.2. Calibración usando el modelo Fisheye

Se utilizaron las siguientes macros en el parámetro *flags* de la función de la biblioteca de OpenCV cv::fisheye::calibrate:

```
cv::fisheye::CALIB_RECOMPUTE_EXTRINSIC | cv::fisheye::CALIB_FIX_SKEW
```

La macro CALIB_RECOMPUTE_EXTRINSIC tiene el efecto de recalcular los parámetros extrínsecos en cada optimización de los parámetros intrínsecos. La macro CALIB_FIX_SKEW tiene el efecto de fijar y mantener $\alpha = 0$, es decir, no se incluye distorsión tangencial. De esta forma se obtuvo una buena calibración con un error de reproyección de 0.664858 (este valor lo devuelve la función cv::fisheye:calibrate), un valor ligeramente mejor que en el caso del modelo estándar. La matriz de la cámara, K, y los coeficientes de distorsión, D, se muestran en las tablas 2.3 y 2.4, respectivamente.

518.867	0.	958.276
0.	517.198	517.963
0.	0.	1.

Tabla 2.3: Parámetros intrínsecos de la cámara en el modelo Fisheye.

k_1	-0.007672617058298234
k_2	-0.01693954398057557
k_3	0.0007337270998740699
k_4	0.000135666136846418

Tabla 2.4: Coeficientes de distorsión de la cámara en el modelo Fisheye.

Corrigiendo la distorsión

Para corregir la distorsión en las imágenes se utiliza la función de la biblioteca de OpenCV cv::fisheye::initUndistortRectifyMap, la cual calcula un par de matrices auxiliares map1 y map2, que son utilizadas en la función cv::remap. La matriz de la cámara K y los coeficientes de distorsión D calculadas anteriormente son los elementos indispensables para llevar a cabo la remoción de la distorsión en las imágenes capturadas por la cámara. Enseguida se presentan los argumentos de estas funciones:

```
cv::fisheye::initUndistortRectifyMap(
     K,                  // matriz de la cámara (paramétros intrínsecos)
```

```
        D,                    // coeficientes de distorsión
        cv::Mat(),            // rectificación opcional (ninguna)
        K,                    // nueva matriz de la cámara (ninguna)
        cv::Size(1920,1080), //tamaño de la imagen sin distorsión
        CV_32FC1,             // type of output map
        map1, map2);          // las matrices de mapeo en X y en Y
  cv::remap(
        imagen_entrada,       // imagen de entrada distorsionada
        imagen_salida,        // imagen de salida sin distorsión
        map1, map2,           // matrices de mapeo
        cv::INTER_LINEAR);    // tipo de interpolación
```

Es conveniente resaltar que a la función cv::fisheye::initUndistortRectifyMap sólo se requiere llamar una única vez, para calcular las matrices map1 y map2. En la figura 2.18 se presenta la imagen de salida de la función cv::remap, correspondiente a la imagen mostrada en la figura 2.13, de la página 24.

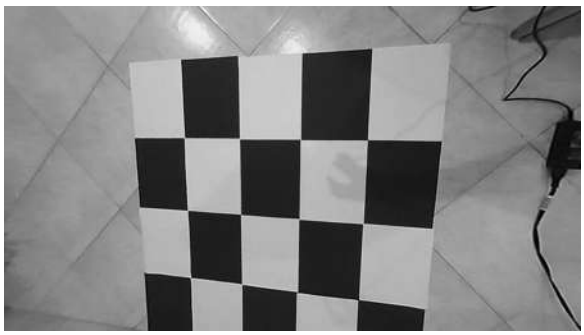


Figura 2.18: Imagen con la distorsión removida con el modelo Fisheye.

Esta imagen es muy similar a la obtenida en el modelo estándar de OpenCV. Sin embargo, para conservar el tamaño de la imagen de salida en la misma resolución 1920×1080 e introducir una matriz K2 con una distancia focal reducida, se tiene que llamar a la función cv::fisheye::estimateNewCameraMatrixForUndistortRectify, para generar las matrices U y P, las cuales se utilizan en la función cv::fisheye::initUndistortRectifyMap.

El siguiente fragmento de código fija las nuevas distancias focales usando un factor de 1/1.9 con respecto a las distancias focales originales.

```
cv::Mat K2;
K2 = K.clone();
```

```
K2.at<double>(0,0) /= 1.9;
K2.at<double>(1,1) /= 1.9;
cv::fisheye::estimateNewCameraMatrixForUndistortRectify(K2, D, cv::Size(1920,1080),
  U, P, 0.0, cv::Size(1920,1080), 1.0);
cv::fisheye::initUndistortRectifyMap(K,D,U,P,cv::Size(1920,1080),CV_32FC1,
  map1, map2);
```

En la figura 2.19 se presenta la imagen de salida de la función cv::remap, con las nuevas distancias focales.

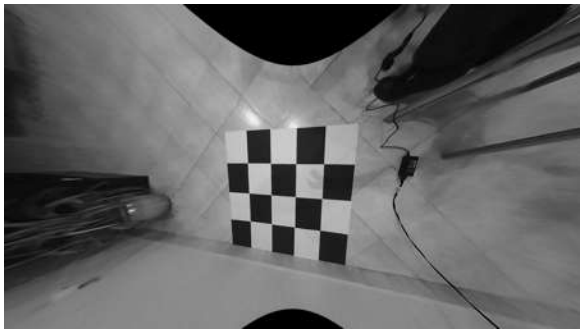


Figura 2.19: Imagen corregida con distancias focales reducidas por un factor de 1/1.9.

2.5. Generación de una vista desde arriba

Para la aplicación del robot de sumo, es conveniente generar una imagen que corresponda a una vista superior centrada en el robot, donde cada píxel represente un área cuadrada. En particular, se eligió tener píxeles que correspondan a cuadrados de $3mm \times 3mm$.

Para este propósito se colocó una cartulina blanca, de $700mm \times 500mm$, con pequeñas marcas negras en sus esquinas y el robot se colocó justo en la parte central de la cartulina. La figura 2.20 muestra la imagen tomada por la cámara del robot.

La figura 2.21 muestra la imagen correspondiente sin distorsión, utilizando el modelo de distorsión fisheye con las distancias focales reducidas por un factor de 1/1.9.

De esta imagen se obtuvieron las coordenadas de las cuatro esquinas de la cartulina y se calcularon los puntos correspondientes en una nueva imagen de tamaño 1080×700, asumiendo que cada píxel en la nueva imagen corresponde a un cuadro de $3mm \times 3mm$.

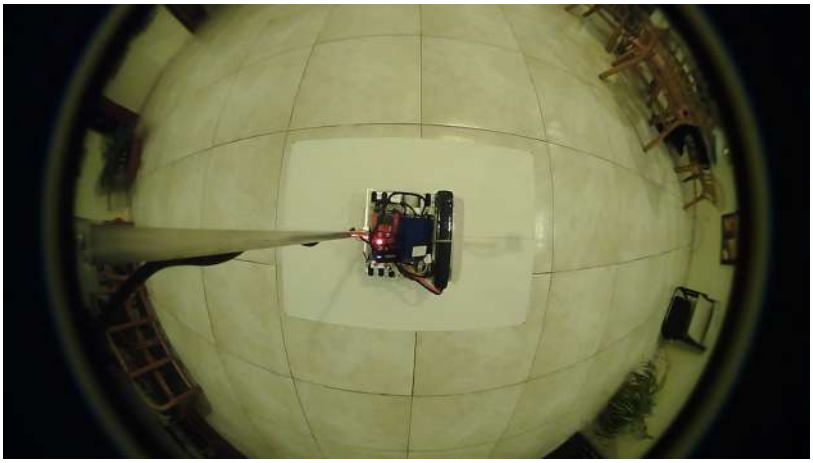


Figura 2.20: Imagen para calcular una vista superior.

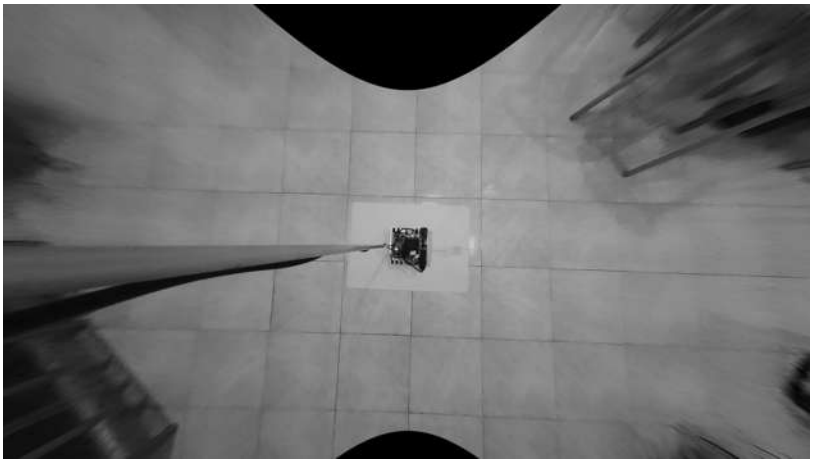


Figura 2.21: Imagen sin distorsión para calcular una vista superior.

Para calcular la matriz H de dimensión 3×3, que modela esta transformación entre las imágenes, a partir de los 4 pares de puntos entre las imágenes, se utilizó la función de OpenCV c::getPerspectiveTransform. Con la matriz H y la función cv::warpPerspective se generó la nueva imagen mostrada en la figura 2.22, de tamaño 1080×700.

Las parejas de imágenes mostradas en la figura 2.23 corresponden a imágenes cuando el robot tiene el tablero de combate al frente y al lado izquierdo. Puede observarse que la cámara es capaz de ver totalmente el tablero, si éste se ubica hacia adelante o hacia atrás del robot. Sin embargo, si el tablero está de lado y el robot está en los límites del tablero, existe una zona del tablero que no alcanza a ser capturada por la cámara.

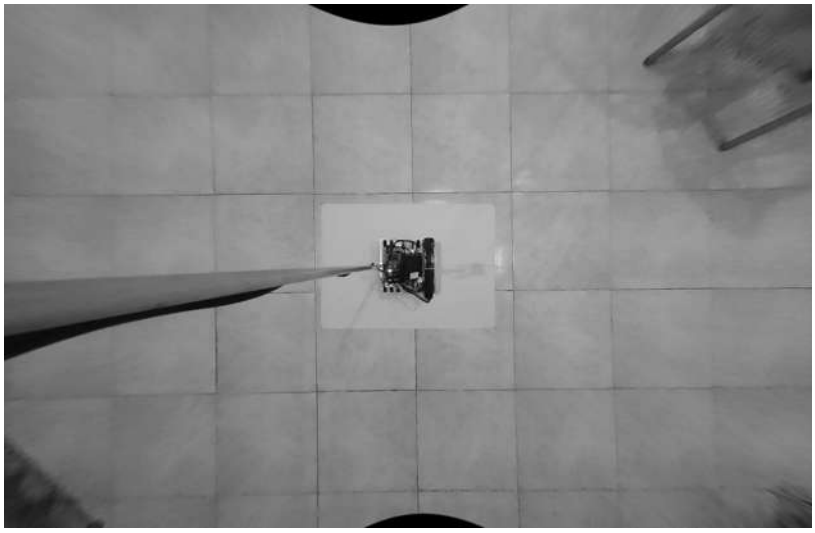
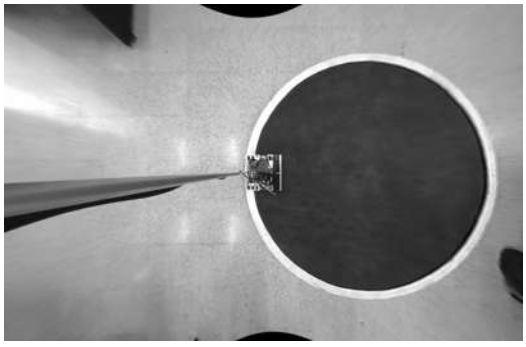
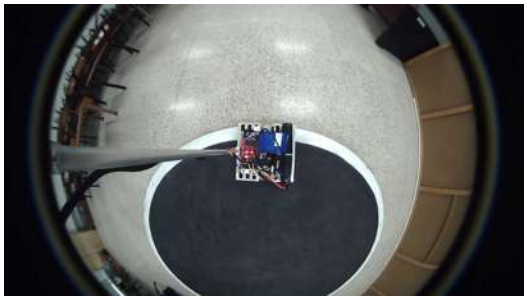


Figura 2.22: Vista superior donde cada píxel corresponde a un cuadro de $3mm \times 3mm$.

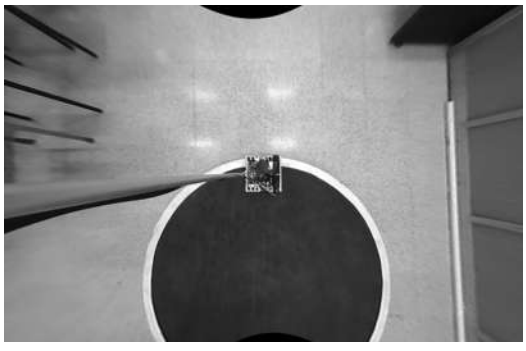
(a) Imagen con tablero al frente



(b) Vista superior



(c) Imagen con el tablero al lado



(d) Vista superior

Figura 2.23: Imágenes cuando el tablero está al frente y al lado de la cámara.

Las imágenes con distorsión radial obtenidas mediante la cámara fueron corregidas por medio de funciones de la librería OpenCV, obteniendo imágenes sin distorsión. Las vistas superiores calculadas a partir de las imágenes de la cámara serán el punto de partida para detectar la zona blanca del tablero de combate, así como el robot oponente, objetivos de los siguientes capítulos.

Capítulo 3

Detección del área de combate

En este capítulo se analiza el proceso llevado a cabo para la detección del dojo por medio de imágenes capturadas por la cámara instalada en el robot. Para realizar los cálculos más rápido, se trabaja con imágenes en escala de grises. También, se aborda una función de calibración de colores. Con esta función se obtiene un modelo matemático que sirve de apoyo para distinguir entre los dos colores del dojo (blanco y negro).

3.1. Modelo de color del área negra

En la Figura 3.1 se observa el histograma de la zona negra del área del dojo. El histograma se distribuye de tal forma que el eje horizontal representa el valor del píxel y el eje vertical representa la frecuencia en que se repite cierto valor. El intervalo del eje horizontal va desde 0 a 255, esto es de color negro a blanco. En el histograma se observa un pico que representa al color negro. El pico se asemeja a una distribución normal, lo se representan por medio de una media y una desviación estándar. Por lo tanto, la función de calibración calcula la media y la desviación estándar de una distribución normal para cada color.

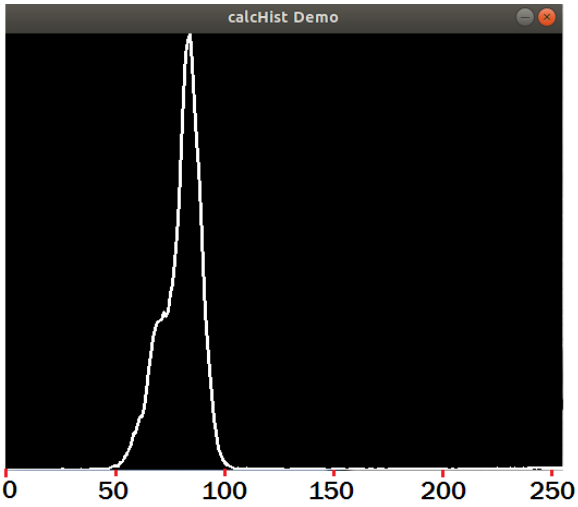


Figura 3.1: Histograma del dojo.

Según John E. Freud [Freud14], una variable aleatoria X tiene una distribución normal y se conoce como variable aleatoria normal si y solo si su densidad de probabilidad esta dada por:

$$n(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2}(\frac{x-\mu}{\sigma})^2} \qquad \text{Para } -\infty < x < \infty, \text{ donde } \sigma > 0 \tag{3.1}$$

Donde μ representa la media y σ la desviación estándar.
La gráfica de una distribución normal con media igual a μ y una desviación estándar unitaria se muestra en la Figura 3.2.

