

---

# Algoritmos de Visión por Computadora para el Reconocimiento de la Pose de Agentes Empleando Programación Orientada a Objetos y Multi- hilos

---

José Pablo Guerra Jordán



UNIVERSIDAD DEL VALLE DE GUATEMALA  
Facultad de Ingeniería



**Algoritmos de Visión por Computadora para el  
Reconocimiento de la Pose de Agentes Empleando  
Programación Orientada a Objetos y Multi-hilos**

Trabajo de graduación presentado por José Pablo Guerra Jordán para  
optar al grado académico de Licenciado en Ingeniería Electrónica

Guatemala,

2020

Vo.Bo.:

(f) \_\_\_\_\_  
Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) \_\_\_\_\_  
Dr. Luis Alberto Rivera Estrada

(f) \_\_\_\_\_  
XXX

(f) \_\_\_\_\_  
XXX

Fecha de aprobación: Guatemala, DIA X de MES XXX de 2020.

---

## Prefacio

---

El interés de este trabajo nace por explorar el área de robótica de enjambre y sus diferentes obstáculos, y explorar diferentes soluciones, entre ellas, la Visión por Computadora. Sin embargo, el área de robótica de enjambre es aún muy nueva, y por tanto existen diferentes retos todavía. Entre ellos está la detección de la pose de los agentes dentro de un espacio físico o mesa de pruebas.

Con este trabajo y sus resultados espero poder brindarle a futuros investigadores de esta área, herramientas para explorar diferentes aplicaciones de la robótica de enjambre. Además, espero brindar una herramienta versátil así como una visión general de una de tantas soluciones para ayudar al rendimiento de los programas mediante el uso de Programación Orientado a Objetos o multi-hilos.

Quisiera aprovechar este espacio para darle las gracias a las distintas personas que han estado conmigo durante este proceso y sin los cuales esto no hubiese sido posible:

- A Dios, por que de Él viene la sabiduría obtenida y los conocimientos aplicados en este trabajo.
- A mis padres, Israel y Cony, por su apoyo en cada etapa de mi vida, y especialmente en medio de esta pandemia, porque fueron una fortaleza para seguir adelante y culminar, no solo con este trabajo, sino muchos otros proyectos.
- A mi asesor, Luis Rivera, por sus múltiples consejos y apoyo en este trabajo y especialmente, por despertar en mí ese gusto por la programación que me ha llevado a explorar sus diferentes aplicaciones.
- A mis amigos, no solo de la carrera sino también fuera de ella, porque gracias a ellos hubo momentos de los cuales se pudo aprender y con su apoyo y consejo, también fueron parte importante de este proceso.

---

## Índice

---

<b>Prefacio</b>	<b>III</b>
<b>Lista de figuras</b>	<b>VII</b>
<b>Lista de cuadros</b>	<b>VIII</b>
<b>Resumen</b>	<b>IX</b>
<b>Abstract</b>	<b>X</b>
<b>1. Introducción</b>	<b>1</b>
<b>2. Antecedentes</b>	<b>3</b>
<b>3. Justificación</b>	<b>5</b>
<b>4. Objetivos</b>	<b>7</b>
<b>5. Alcance</b>	<b>8</b>
<b>6. Marco teórico</b>	<b>9</b>
6.1. Visión por Computadora . . . . .	9
6.1.1. OpenCV . . . . .	10
6.1.2. Calibración de Cámaras . . . . .	10
6.2. Programación Multi-hilos . . . . .	12
6.3. Programación Orientada a Objetos . . . . .	13
6.3.1. Objetos . . . . .	14
6.3.2. Clase . . . . .	14
6.3.3. Encapsulación . . . . .	14
6.3.4. Abstracción . . . . .	14
<b>7. Metodología</b>	<b>15</b>