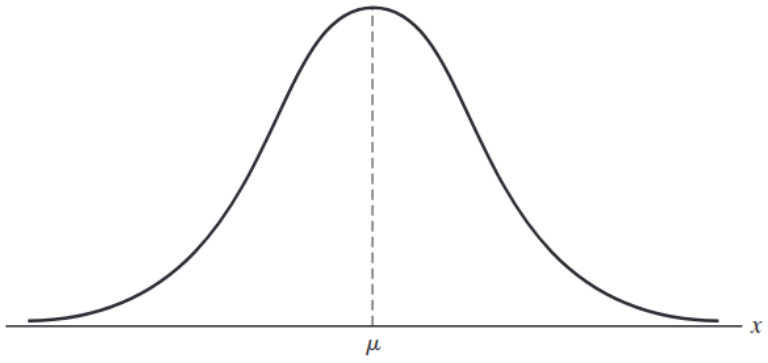


Figura 3.2: [Freud14] Gráfica de una distribución normal.

Los datos para calcular μ y σ se obtienen de una imagen, donde el robot se sitúa

en el centro del dojo. Estos datos se toman del área que rodea al robot, la cual se muestra de color azul en la Figura 3.3. Esta área de forma circular tiene un radio de menor magnitud que el dojo, de tal forma que se toman solamente píxeles que correspondan a la zona negra del dojo.

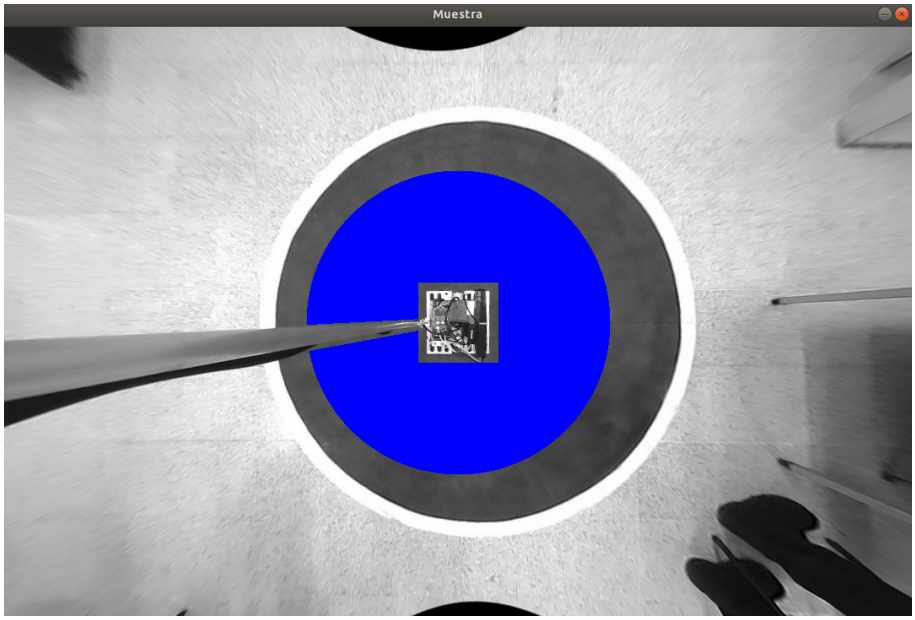


Figura 3.3: Área de muestra.

Entonces, la media se puede calcular como:

$$\mu = \frac{1}{N} \sum_{I \in i,j} x_{i,j} \tag{3.2}$$

Y la desviación estándar como:

$$\sigma = \sqrt{\frac{1}{N} \sum_{I \in i,j} (x_{i,j} - \mu)^2} \tag{3.3}$$

Donde N es la cantidad de píxeles que se encuentran en el área de muestra y $x_{i,j}$ es el valor del píxel en la coordenada (i, j) de la imagen I.
Por ejemplo, al calcular la media y desviación estándar del área azul de la imagen de la Figura 3.3, da como resultado:

$$\mu = 83 \qquad \text{y} \qquad \sigma = 7 \tag{3.4}$$

Al comparar estos resultados con lo que se muestra en la Figura 3.1, se puede observar que el resultado es aceptable ya que corresponde al pico de la izquierda que representa al color negro.
Es posible observar que píxeles de la imagen cumplen con este modelo por medio de la función de OpenCV llamada cv::inRange(), la cual recibe los siguientes parámetros:

```
void inRange(InputArray src, InputArray lowerb, InputArray upperb, OutputArray dst);
```

Esta función revisa que píxeles en **src** tienen un valor entre **lowerb** y **upperb**. Los píxeles que cumplen con esta condición se les asigna un valor de 255 en **dst** y todos los demás píxeles que no cumplan con la condición se les asigna un valor de 0.
Para el ejemplo de la Figura 3.3, se establece un límite inferior igual a $83 - 7 = 76$ y un límite superior igual a $83 + 7 = 90$ en la función cv::inRange(), de la cual se obtiene la imagen que se muestra en la Figura 3.4.

Figura 3.4: Resultado de inRange().

Una propiedad importante de una distribución normal es que se tiene el 99.7 % del área bajo la curva dentro de tres desviaciones estándar del promedio, como se muestra en la Figura 3.5.

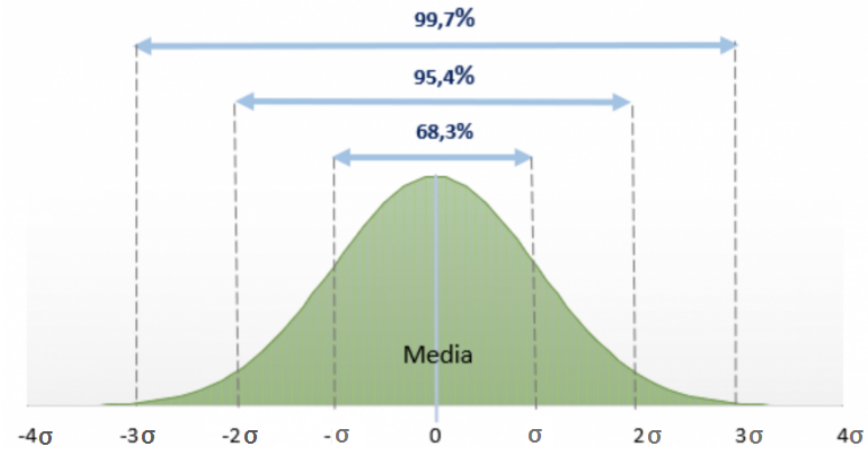
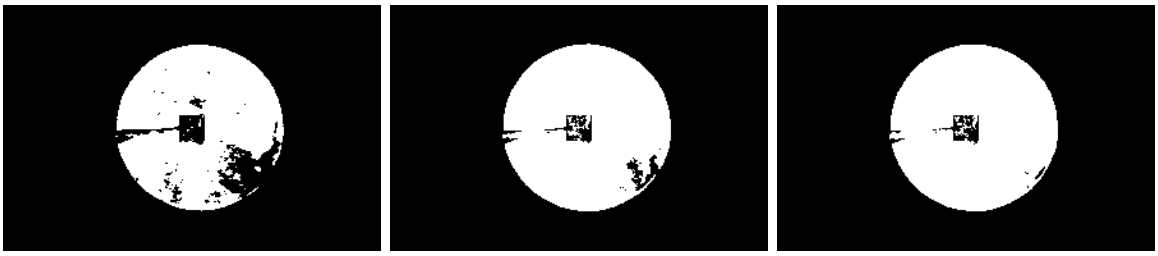


Figura 3.5: [Inbestme19] Gráfica del área bajo la curva de una distribución normal.

Entonces, se multiplica σ por una constante K que se ajusta manualmente para mejorar el resultado, como muestra en la Figura 3.6. Se observa que se tiene mejor resultado cuando $K = 4$, ya que se detecta perfectamente el área negra del dojo.



(a) K=2 (b) K=3 (c) K=4

Figura 3.6: Resultados para K = 2, 3 y 4.

Antes de determinar el modelo de la zona blanca del dojo, se necesita obtener un modelo matemático que represente la circunferencia del área blanca de la Figura 3.6(c). Las características que describen esta circunferencia se calculan a partir de tres puntos que se encuentren sobre ella. A continuación se describe el proceso que se lleva a cabo para la detección de estos puntos.

3.2. Detección de los puntos sobre la circunferencia

Para la detección de puntos sobre la circunferencia, se toman rectas de píxeles de la imagen, como se muestra en la Figura 3.7 en color azul.

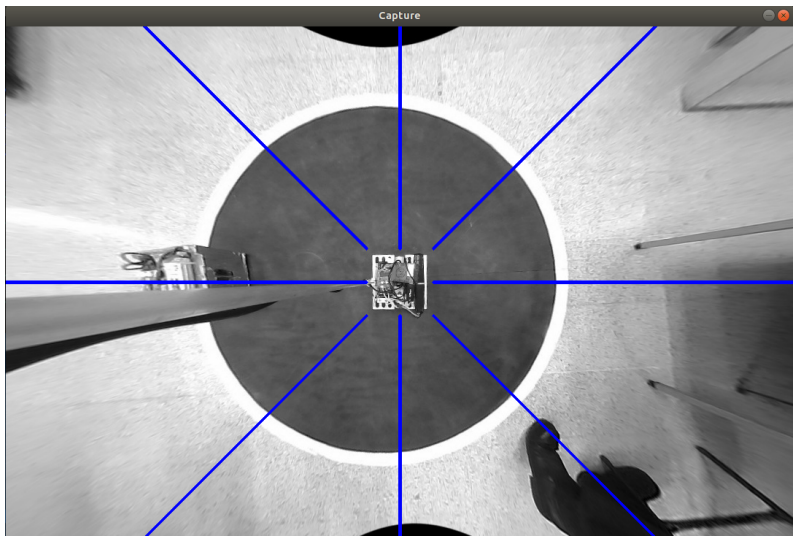


Figura 3.7: Rectas de píxeles tomados de una captura de la cámara.

Para generar estas rectas se utiliza el algoritmo de Bresenham [Hearn06]. Este es un algoritmo preciso y eficiente para la generación de líneas digitalizadas, al cual se le da como parámetros dos puntos (uno de inicio y otro de finalización). Para nuestro caso se utiliza este algoritmo para determinar que píxeles se encuentran en estas rectas. Se toman rectas de píxeles en forma de rayos al rededor del robot, las cuales tienen un ángulo de 45° entre ellas.

La función donde se implementa el algoritmo de Bresenham retorna un vector de puntos que pertenecen a la línea que conecta los dos puntos dados como parámetros. Entonces, se utiliza esta función para generar las rectas deseadas.
Se recorren estas rectas píxel por píxel hasta que se encuentren píxeles que no correspondan al modelo del color, recorriendo desde el centro del dojo hacia afuera. Después, se toman los puntos del los píxeles donde se encuentre el cambio de color, como se observa en la Figura 3.8.
Teniendo ya los puntos sobre la circunferencia de la zona negra, se procede a calcular el

centro y el radio que la describen matemáticamente.

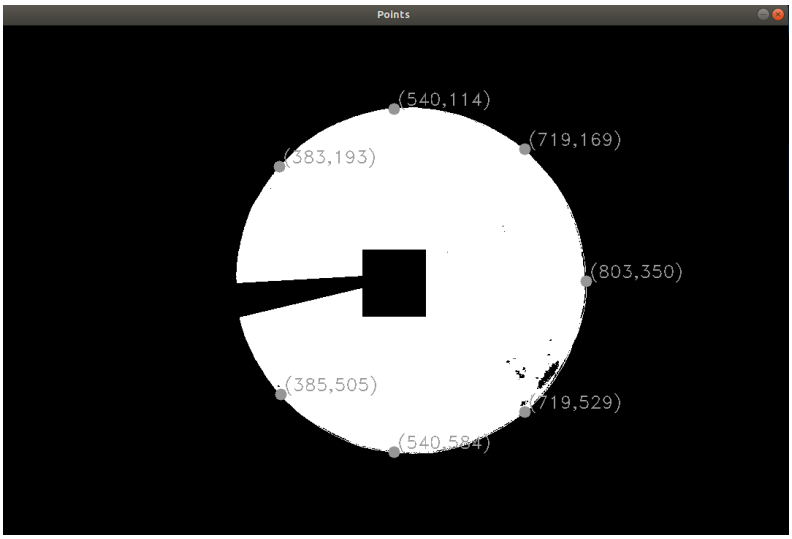


Figura 3.8: Coordenadas de los puntos en la circunferencia.

3.3. Modelo matemático de una circunferencia

Una *circunferencia* se define como [Ditutor18] una línea curva cerrada cuyos puntos están a la misma distancia de un punto fijo llamado centro. Entonces, el *centro* es el punto del que equidistan los puntos de la circunferencia y el *radio* es el segmento que une el centro de la circunferencia con un punto cualquiera de la misma.
La ecuación general de una circunferencia en un plano se expresa como:

$$r^2 = (x-a)^2 + (y-b)^2 \tag{3.5}$$

Donde **r** es el radio, **(x,y)** es un punto de la circunferencia, y **(a,b)** es la coordenada del centro de la circunferencia. Si desarrollamos la ecuación anterior se tiene que:

$$x^2 + y^2 - 2ax - 2by + a^2 + b^2 - r^2 = 0 \tag{3.6}$$

Como a, b y r son constantes, se puede decir que:

$$A = -2a \tag{3.7}$$

$$B = -2b \tag{3.8}$$

$$C = a^2 + b^2 - r^2 \tag{3.9}$$

Sustituyendo estas constantes en la ecuación 3.6 se obtiene:

$$x^2 + y^2 + Ax + By + C = 0 \tag{3.10}$$

Donde el centro se encuentre en:

$$\left(-\frac{A}{2}, -\frac{B}{2}\right) \tag{3.11}$$

Y el radio esta dado por:

$$r = \sqrt{\left(\frac{A}{2}\right)^2 + \left(\frac{B}{2}\right)^2 - C} \tag{3.12}$$

Teniendo el modelo matemático de la circunferencia se procede a calcular el centro y el radio a partir de los puntos obtenidos en la sección anterior.

3.4. Ecuación de una circunferencia que pasa por tres puntos

Para calcular los parámetros A, B y C de la ecuación 3.10 se genera un sistema de ecuaciones.

Se reacomoda la ecuación:

$$Ax + By + C = -(x^2 + y^2) \tag{3.13}$$

Después, con tres de los puntos obtenidos de la circunferencia, se genera el siguiente sistema de ecuaciones a partir de la ecuación anterior, tomando en cuenta los puntos como $P1(x_1, y_1)$, $P2(x_2, y_2)$ y $P3(x_3, y_3)$.

$$Ax_1 + By_1 + C = -(x_1^2 + y_1^2) \tag{3.14}$$

$$Ax_2 + By_2 + C = -(x_2^2 + y_2^2) \tag{3.15}$$

$$Ax_3 + By_3 + C = -(x_3^2 + y_3^2) \tag{3.16}$$

Teniendo el sistema de ecuaciones anterior se crean las siguientes matrices.

$$U = \begin{bmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{bmatrix} \tag{3.17}$$

$$V = \begin{bmatrix} -(x_1^2 + y_1^2) \\ -(x_2^2 + y_2^2) \\ -(x_3^2 + y_3^2) \end{bmatrix} \tag{3.18}$$

$$Z = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \tag{3.19}$$

A partir de 3.17, 3.18 y 3.19 se genera el siguiente sistema matricial.

$$U * Z = V \tag{3.20}$$

Como se desea obtener Z, se multiplica ambos lados de la ecuación por la inversa de U, obteniendo lo siguiente.

$$Z = U^{-1} * V \tag{3.21}$$

En este método se obtiene el radio y el centro de la circunferencia por medio de tres puntos, pero para nuestro caso se pueden tener más de tres. Entonces, se propone resolver este problema mediante el método de mínimos cuadrados.

3.5. Método de mínimos cuadrados

Se pueden tener un máximo de ocho puntos de la circunferencia, ya que se tienen ocho rectas. Sin embargo, el sistema matricial establecido en la ecuación 3.21 se vuelve un sistema sobre dimensionado porque se tendrían ocho ecuaciones y tres incógnitas.
Entonces, para resolver este sistema se tendrá que aproximar los valores que mejor se ajusten para A, B y C mediante el método conocido como método de mínimos cuadrados.
La matriz U ahora es de tamaño 8x3, la matriz V de 8x1 y la matriz Z de 3x1, por lo que se vuelve un sistema sobre-determinado. Entonces, la solución por el método de mínimos cuadrados esta dada por:

$$Z = (U^t U)^{-1} U^t V \tag{3.22}$$

Como el método de mínimos cuadrados se ajusta a todos los puntos dados, se debe tener en cuenta que algunos puntos pueden ser datos atípicos, o sea que no pertenecen al modelo matemático. Para solucionar este problema, se hace uso del algoritmo RANSAC, el cual se explica a continuación.

3.6. Algoritmo RANSAC para la determinación de puntos válidos

El algoritmo de RANSAC (RANdom SAmple Consensus)[Flores11] es un algoritmo iterativo utilizado para estimar los parámetros de un modelo matemático de un conjunto de datos que contiene valores atípicos. Un *outlier* ó valor atípico es una observación que es numéricamente distante del resto de los datos. Por otro lado, un *inlier* es un dato que es válido y se ajusta al modelo. El método de mínimos cuadrados le da el mismo peso a todos los datos, por lo tanto la presencia de un dato atípico puede llegar a distorsionar el modelo obtenido.

El algoritmo RANSAC logra su objetivo mediante la repetición de los siguientes pasos:

1. Tomar tres puntos aleatorios del conjunto de puntos.

2. Se ajusta el modelo de la circunferencia al conjunto de posibles puntos válidos, donde se calcula el radio y la coordenada del centro.

3. Todos los demás datos se prueban contra el modelo ajustado. Esos puntos que se ajustan al modelo estimado, de acuerdo con un parámetro de error al cual no debe rebasar, se consideran como parte del conjunto de consenso. El error máximo permitido se establece de cinco píxeles y el error se calcula mediante:

$$err = |r - \sqrt{(x_p - x_c)^2 + (y_p - y_c)^2}| \tag{3.23}$$

   Donde err es el error en píxeles, r y (x_c, y_c) son el radio y centro calculado para el conjunto posibles puntos válidos y (x_p, y_p) es uno de los puntos en la circunferencia.

4. El modelo estimado es bueno si se han clasificado la mayoría de puntos como parte del conjunto de consenso tomando en cuenta que el error calculado en cada uno de los puntos sea menor al error máximo tolerado.

5. Estos pasos se repiten hasta lograr todas las combinaciones posibles en las que se pueden generar tres puntos del conjunto de puntos, tomando en cuenta que no se pueden repetir. Entonces, para calcular número máximo de combinaciones posibles mediante:

$$C_m^n = \frac{m!}{n!(m-n)!} \tag{3.24}$$

Donde C es la cantidad de combinaciones posibles de m elementos tomados de n en n. Por ejemplo, se tienen 8 rayos y se toman sólo 3 puntos aleatorios a la vez, entonces, la cantidad de combinaciones posibles son:

$$C_8^3 = \frac{8!}{3!(8-3)!} = 56$$

Sin embargo, se puede calcular el número de iteraciones necesario para llegar a un resultado aceptable, en lugar de realizar todas las combinaciones posibles.
El número de iteraciones [Derpanis10], N, se elige lo suficientemente grande para asegurar que al menos uno de los conjuntos aleatorios de las muestras no incluya algún dato atípico. Sea u la probabilidad de que cualquier punto de los datos sea válido y p la probabilidad de que un conjunto aleatorio no contenga algún dato atípico. Se requieren N iteraciones para m cantidad de puntos, donde:

$$1 - p = (1 - u^m)^N \tag{3.25}$$

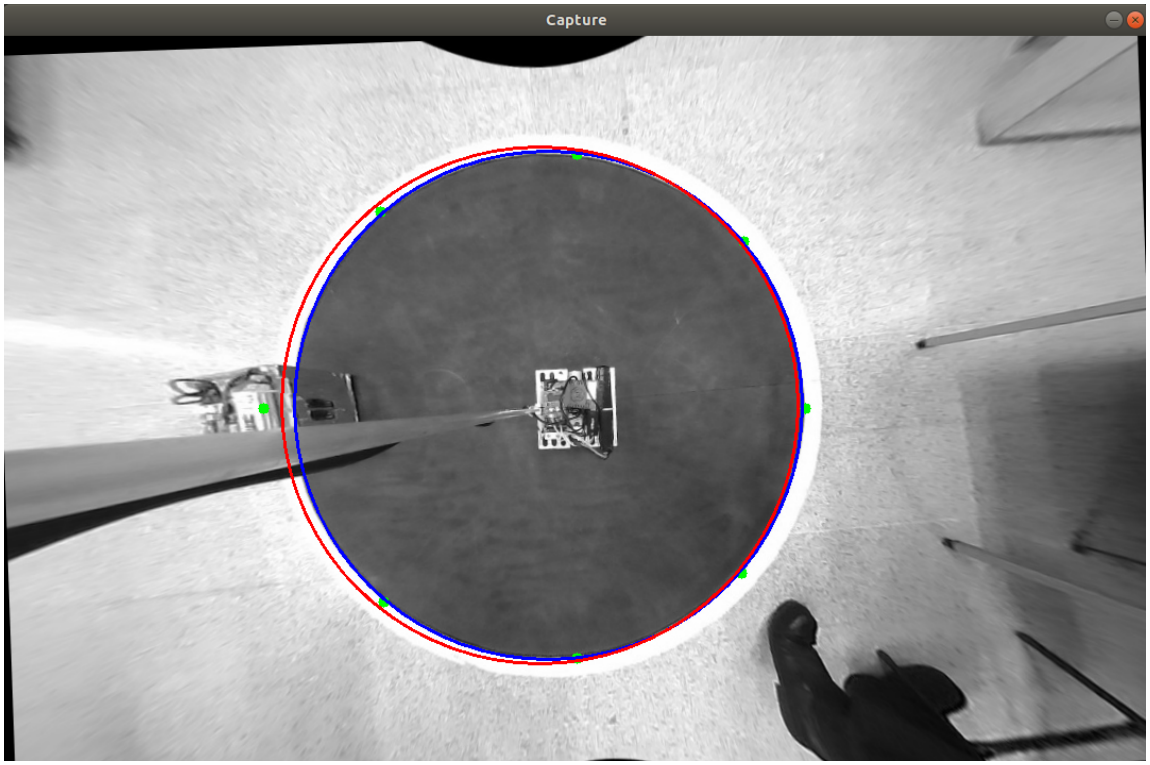
Despejando N:

$$N = \frac{log(1-p)}{log(1-u^m)} \tag{3.26}$$

Para nuestro caso, se desea tener al menos un conjunto de puntos aleatorios que no tenga algún dato atípico, entonces, $p = 0.99$. Se tienen 8 rayos y los conjuntos son de 3 puntos y se tiene un punto para cada rayo, por lo tanto, $u = 1 - \frac{3}{8} = 0.625$ y $m = 3$. Al realizar el cálculo de las N iteraciones da como resultado:

$$N = \frac{log(1-0.99)}{log(1-0.625^3)} = 16.4529$$

Se observa que son mucho menos iteraciones a si se realizan todas las combinaciones posibles. En la Figura 3.9 se muestran algunos de los resultados que se obtienen al utilizar el algoritmo RANSAC.



(a) Con un dato atípico.

(b) Con dos datos atípicos.

Figura 3.9: Resultados de utilizar el algoritmo RANSAC

El círculo azul es el resultado de aplicar el método de mínimos cuadrados utilizando el algoritmo RANSAC y el rojo el resultado de aplicar el método de mínimos cuadrados con todos los puntos.

3.7. Modelo de color del área blanca

Para realizar el cálculo de la media y la desviación estándar del borde blanco del dojo, se utilizan el centro y radio calculado anteriormente para la circunferencia de la zona negra. El área de muestra se toma como un anillo que rodea la parte negra del dojo con 10 píxeles de gruesor, como se observa de color rojo en la Figura 3.10.

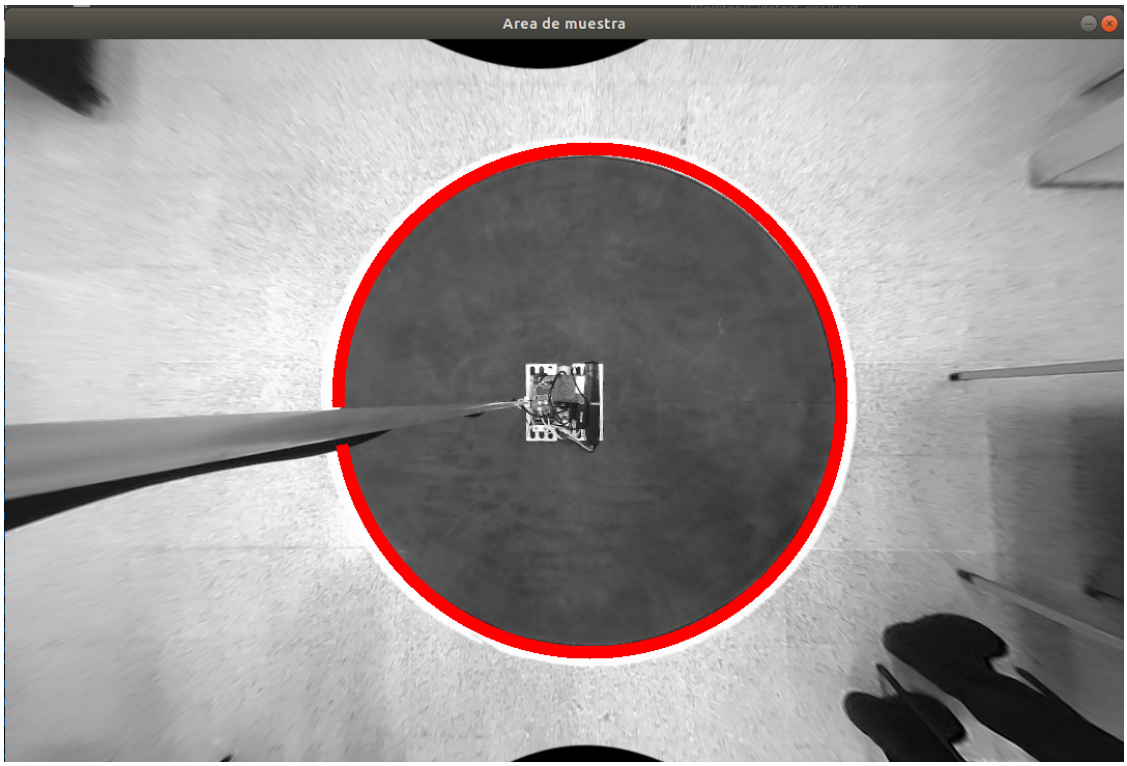


Figura 3.10: Área de muestra del color blanco.

Se calcula la media y la desviación estándar de ésta área. Por ejemplo, para la Figura 3.10 da los siguientes resultados.

$$\mu = 251 \qquad \text{y} \qquad \sigma = 14 \tag{3.27}$$

Aplicando la función cv::inRange() con un rango el rango [237,255] nos da como resultado la imagen de la Figura 3.11.

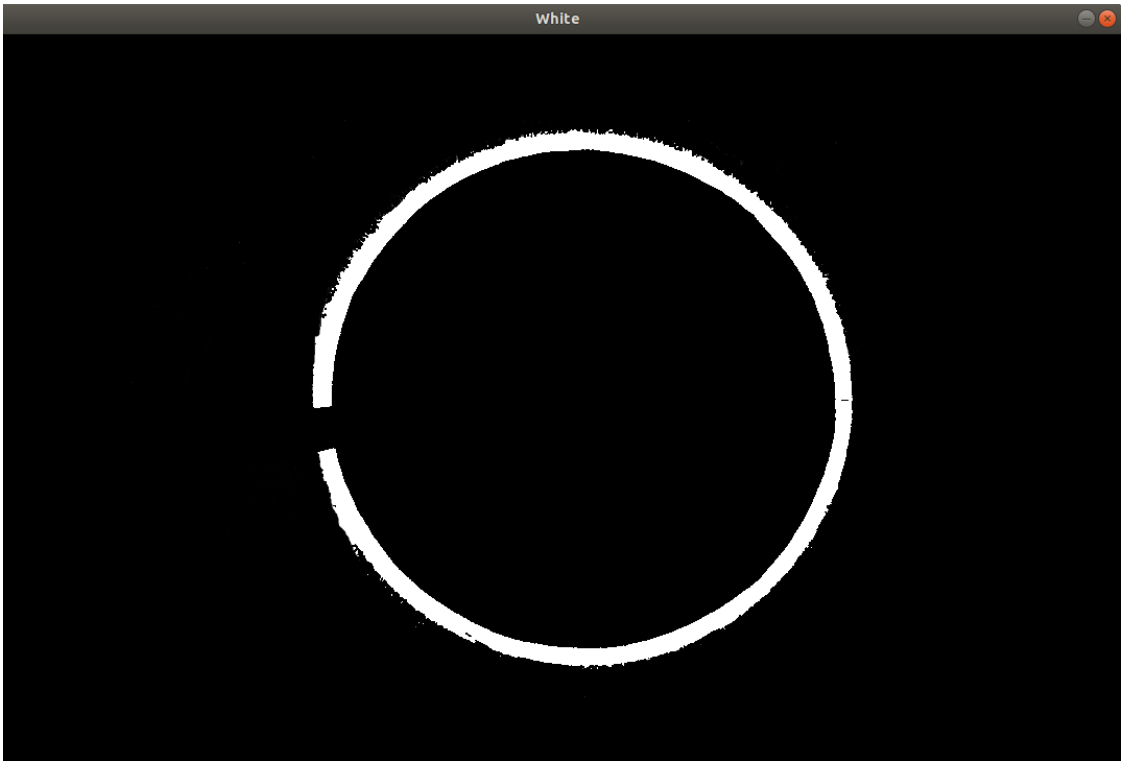


Figura 3.11: Zona blanca del dojo.

Los parámetros calculados de la media y la desviación estándar para ambas zonas se guardan en un archivo y se leen al inicio de cada combate. Antes de iniciar el combate, se deja presionado el botón de encendido unos segundos para que el robot entre en modo de calibración para el cálculo de estas variables. Y durante el combate, se utiliza el modelo que representa la zona blanca del dojo para continuamente calcular el centro y el radio de la circunferencia de la zona negra.

Para obtener resultados más precisos, se ejecuta el algoritmo de RANSAC dos veces. La primera vez se ejecuta el algoritmo con todos los radios y centros estimados. La segunda vez se ejecuta utilizando el cálculo del radio y centro anterior.
En la Figura 3.12 se observan los cuatro principales que se siguen para la determinación de las características que describen al dojo, las cuales se describen a continuación:

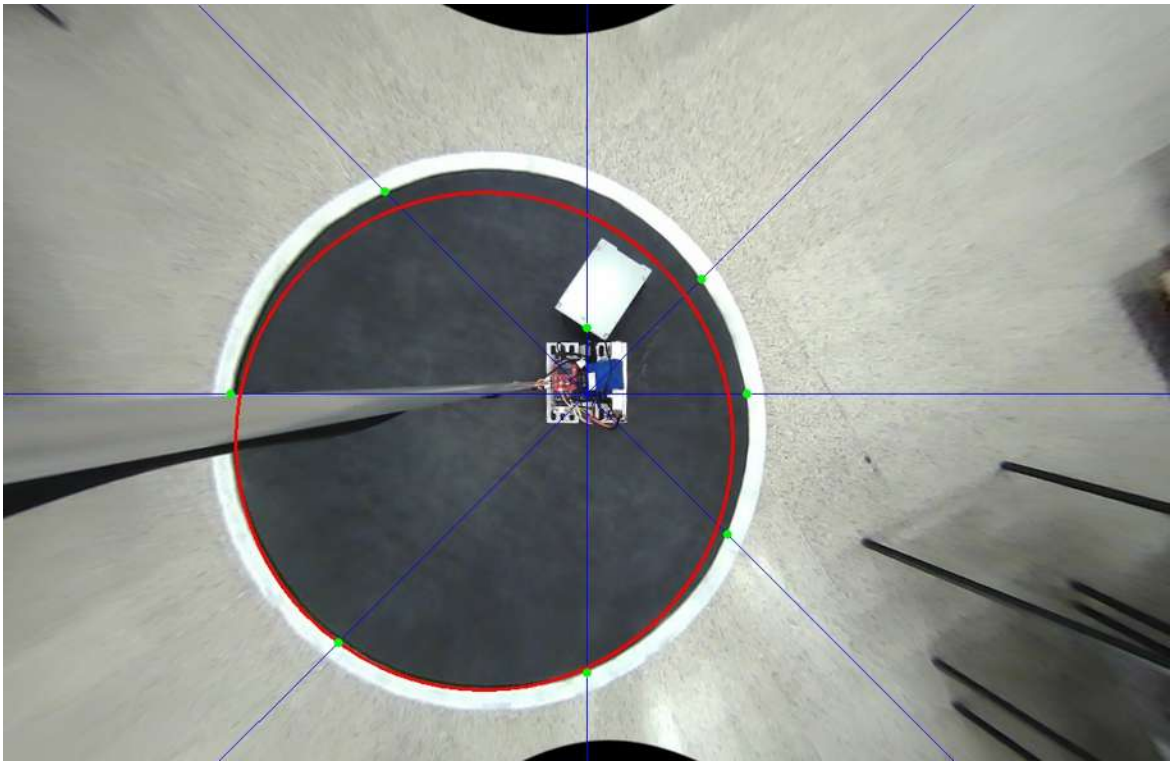
1. **Paso 1:** Se toman todos los puntos encontrados por medio de los rayos.

2. **Paso 2:** Con los puntos anteriores, se ejecuta el algoritmo RANSAC para obtener el primer conjunto de puntos sin datos atípicos. Con estos datos filtrados se calcula el centro y el radio de la primera circunferencia que representa al dojo.

3. **Paso 3:** Se toman de nuevo los puntos encontrados por medio de los rayos, pero esta vez los rayos empiezan desde el centro del dojo calculado en el Paso 2.

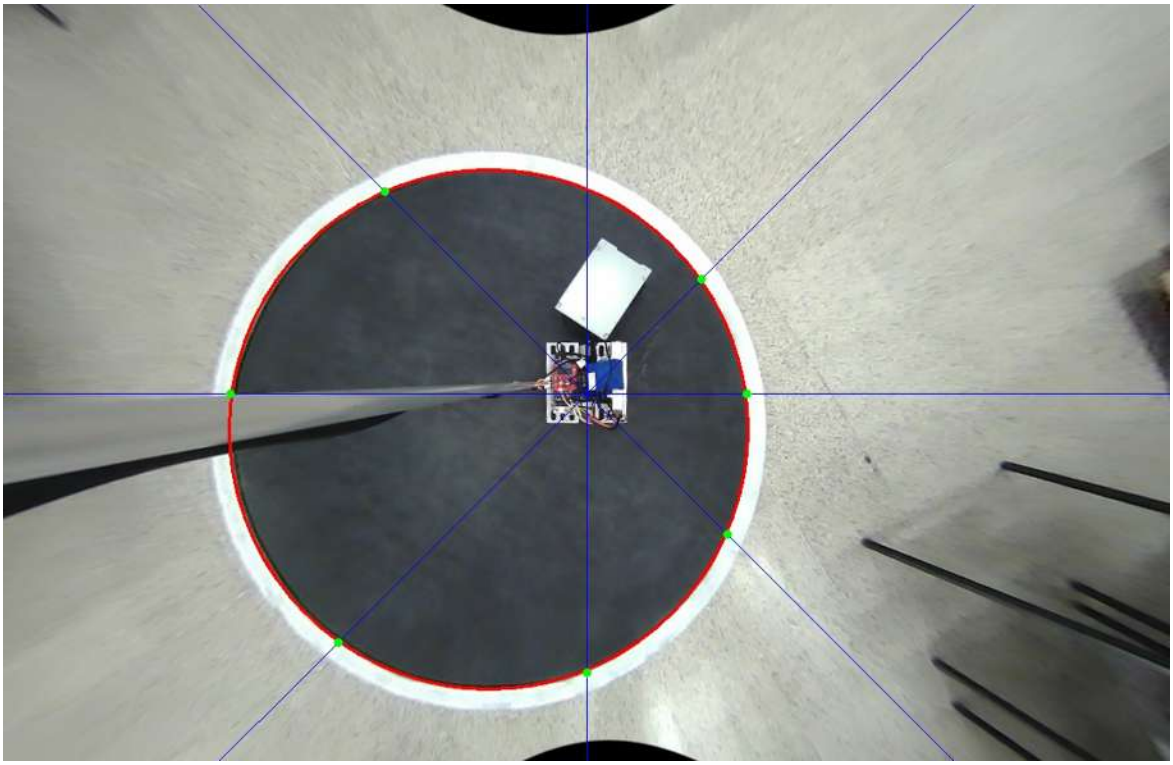
4. **Paso 4:** Con los puntos anteriores, se ejecuta de nuevo el algoritmo RANSAC pero ahora se utiliza el centro y el radio calculado en el Paso 2. Con estos datos libres de datos atípicos se realiza el calculo final del centro y radio de la circunferencia que describe al dojo.

Mediante imágenes en escala de grises y haciendo uso de un modelo de distribución normal, se logró obtener un conjunto de puntos sobre la circunferencia del área negra del dojo. Con estos puntos, se calculó el centro y el diámetro de la circunferencia del dojo, estableciendo

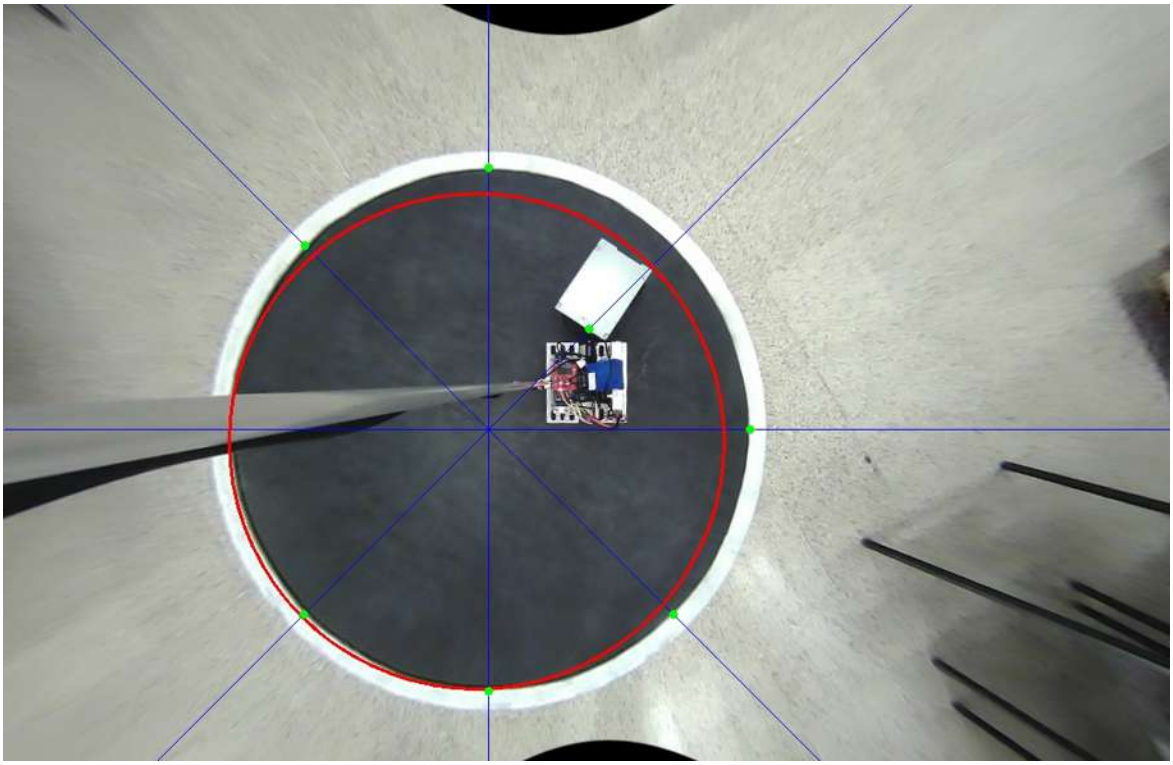


(a) Paso 1: Utiliza todos los puntos.

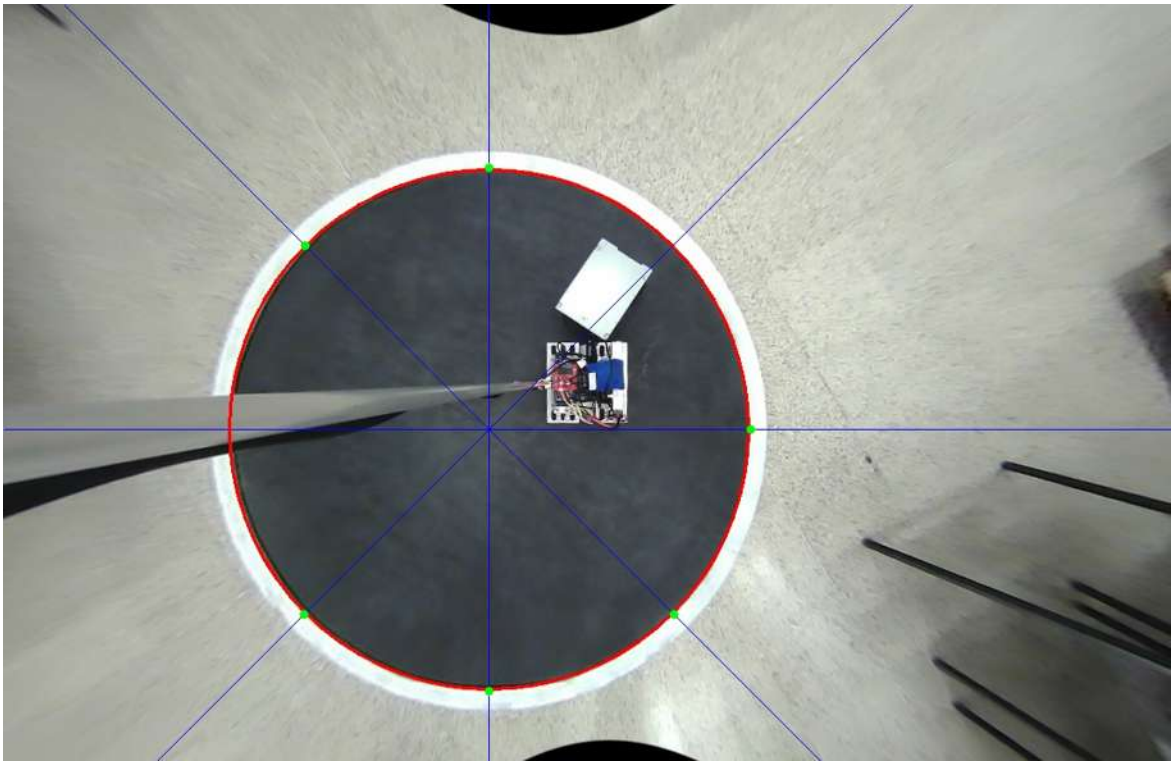
con estos parámetros el límite del área de combate. Los parámetros que representan al color blanco y al negro, calculados a partir del modelo de distribución normal, serán el punto de partida para la detección del robot oponente, lo cual se describe en el siguiente capítulo.



(b) Paso 2: Utiliza RANSAC.



(c) Paso 3: Utiliza todos los puntos con el nuevo centro.



(d) Utiliza RANSAC con el nuevo centro.

Figura 3.12: Pasos para la detección del dojo.

Capítulo 4

Detección del oponente

En este capítulo se analiza el proceso de la detección del oponente por medio del modelo de color obtenido anteriormente. Con el modelo de color de la zona blanca se establece el área en la que se busca al oponente. Con el modelo de color de la zona negra se genera una imagen binaria, con la cual se obtienen los contornos de los píxeles que no pertenecen a este modelo y se encuentren dentro del dojo. Con estos contornos se calcula la posición del oponente.

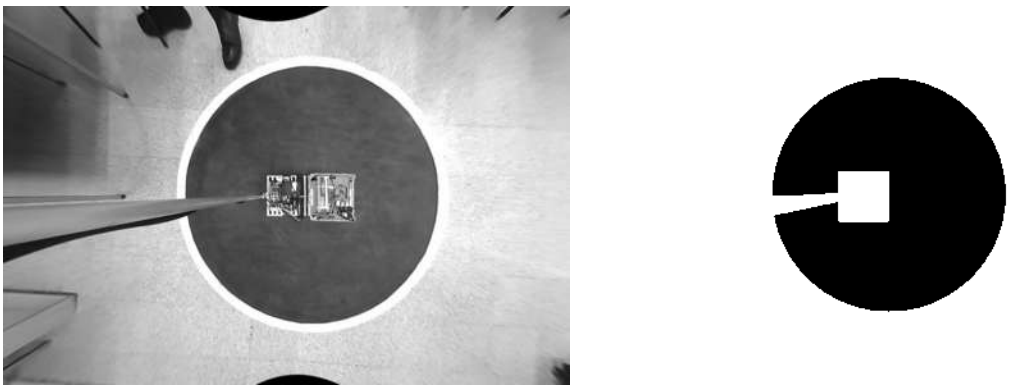
4.1. Región de búsqueda del oponente

Para determinar la región en donde se buscará al oponente, se utiliza el modelo del color blanco para detectar la circunferencia alrededor del dojo. En una imagen binaria de color blanco se dibuja un círculo de color negro con centro y radio igual al obtenido mediante lo establecido en el capítulo anterior. Este círculo se dibuja utilizando la función de OpenCV llamada cv::circle(). La función cv::circle() [OpenCV19a] dibuja un círculo simple o lleno con un centro y radio dado. La función cv::circle() recibe los siguientes parámetros.

```
cv::circle(Mat& img, Point center, int radius, const Scalar& color, int thickness);
```

De la función anterior, el parámetro **img** es la imagen en donde se dibujará el círculo, **center** es donde se localiza el centro del círculo, **radius** es el radio del círculo, **color** es el color del círculo, **thickness** es el gruesor de la circunferencia, si es positivo. Si es negativo, significa que se dibujará un círculo lleno.

En la región que crea este círculo no se toma en cuenta el área que abarca el robot y su mástil, de tal forma que se tendrá una imagen como la que se muestra en la siguiente figura.



(a) Original (b) Imagen binaria

Figura 4.1: Región de búsqueda del oponente.

Entonces, la región de color negro de la imagen binaria es el área donde se buscará al oponente. En la siguiente sección se analiza el proceso de detección del oponente dentro de la región de búsqueda establecida.

4.2. Detección del robot oponente

Para la detección del oponente se utiliza el modelo de color negro. Usando la función cv::inRange() con los parámetros del modelo del color negro se obtiene la imagen que se muestra a continuación.

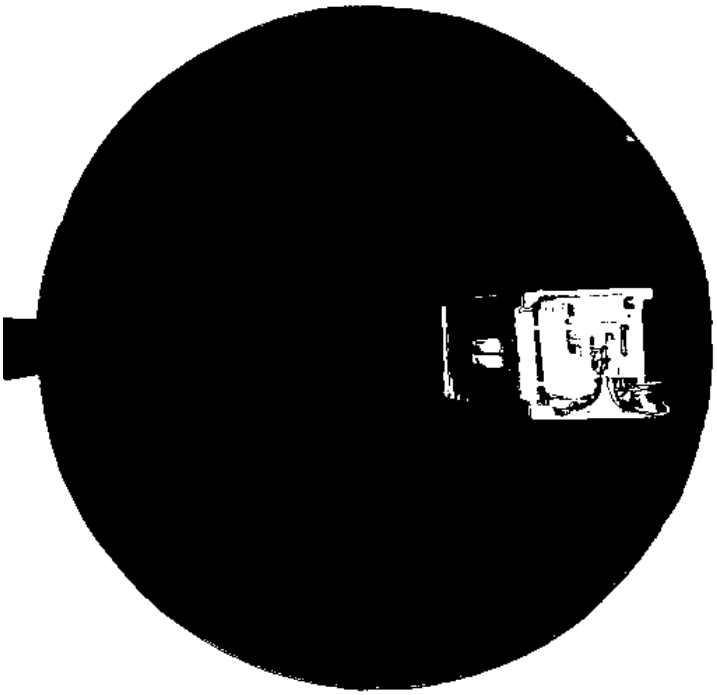


Figura 4.2: Imagen binaria del modelo de color negro.

Se puede observar que existen algunos píxeles de color blanco que no pertenecen al robot oponente. Para eliminar estas pequeñas agrupaciones de píxeles blancos se hace uso de las funciones cv::erode() y cv::dilate().

La función cv::erode() [OpenCV19b] erosiona una imagen usando un elemento de estructuración específico, por lo elimina pequeños grupos de píxeles de color blanco. En cambio la función cv::dilate() dilata una imagen usando también un elemento de estructuración específico, por lo que regresara la imagen original sin los grupos de píxeles blancos eliminados anteriormente.

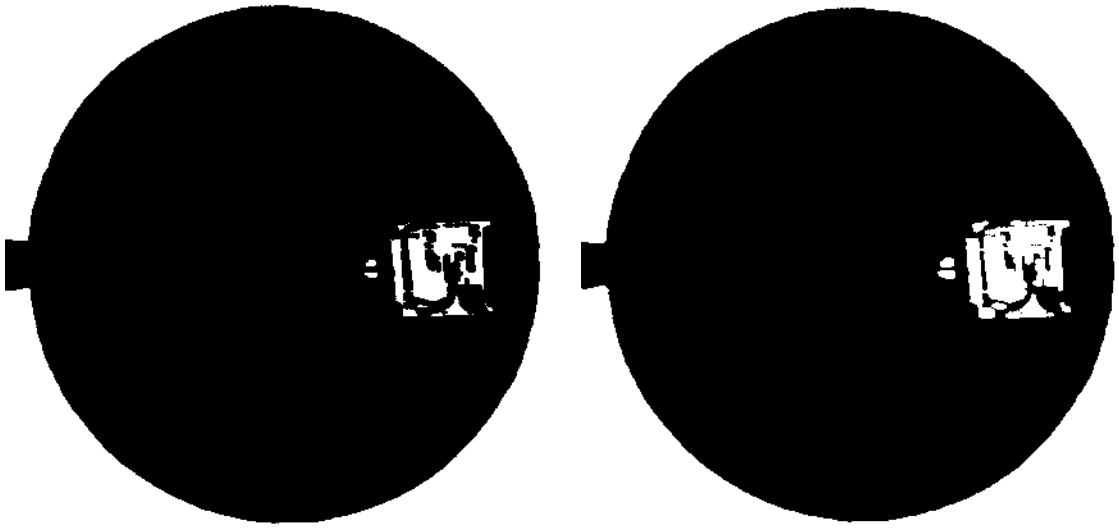
El siguiente fragmento de código obtiene el elemento de estructuración para realizar la erosión y la dilatación de la imagen.

```
cv::Mat element = cv::getStructuringElement(MORPH_ELLIPSE,cv::Size(3,3),
                  cv::Point(1,1));
```

Para las funciones cv::erode() y cv::dilate() se requiere dar como parámetros los siguiente.

```
cv::erode(InputArray src, OutputArray dst, Mat element);
cv::dilate(InputArray src, OutputArray dst, Mat element);
```

Donde **src** es la imagen fuente, **dst** es la imagen del resultado de la operación y **element** es el elemento de estructuración utilizado para las funciones.
En la Figura 4.3 se muestra el resultado de la erosión y posteriormente la dilatación de la imagen de la Figura 4.2.



(a) Erosión (b) Dilatación posterior a la erosión

Figura 4.3: Resultados de las funciones cv::erode() y cv::dilate().

A partir de este resultado se procede a obtener los puntos sobre los contornos de los grupos de píxeles blancos. Esto se puede realizar fácilmente mediante la función cv::findContours() de OpenCV. Esta función recibe los siguientes parámetros:

```
cv::findContours( &Mat src,vector<vector<Point>> contours,CV_CHAIN_APPROX_NONE)
```

En los cuales **src** es la imagen fuente, **contours** es un vector de vectores de puntos que pertenece a cada contorno y CV CHAIN APPROX NONE es una bandera para tomar todos los píxeles adyacentes como un solo contorno. Para apreciar visualmente estos contornos, se pueden dibujar por medio de la siguiente función.

```
cv::drawContours(&Mat src,vector<vector<Point>> contours,int contourIdx,
 const Scalar color, int thickness)
```

Recibe los mismos parámetros que la función cv::findContours(), con la adición de **contourIdx** que es el índice del contorno a dibujar si es positivo, si es negativo dibuja todos los

contornos.

Al dibujar los contornos de la Figura 4.3(b) se obtiene imagen de la Figura 4.4.

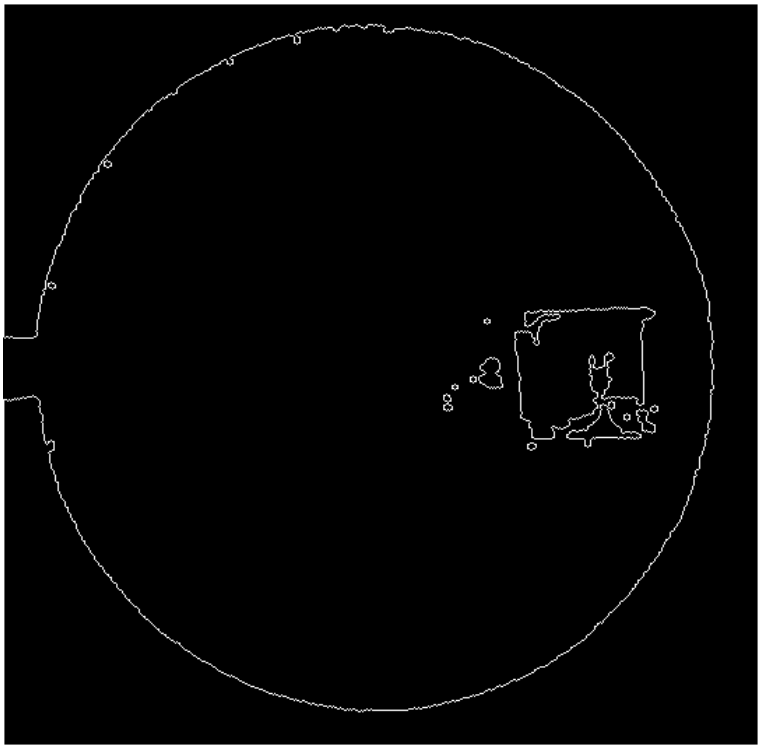


Figura 4.4: Todos los contornos.

Se desechan los contornos que tengan menos de diez y más de mil píxeles para quitar todos los contornos no deseados, lo cual da como resultado la imagen que se muestra en la Figura 4.5.

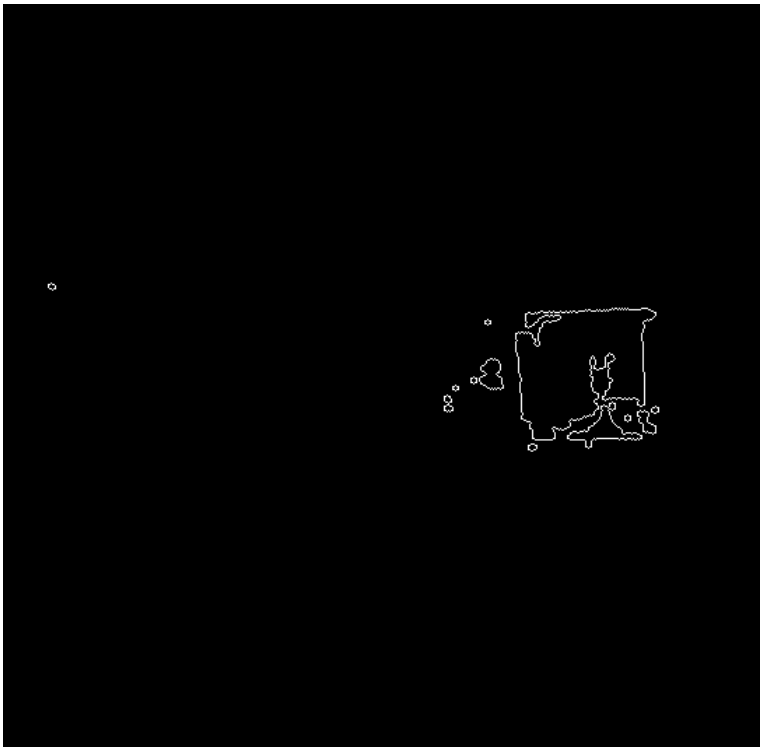


Figura 4.5: Contornos dentro del rango.

Existe una función de OpenCV llamada cv::minEnclosingCircle() [OpenCV19d] la cual encuentra el círculo del área mínima que encierra un conjunto de puntos usando un algoritmo iterativo. Esta función recibe los siguientes parámetros.

```
cv::minEnclosingCircle(InputArray points,Point2f& center,float& radius)
```

Donde **points** es el vector de los puntos de los contornos, en **center** se guarda el centro del círculo y en **radius** el radio del círculo. Al dibujar estos círculos nos da la siguiente imagen.

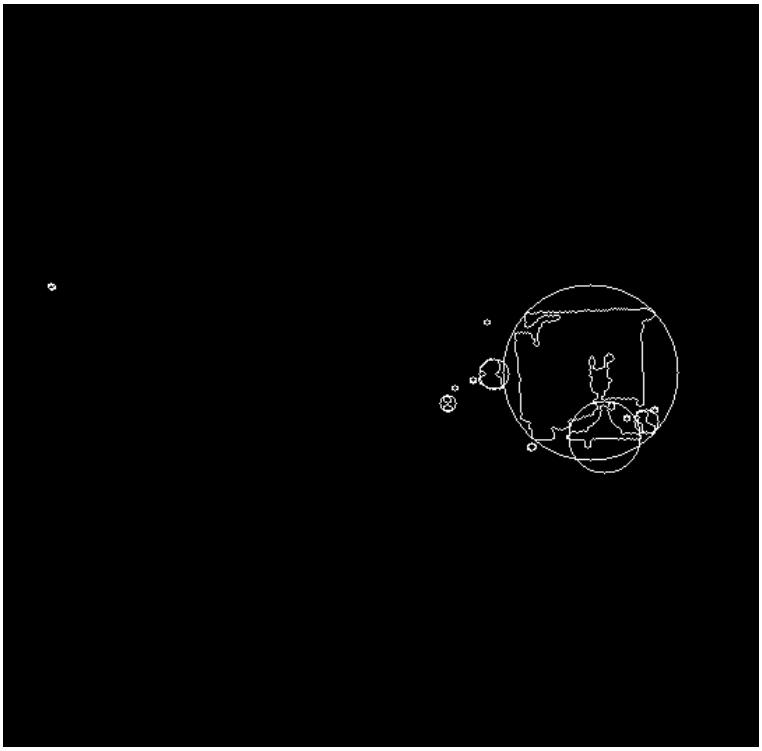


Figura 4.6: Círculos de cierre mínimo.

Se observa que los contornos forman parte del oponente, sin embargo no forman un solo contorno. Por lo tanto, se unen los centros de los contornos obtenidos al centro del contorno más grande. De esta forma, se tendrá un solo contorno el cual represente a todo el robot oponente. En la siguiente imágenes se muestra la unión de estos contornos y el círculo que lo encierra.

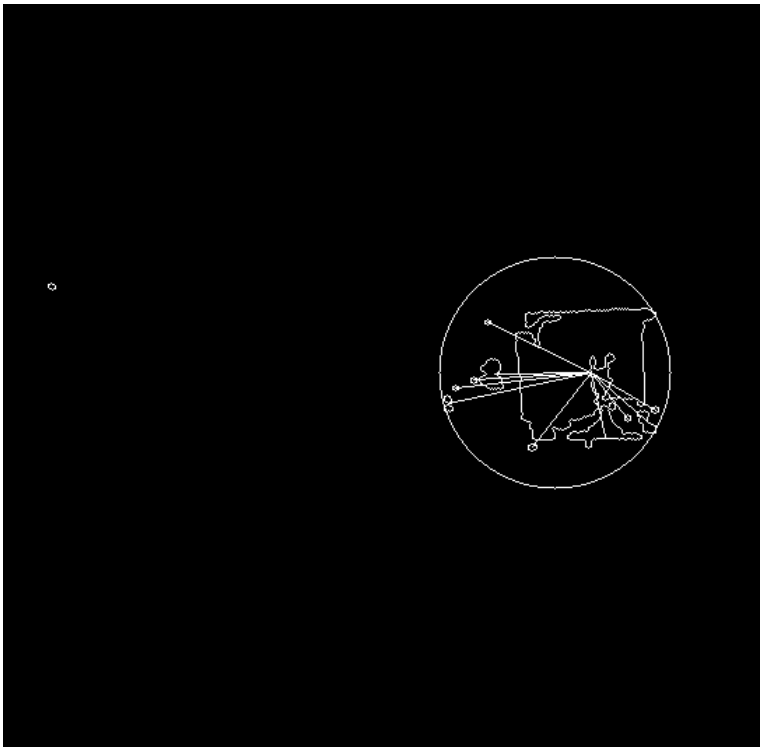


Figura 4.7: Círculo de cierre mínimo de todo el robot oponente.

Con la función cv::minEnclosingCircle() se encuentra el centro del robot oponente y el radio del círculo que lo rodea. Sin embargo, se desea obtener el punto más cercano del oponente al robot.

Para solucionar este problema, se propone que en una imagen binaria de color negro se dibuje el círculo que encierra al contorno del oponente. Con la función de Bresenham del Capítulo 3 se obtiene un vector de puntos que conectan el centro de este círculo y el centro del robot. Después, se recorren estos píxeles hasta encontrar un color diferente. Donde se encuentre este cambio se toma como el punto más cercano del oponente al robot. Esto se puede apreciar en la siguiente imagen, donde el círculo rojo es el límite de la región de búsqueda, el círculo amarillo es el que encierra al contorno del oponente, el punto rojo es el centro del oponente, la línea azul es la que une los centros de los robots y el punto morado es el punto más cercano del oponente hacia el robot.

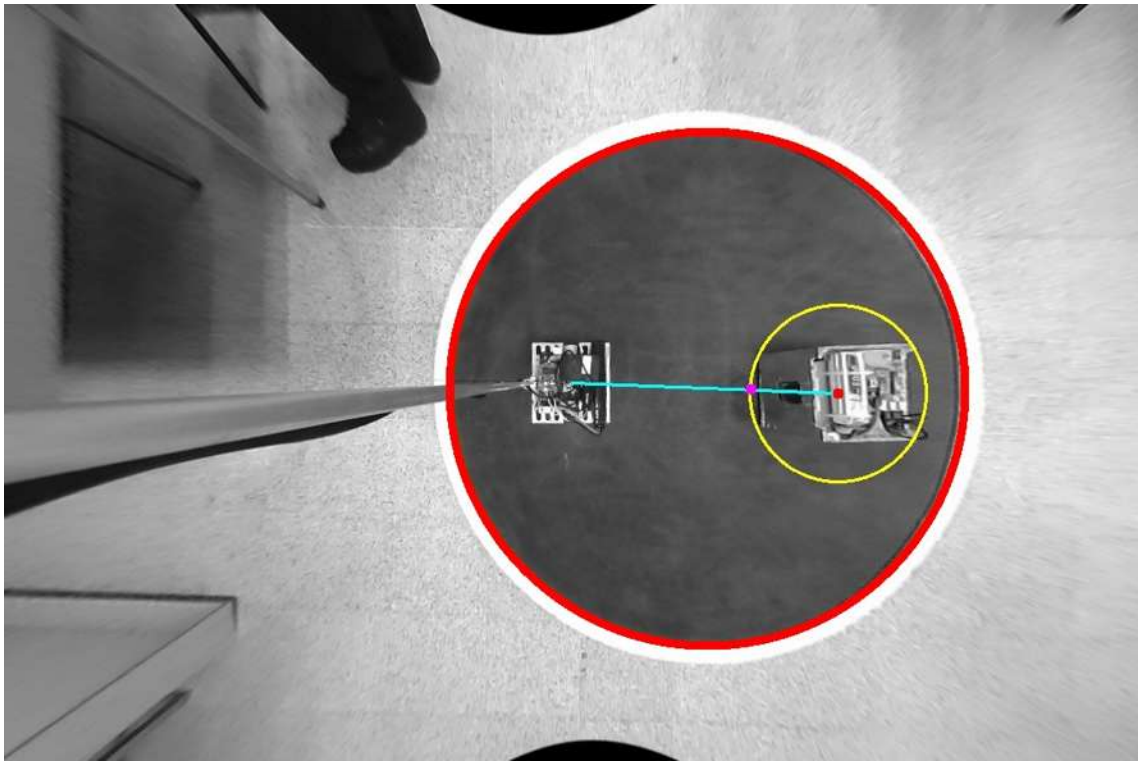


Figura 4.8: Punto más cercano del oponente al robot.

Como ya se tiene el punto de la ubicación del robot oponente, falta determinar el ángulo del giro que debe realizar el robot para poderse dirigir hacia el oponente. Esto se analiza a continuación.

4.3. Dirección del robot

En las imágenes capturadas por la cámara del robot, el frente del robot se encuentra hacia la derecha. Por lo tanto, se toma un plano cartesiano sobre la imagen, donde el origen se encuentra en el centro del robot. Se calcula la pendiente de la recta que une al origen y el punto más cercano hacia el oponente. Con esto, se calcula el ángulo agudo que forma la intersección de esta recta y el eje horizontal (eje $\mathbf{x}$). Esto se puede observar en la imagen de la Figura 4.9, donde este ángulo esta representado por α.

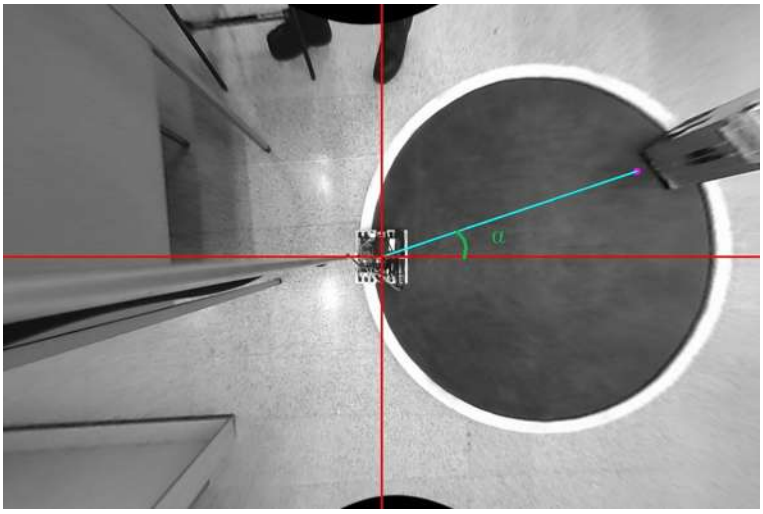


Figura 4.9: Ángulo agudo formado por el eje **x** y la recta que une el origen al robot oponente.

La tangente del ángulo α es igual la pendiente, ya que es el cociente del cateto opuesto (sobre el eje **y**) y el cateto contiguo (sobre el eje **x**) del triángulo rectángulo formado por el eje horizontal y la recta que une al origen con el oponente. Entonces, este ángulo se calcula por medio de la arco tangente de la pendiente, como se muestra en la siguiente ecuación.

$$\alpha = \arctan\left(\frac{y_r - y_0}{x_r - x_0}\right) \tag{4.1}$$

Donde (x_0, y_0) es la coordenada del robot que se encuentra en el origen y (x_r, y_r) es la coordenada del robot oponente.
En el caso de la Figura 4.9, el oponente se encuentra en el primer cuadrante, por lo que se tendrá un ángulo positivo. Sin embargo, sí el oponente se encuentra en segundo cuadrante, este ángulo será negativo. Entonces, se le suman 180° para que se tenga el ángulo formado desde la parte positiva del eje **x**. En el caso contrario, cuando el oponente se encuentre en el tercer cuadrante, se le restarán 180° ya que el ángulo α es negativo.
Entonces, cuando el oponente se encuentre en el primer y segundo cuadrante, el ángulo será positivo, en caso contrario, el ángulo será negativo. Esto se puede observar la imagen de la Figura 4.10.

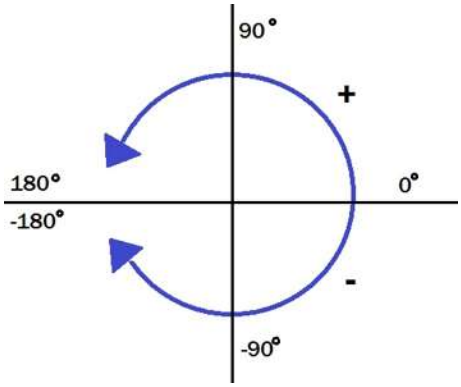


Figura 4.10: Grados de giro del robot.

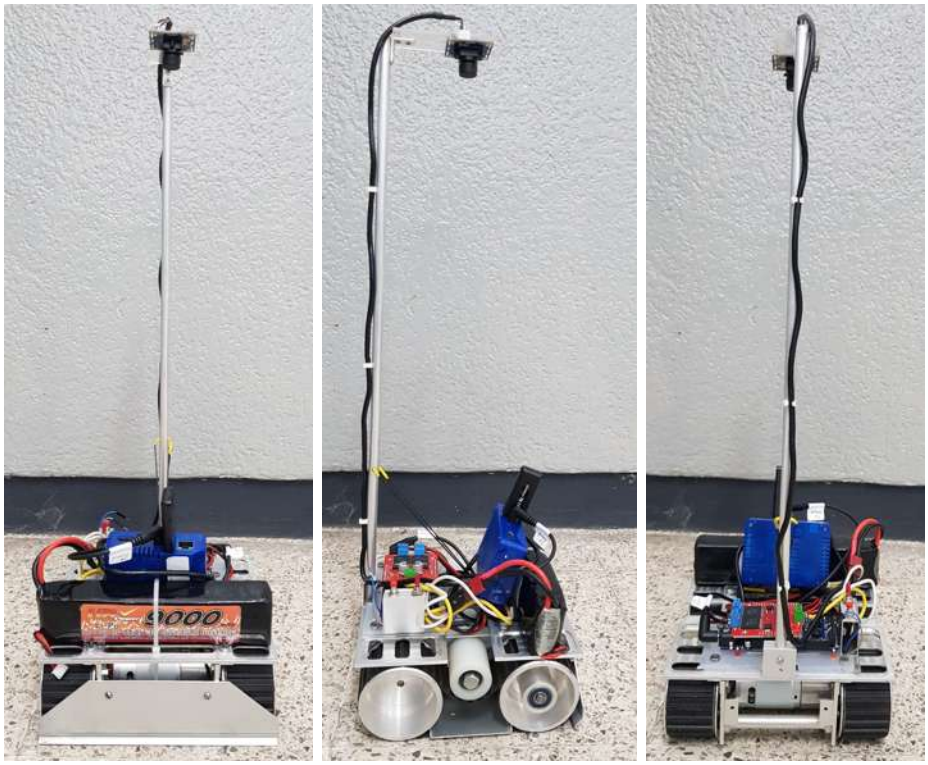
Sí el robot oponente no fuese detectado, este ángulo se toma como un número mayor a 180°.

Mediante los modelos de distribución normal que representan a el color negro y el blanco de las zonas del dojo, se logró detectar el robot oponente. Con el uso de funciones de la librería OpenCV se pudo generar en una imagen los contornos que representan al robot del adversario y así establecer la locación de éste. El ángulo formado por el eje horizontal que pasa por el centro del robot y la recta que conecta al centro del robot con el punto más cercano hacia el oponente, será una variable que servirá de base para el control de los motores, lo cual se describe en el siguiente capítulo.

Capítulo 5

Robot de sumo

En este capítulo se describe el robot móvil autónomo desarrollado para la competencia de lucha sumo, equipado con una cámara y capacidad de procesamiento a bordo del robot, incluyendo los componentes físicos del robot y también la parte de software.



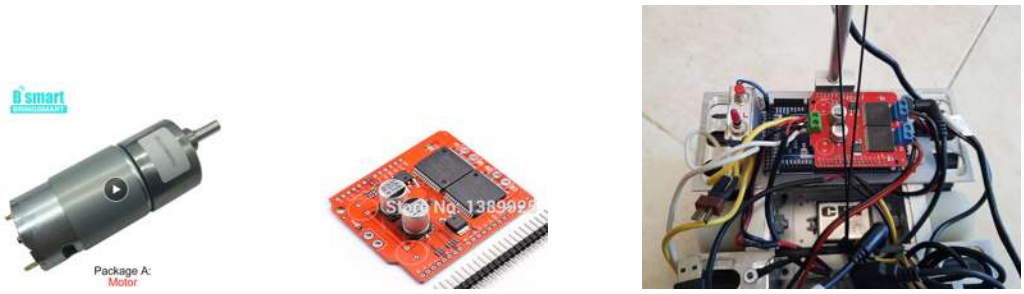
(a) Frente (b) Lado (c) Atrás

Figura 5.1: El robot desarrollado.

5.1. Componentes físicos del robot

En la figura 5.1 se muestra el robot desarrollado. El robot dispone de una cámara con un lente de muy amplia apertura (cercana a los 180 grados) y se ubica al final de un mástil que sale de la base del robot, viendo hacia abajo. Con esta cámara el robot puede ver prácticamente todo el área del tablero o ring. Se utiliza la cámara USB modelo OV2710 de la marca ELP, con una lente ojo de pez (apertura de 180 grados). La cámara tiene un sensor de 2 Megapixeles y es capaz de adquirir 30 imágenes por segundo con resolución de 1920x1080 en color.

El robot tiene forma cuadrada de 20cm x 20cm y es de tipo oruga con dos bandas de tracción accionadas por dos motores independientes. Los motores son de la marca Bringsmart modelo JGB37-550, alto par, 12V, 110rpm sin carga, a 88rpm desarrolla un par de 7kg-cm y en detenido un par máximo de 35kg-cm consumiendo 2.5A. El controlador dual utilizado para controlar los motores es una tarjeta Monster Moto Shield VNH2SP30 que se acopla directamente sobre un microcontrolador Arduino Due. La tarjeta VNHSP30 puede operar dos motores de CD, con un voltaje máximo de 16V y corriente de 14A (30A pico). En la figura 5.2 se muestra una imagen del motor utilizado, la tarjeta VNH2SP30 y su colocación sobre un Arduino Due en el robot. El microcontrolador Arduino Due es un microcontrolador avanzado de 32 bits que se utiliza para controlar los motores del robot.



(a) motor (b) Controlador (c) Controlador sobre arduino Due

Figura 5.2: El motor utilizado y el controlador dual acoplado a un arduino Due.

El robot incluye una pequeña computadora a bordo de una sola tarjeta Odroid XU4 con 8 procesadores, 2GB de memoria RAM y 32GB de memoria eMMC. Esta computadora utiliza el sistema operativo Linux Ubuntu Mate 18.04 y se le instaló la biblioteca OpenCV.

5.2. Programa del microcontrolador

Enseguida se presentan algunos fragmentos de código que se descarga al microcontrolador Arduino Due.

5.2.1. Conexiones de la tarjeta controladora de los motores al Arduino

El siguiente fragmento ayuda para conocer los pines importantes de la tarjeta controladora de los motores que se conecta al Arduino.

```
#define IN1_LEFT_MOTOR 4 //MOTOR 1 (motor izquierdo visto desde atrás del robot)
#define IN2_LEFT_MOTOR 9

#define IN1_RIGHT_MOTOR 7 //MOTOR 2 (motor derecho visto desde atrás del robot)
#define IN2_RIGHT_MOTOR 8

#define PWM_LEFT_MOTOR 6
#define PWM_RIGHT_MOTOR 5

#define CURRENT_SEN_1 A3
#define CURRENT_SEN_2 A2

#define EN_LEFT_MOTOR A1
#define EN_RIGHT_MOTOR A0

#define BUTTON_PIN 53      // Botón inicio. Está en LOW, al presionar pasa a HIGH
#define LED_PIN    13      // pin del LED integrado en el Arduino Due
```

Los pines IN1 e IN2 controlan la dirección del motor (IN1=LOW, IN2=HIGH, gira en una dirección, IN1=HIGH, IN2=LOW, gira en la otra dirección), los pines PWM se utilizan para las señales de control de velocidad mediante el esquema de Modulación por Ancho de Pulso (PWM, en inglés), los pines EN habilitan el funcionamiento del controlador (EN=HIGH indica controlador habilitado). El botón de inicio de la lucha sumo se conecta al BUTTON_PIN y finalmente LED_PIN identifica el led integrado en el Arduino Due.

Todos los pines se configuran como salidas, con excepción de los pines BUTTON_PIN y las entradas analógicas CURRENT_SEN que permiten medir la corriente que circula por los motores, como se muestra en el siguiente código:

```
void setup()
{
  pinMode(IN1_LEFT_MOTOR, OUTPUT);
  pinMode(IN2_LEFT_MOTOR, OUTPUT);
  pinMode(IN1_RIGHT_MOTOR, OUTPUT);
  pinMode(IN2_RIGHT_MOTOR, OUTPUT);
  pinMode(PWM_LEFT_MOTOR, OUTPUT);
  pinMode(PWM_RIGHT_MOTOR, OUTPUT);
  pinMode(EN_LEFT_MOTOR, OUTPUT);
  pinMode(EN_RIGHT_MOTOR, OUTPUT);

  pinMode(BUTTON_PIN, INPUT_PULLUP); // Para evitar poner una resistencia externa
  pinMode(LED_PIN, OUTPUT);

  setMotorsPWM(0,0);
  digitalWrite(EN_LEFT_MOTOR, HIGH);
  digitalWrite(EN_RIGHT_MOTOR, HIGH);
  Serial.begin(9600);
}
```

Es interesante mencionar que la entrada INPUT_PULLUP permite conectar una resistencia interna del pin BUTTON_PIN al voltaje HIGH del Arduino, simplificando la conexión del botón. Una terminal del botón se conecta al BUTTON_PIN y la otra a la terminal GND del Arduino. La función setMotorsPWM se explica a continuación.

5.2.2. Función principal para generar las señales de control PWM

Enseguida se presenta la función setMotorsPWM que recibe dos enteros en el rango de -255 a 255, el primero para definir la señal PWM del motor izquierdo y el segundo para definir la señal PWM del motor derecho. Con un valor positivo, la rueda gira hacia adelante y con un valor negativo la rueda gira hacia atrás y mientras más grande sea el número en valor absoluto, el motor gira más rápido.

```
void setMotorsPWM(int pwmLeftMotor, int pwmRightMotor)
{
  static int MAXPWM = 255;
  \textbackslash
```

```
    if(pwmLeftMotor > MAXPWM)         pwmLeftMotor = MAXPWM;
    else if(pwmLeftMotor < -MAXPWM)   pwmLeftMotor = -MAXPWM;
    if(pwmRightMotor > MAXPWM)        pwmRightMotor = MAXPWM;
    else if(pwmRightMotor < -MAXPWM) pwmRightMotor = -MAXPWM;
    if(pwmLeftMotor == 0) {
       digitalWrite(IN1_LEFT_MOTOR, HIGH);  digitalWrite(IN2_LEFT_MOTOR, HIGH);
     } else if(pwmLeftMotor > 0) {
       digitalWrite(IN1_LEFT_MOTOR, HIGH);  digitalWrite(IN2_LEFT_MOTOR, LOW);
     } else {
       digitalWrite(IN1_LEFT_MOTOR, LOW);   digitalWrite(IN2_LEFT_MOTOR, HIGH);
     }
    if(pwmRightMotor == 0) {
       digitalWrite(IN1_RIGHT_MOTOR, HIGH); digitalWrite(IN2_RIGHT_MOTOR, HIGH);
     } else if(pwmRightMotor > 0) {
       digitalWrite(IN1_RIGHT_MOTOR, LOW);  digitalWrite(IN2_RIGHT_MOTOR, HIGH);
     } else {
       digitalWrite(IN1_RIGHT_MOTOR, HIGH); digitalWrite(IN2_RIGHT_MOTOR, LOW);
     }
     if(pwmLeftMotor >= 0)
       analogWrite(PWM_LEFT_MOTOR,  pwmLeftMotor);
     else
       analogWrite(PWM_LEFT_MOTOR, -pwmLeftMotor);
     if (pwmRightMotor >= 0)
       analogWrite(PWM_RIGHT_MOTOR, pwmRightMotor);
     else
       analogWrite(PWM_RIGHT_MOTOR,-pwmRightMotor);
}
```

5.2.3. Protocolo de comunicación serial

Para comunicar el Arduino con el Odroid se utiliza un protocolo de comunicación serial a 9600bps, utilizando comandos de texto, terminados con el caracter de salto de línea: \n.

El protocolo contiene los comandos: **l** (por led) para encender y apagar el led, **b** (por botón) para leer el estado del botón, y **v** (por velocidad) para fijar las velocidades de las ruedas del robot. Enseguida se explican estos comandos en detalle.

- Comando **l** para encender y apagar el led del arduino. El siguiente comando apaga el led, mostrando en la segunda línea la respuesta del Arduino, una t (por terminado):

```
l 0
t
```

  El siguiente comando enciende el led:

```
l 1
t
```

- Comando **b** para leer el estado del botón de inicio. El siguiente comando pide leer el estado del botón de inicio. Si está presionado regresa un 1:

```
b
1
t
```

  Si no está presionado, regresa un 0:

```
b
0
t
```

- Comando **v** fija las señales de control PWM del motor izquierdo y derecho durante un intervalo de tiempo dado en milisegundos. El siguiente comando pide al robot avanzar con un PWM del motor izquierdo de 60 y un PWM del motor derecho de 80, durante 100 milisegundos, parando el robot al final de ese intervalo de tiempo:

```
v 60 80 100 0
1
t
```

  Si deseamos que avance con el mismo PWM de 80 en ambas ruedas, por 100 milisegundos, dejando ese mismo PWM en los motores, el último parámetro se fija en 1:

```
v 80 80 100 1
1
t
```

Si deseamos que retroceda con el mismo PWM de -80 en ambas ruedas, por 100 milisegundos y pare los motores al terminar ese tiempo, el comando deberá ser:

```
v -80 -80 100 1
1
t
```

5.2.4. Función loop en el Arduino

En la función loop de arduino, que se muestra a continuación, se implementa el protocolo de comunicación mencionado.

```
struct CMD {char cmd; int argv[10]; int argc; } comando;
struct CMD leeCmd(void)
{
  static struct CMD comando;
  char c;
  static int i, cont, digit, signo;
  i = 0;
  while( Serial.available() == 0) ;
  comando.cmd = Serial.read();

  while( Serial.available() == 0) ;
  c = Serial.read();
  while(c ==' ') {
    cont =  0;
    signo = 1;
    do {
       while( Serial.available() == 0) ;
       c = Serial.read();
       if (c == '\n' || c == ' ')
         break;
       if ( c == '-')
         signo = -1;
       else {
         digit = c - '0';
         cont = cont * 10 + digit;
       }
```

```
    } while(c != '\n' && c != ' ' );
    comando.argv[i] = cont * signo;
    i++;
  }
  comando.argc = i;
  return comando;
}

void loop()
{
  if( Serial.available() > 0) {
    comando = leeCmd();
    switch (comando.cmd) {
    case 'l' :                      //Comando "l 0" para apagar el led del micro
      if(comando.argv[0] == 1)      //        "l 1" enciende el led del micro
        digitalWrite(LED_PIN, HIGH);
      else
        digitalWrite(LED_PIN, LOW);
      Serial.print("t\n");
      break;
    case 'b' :                      //Comando "b" para detectar el estado del botón
      Serial.print(digitalRead(BUTTON_PIN));   Serial.print("\nt\n");
      break;
    case 'v' :                  //Comando "v pwm_izq pwm_der ms [01]"
      if(comando.argc >= 2)     //al final un 0 para el robot, un 1 no cambia los PWM
        setMotorsPWM(comando.argv[0], comando.argv[1]);
      if((comando.argc >= 3) && (comando.argv[2] >  0)) {
        delay((unsigned long)comando.argv[2]);
        if((comando.argc == 3) || ((comando.argc == 4) && (comando.argv[3] ==  0))) {
          setMotorsPWM(0,0);
        }
      }
      Serial.print("t\n");
      break;
    }
  }
}
```

La función auxiliar leeCmd implementa una lectura de comando inspirada en la forma como la función main de un programa en lenguaje C recibe los parámetros de la línea de comandos. Conviene mencionar que las funciones de biblioteca del Arduino Due para hacer esta tarea no funcionaron adecuadamente y se decidió implementar una función propia, leyendo caracter por caracter.

5.3. Hilos y Semáforos

Los ordenadores tienen una característica llamada multitarea. Esta característica [Tutorialspoint19] permite al computador ejecutar dos o más programas concurrentemente. Existen dos tipos de multitarea: basado en procesos y basado en hilos. El primero maneja la ejecución concurrente de programas. El segundo maneja la ejecución concurrente de pedazos de un sólo programa.
Un programa multihilos contiene dos o más partes que pueden ejecutarse concurrentemente. Cada una de estas partes del programa se le llama hilo. Para trabajar con hilos se necesita importar la librería **pthread.h**.
Para crear un hilo en el programa se utiliza la siguiente función:

```
pthread_create(pthread_t *thread,const pthread_attr_t *attr,
 void *(*start_routine)(void*),void *arg)
```

Donde **thread** es un identificador único del hilo regresado por la subrutina, **attr** es un atributo que puede ser usado para estalecer atributos en el hilo, **start_routine** es la rutina del programa que se ejecutará cuando se genere el hilo, y **arg** es un argumento que puede ser pasado a la subrutina.
Y para terminar un hilo se utiliza:

```
pthread_exit(void *status)
```

Para evitar problemas de sincronización al terminar de ejecutar el programa, se utiliza función:

```
pthread_join(pthread_t thread,void **status)
```

Esta función bloquea el hilo hasta que **thread** termine.
Un semáforo es una variable utilizada para el control de acceso a recursos compartidos por medio de multiples hilos. [geeksforgeeks19] Para poder utilizar semáforos se utiliza la

librería **semaphore.h**.
La variable del semáforo se declara como:

```
sem_t sem;
```

El semáforo se inicializa con la siguiente función:

```
sem_init(sem_t *sem,int pshared,unsigned int value)
```

Para bloquear o esperar al esperar el semáforo se utiliza la función:

```
sem_wait(sem_t *sem)
```

Para seguir o mandar la señal al semáforo se utiliza la función:

```
sem_post(sem_t *sem)
```

Y por último, para destruir el semáforo se utiliza la función:

```
sem_destroy(sem_t *sem)
```

Para el caso del robot se tienen tres hilos: el hilo principal, el de captura de imágenes y el de control de motores. La Figura 5.3 muestra como se lleva a cabo la ejecución del cada uno de los hilos.

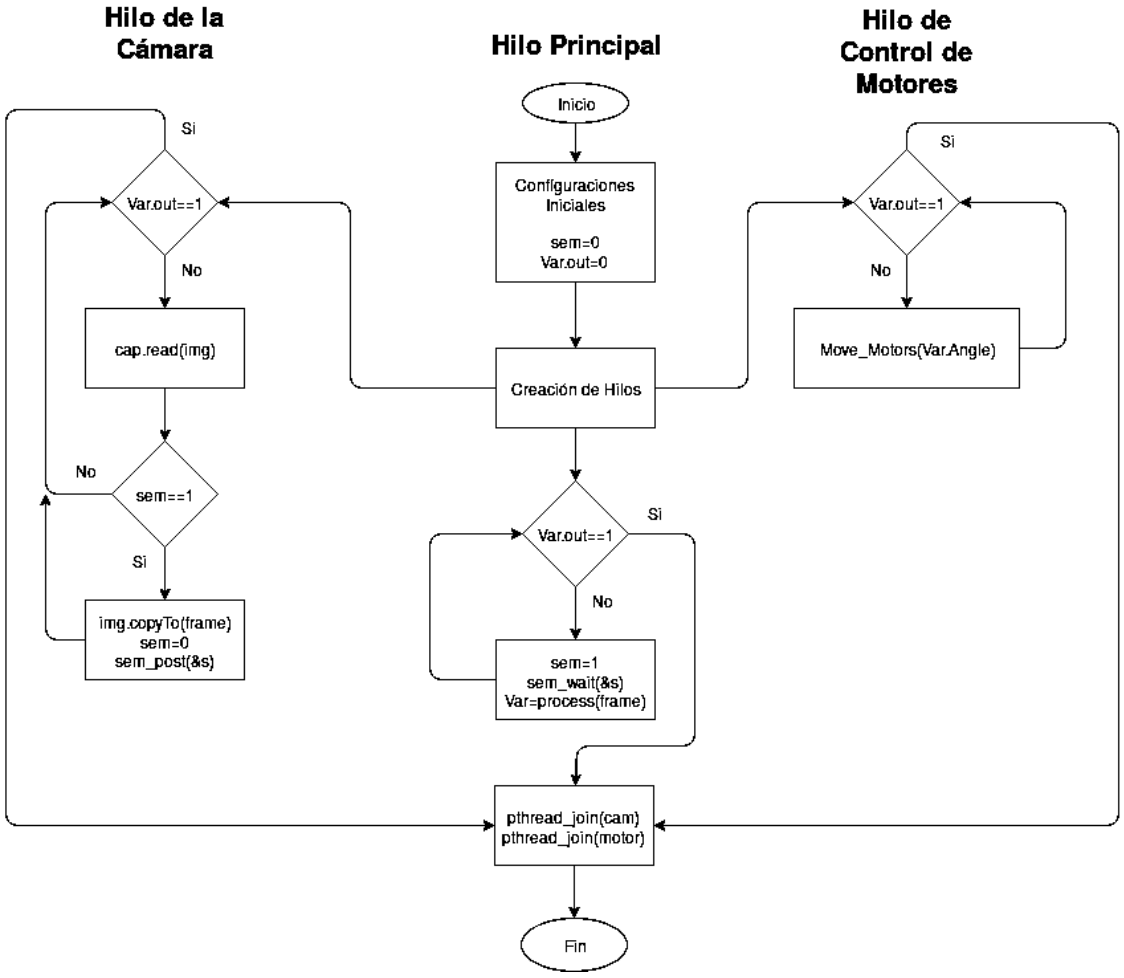


Figura 5.3: Diagrama de Flujo de los Hilos.

En la Figura 5.3, la variable **s** es el semáforo y **sem** es la variable que determina cuando se requiere obtener la siguiente imagen para procesarla. La imagen **frame** es la que se procesa para obtener las variables necesarias para el control de los motores y **img** es la imagen que continuamente estará obteniendo mediante la cámara. La función **process()** calcula las variables para el control de motores y **Move_Motors()** realiza la acción de control de los motores a partir de la variable **Angle**, que se encuentra dentro de **Var** la cual es calculada en la función **process()**. Cada ciclo depende de la variable **out**, que se encuentra dentro de **Var** también, para seguir ejecutándose y ésta determina si el robot se encuentra fuera del dojo.

5.4. Control de Motores

El control de los motores es una de las partes esenciales del robot para poder cumplir con su objetivo, ya que este determina que movimientos realizará el robot.
Para realizar el movimiento de cada uno de los motores se tiene una función a la cual se le pasan los parámetros de los PWM de cada motor (izquierdo y derecho). Para que el robot se mueva hacia enfrente el PWM debe ser positivo y para ir en reversa será negativo para ambos motores.
La función de control de motores recibe el ángulo al cuál el robot debe girar para quedar frente al oponente. Entonces, para el control de los motores se consideran los siguientes casos, donde α es el ángulo.

1. Cuando $\alpha > 180°$, significa que no se encontró al oponente, por lo tanto el robot gira lentamente en su propio eje para buscar al oponente. Esto se logra estableciendo el PWM de cada motor a la misma magnitud pero con diferente signo.

2. Cuando $\alpha = 0°$, ambos motores giran al mismo PWM hacia el mismo lado de tal forma que el robot se mueva hacia delante.

3. Cuando $0° < \alpha < 20°$ o $-20° < \alpha < 0°$ se aplica una función de control para calcular los PWM de cada motor. Esta función es la siguiente.
   Para $0 < \alpha < 20$:

$$PWM_{izquierdo} = PWM_{inicial} - \alpha * Inc$$

$$PWM_{derecho} = PWM_{inicial} + \alpha * Inc$$

   Para $-20 < \alpha < 0$:

$$PWM_{izquierdo} = PWM_{inicial} + \alpha * Inc$$

$$PWM_{derecho} = PWM_{inicial} - \alpha * Inc$$

   Donde $PWM_{inicial}$ es una velocidad constante e Inc es un incremento constante.

4. Cuando $20° < \alpha \leq 180°$ o $-180° \geq \alpha < 0°$ el robot gira en su propio eje hacia donde se encuentre el oponente.

Capítulo 6

Pruebas experimentales

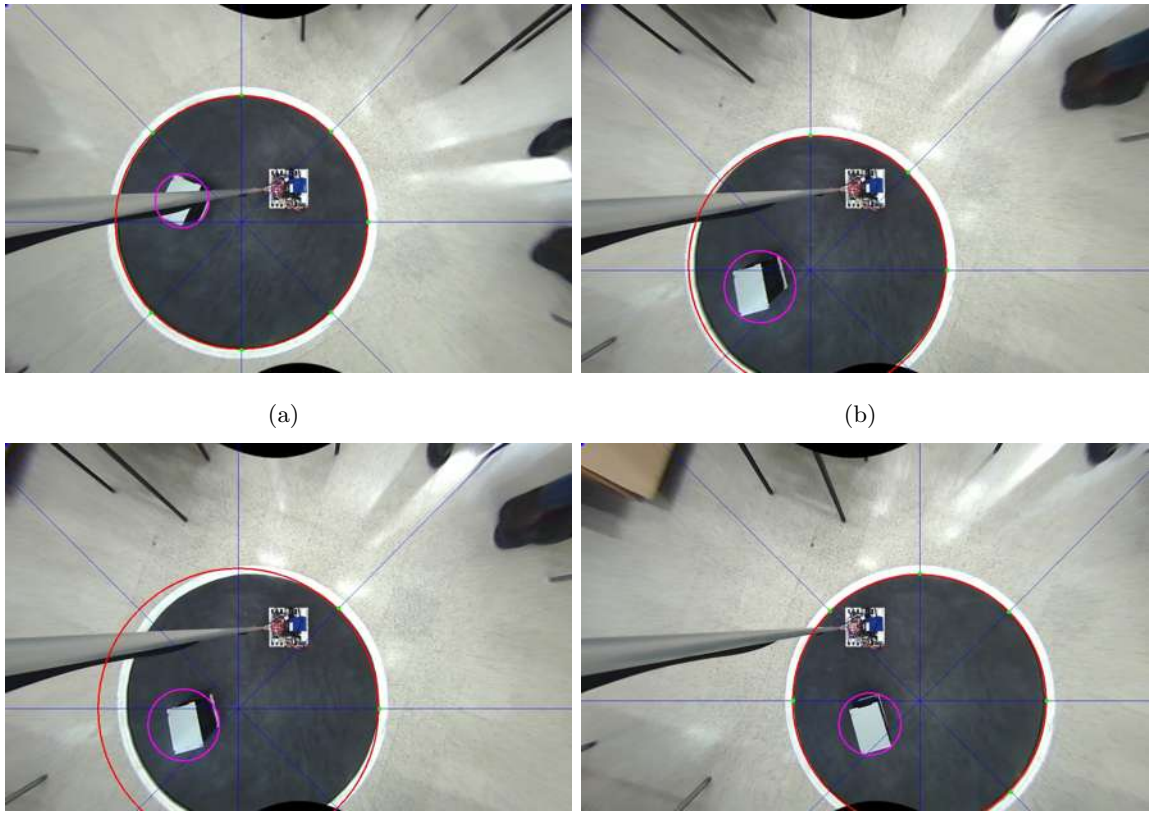
En este capítulo se realizan las pruebas finales offline del software desarrollado en este trabajo de tesis. Durante estas pruebas se ajustan ciertos parámetros para mejorar los resultados.

6.1. Pruebas Offline

Las pruebas *offline* son realizadas a partir de videos tomados por el robot de sumo en movimiento en la computadora ODROID-XU4 instalada en el robot. En estas pruebas se pretende observar el comportamiento de la rapidez con que se procesa cada imagen y la precisión con se detecta el dojo y al oponente.

6.1.1. Prueba 1

Para la primera prueba se establece un número de rayos para la detección del dojo igual a 8, por lo tanto el algoritmo RANSAC itera 17 veces. Se procesaron 22 imágenes en un tiempo de 7684.66ms, lo cual da una rapidez de 349.3ms por frame, o bien 2.86fps. En la Figura 6.1 se muestran algunos de los resultados.



(a) (b)

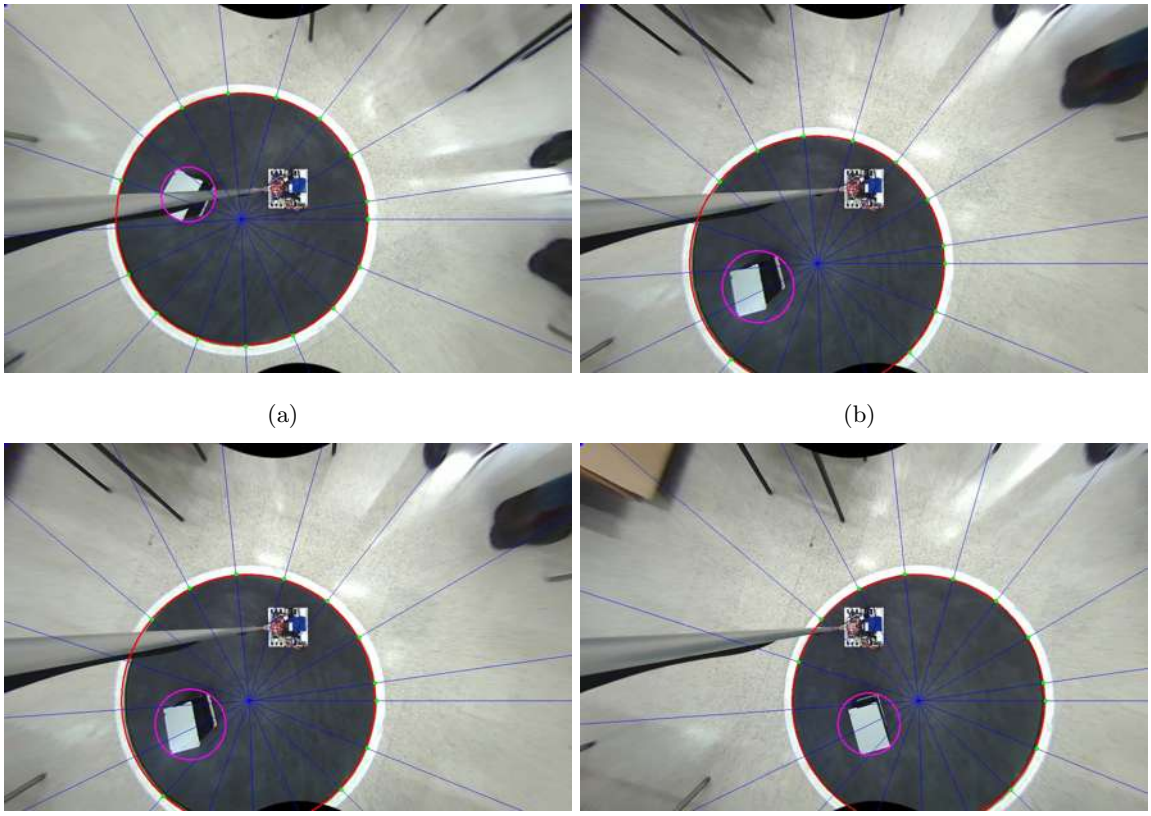
(c) (d)

Figura 6.1: Prueba 1

Las líneas azules son los rayos, los puntos verdes son los que se utilizan para calcular la circunferencia que se encuentra en color rojo y la circunferencia morada es donde se encontró al oponente.
Se observa que en algunas imágenes no se encuentran algunos puntos de los rayos, por lo que se opta por aumentar el número de rayos para tener más datos sobre la circunferencia y obtener resultados más precisos.

6.1.2. Prueba 2

Para la siguiente prueba, se utilizan 16 rayos, con lo cual itera 6 veces. Se registró un tiempo de 7158.36ms para el procesamiento de 21 imágenes. Esto da como resultado una rapidez de 340.87ms por frame, o bien 2.93fps. Algunos de los resultados se muestran en la Figura 6.2.



(a) (b)

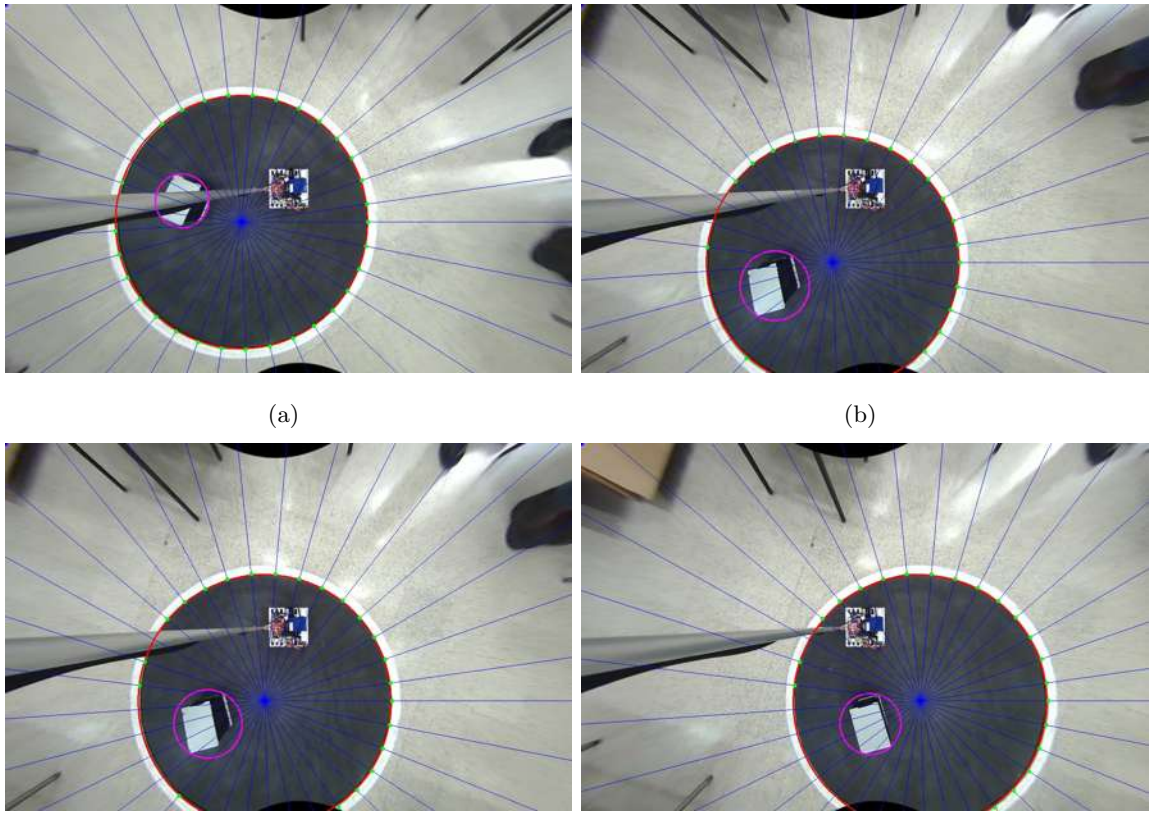
(c) (d)

Figura 6.2: Prueba 2

Los resultados tuvieron una mejora notable a comparación de las imágenes de la Figura 6.1. En todas las imágenes se obtenidas como resultado se tiene una precisión muy buena al detectar el dojo y al oponente.
Con respecto al tiempo, se obtiene un tiempo menor a cuando se implementaron 8 rayos ya que realiza menos iteraciones en el algoritmo RANSAC.
Se propone realizar pruebas utilizando el doble de rayos.

6.1.3. Prueba 3

En esta prueba se utilizan 32 rayos, con lo cual itera 4 veces. Se registró un tiempo de 7697.08ms durante el cual se procesaron 22 imágenes. Esto da como resultado una rapidez de 349.87ms por frame, o bien 2.86fps. Algunos de los resultados se muestran en la Figura 6.3.



(a) (b)

(c) (d)

Figura 6.3: Prueba 3

Se observa que, igual que la prueba anterior, la detección del dojo y el oponente es muy precisa. Sin embargo, se registró una rapidez menor al procesar las imágenes, aunque se realizaron menos iteraciones en el algoritmo RANSAC. Esto es debido a que se tienen que recorrer mayor número de rayos píxel por píxel.

Tomando en cuenta estos datos, se concluye que la mejor opción es utilizar 8 rayos para la detección del dojo.

A falta de tiempo, no se logró realizar las pruebas online sobre el robot físico.

Capítulo 7

Conclusiones y trabajos futuros

En esta tesis se presentó el desarrollo de un robot autónomo de sumo con la capacidad de ver su ambiente y detectar el dojo y el robot oponente con la finalidad de empujar al otro robot fuera del dojo.

7.1. Conclusiones

Se logró calibrar la cámara para remover la distorsión de las imágenes capturadas por la cámara con lente ojo de pez utilizando el modelo *fisheye* que brinda la librería de OpenCV.

Por medio de las imágenes sin distorsión se alcanzó el objetivo de detectar el área negra y blanca del dojo utilizando un modelo de distribución normal. También, se logró calcular el centro y el radio de la circunferencia que forma la franja de color blanco del dojo.

Se consiguió detectar al robot oponente dentro del dojo empleando los modelos de color obtenidos previamente. Se aplicaron filtros a las imágenes para obtener los contornos y así obtener la posición del robot oponente.

Se logró también obtener un ángulo de giro hacia la ubicación del robot oponente, para que el robot pueda empujarlo y sacarlo del dojo.

A causa de la falta de tiempo, no se realizaron pruebas *online*, esto es, con el robot completamente en operación. Sin embargo, se logró realizar las pruebas de la detección del dojo y el oponente y comprobar los cálculos de la posición del oponente y el ángulo de giro.

7.2. Trabajos Futuros

Algunos trabajos futuros que se pueden desarrollar para mejorar el funcionamiento del robot son los siguientes:

- Se utilizó un modelo de distribución normal para la detección de las áreas negra y blanca del dojo, utilizando imágenes en escala de grises. Se propone como trabajo futuro trabajar con imágenes en el espacio de color RGB o en el espacio de color HSV.

- Se propone implementar algún algoritmo que resuelva el problema de detección del oponente si éste se asemeja al área de combate y no se puede distinguir fácilmente. Una técnica que se puede tomar en cuenta para resolver este problema es que el robot tenga un láser y por medio de la luz emitida se puede ver en donde se refleja. En el lugar donde se observe este reflejo, posiblemente se encuentre el robot oponente.

- Se propone implementar mejoras a la inteligencia del robot. Una de estas mejoras es lograr la predicción de movimiento del robot oponente mediante los eventos pasados o imágenes obtenidas anteriormente.

- Otra mejora a la inteligencia del robot es establecer distintas estrategias de combate, las cuales son determinadas por las posiciones de los robots y los eventos pasados.

Referencias

[Bouguet15] Bouguet, J. Matlab calibration tool. http://www.vision.caltech.edu/bouguetj/calib_doc/, 2015. Consulta: Enero 2015.

[Bradski08] Bradski, G. y Kaehler, A. *Learning OpenCV*. O'Reilly Media, Inc, 2008. ISBN 978-0-596-51613-0.

[Derpanis10] Derpanis, K. G. Overview of the ransac algorithm, 2010. URL `http://www.cse.yorku.ca/~kosta/CompVis_Notes/ransac.pdf`

[Ditutor18] Ditutor. Circunferencia, 2018. URL `https://www.ditutor.com/geometria/circunferencia.html`

[ELP19] ELP. Cámara ov2710 lente ojo de pez 2mp, full hd. https://es.aliexpress.com, 2019.

[Flores11] Flores, P. y Braun, J. Algoritmo ransac: fundamento téorico, 2011. URL `http://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2011/keypoints/FundamentoRANSAC.pdf`

[Freud14] Freud, J. E. *Mathematical Statistics with Applications*. Pearson, 2014.

[geeksforgeeks19] geeksforgeeks. How to use posix semaphores in c language, 2019. URL `https://www.geeksforgeeks.org/use-posix-semaphores-c/`

[Hearn06] Hearn, D. y Baker, M. P. *Gráficos por Computadora con OpenGL*. Pearson Prentice Hall, 2006.

[Inbestme19] Inbestme. Desviación estándar, volatibilidad, riesgo, movimientos esperados e inesperados, 2019.

URL https://www.inbestme.com/blog/desviacion-estandar-volatilidad-riesgo-movimientos-esperados-e-inesperados/

[Jiang17] Jiang, K. Calibrate fisheye lens using opencv. https://medium.com/@kennethjiang/calibrate-fisheye-lens-using-opencv-333b05afa0b0, 2017.

[Kaehler16] Kaehler, A. y Bradski, G. *Learning OpenCV: Computer Vision in C++ with the OpenCV Library*. O'Reilly, 2016.

[OpenCV19a] OpenCV. Draw functions - opencv 2.4.13.7 documentation, 2019. URL https://docs.opencv.org/2.4/modules/core/doc/drawing_functions.html

[OpenCV19b] OpenCV. Opencv: Eroding and dilating, 2019. URL https://docs.opencv.org/3.4/db/df6/tutorial_erosion_dilatation.html

[OpenCV19c] OpenCV. Opencv fisheye camera model. https://docs.opencv.org/master/db/d58/group__calib3d__fisheye.html, 2019.

[OpenCV19d] OpenCV. Opencv: Structural analysis and shape descriptors, 2019. URL https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga8ce13c24081bbc7151e9326f412190f1

[OpenCV_3D19] OpenCV_3D. Camera calibration and 3d reconstruction, 2019. Date Updated: 2019-05-16. URL https://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html

[Robomatrix18] Robomatrix. Reglamento sumo, 2018. Date Updated: 2018-11-27. URL http://robomatrix.org/wp-content/uploads/2016/08/ReglamentoSumo.pdf

[Tutorialspoint19] Tutorialspoint. C++ multithreading, 2019. URL https://www.tutorialspoint.com/cplusplus/cpp_multithreading

[Zhong99] Zhong, Z. A categorization of multiscale-descomposition-based image fusion schemes with a performance study for a digital camera application. *Proceedings of the IEEE*, págs. 1315–1326, 1999.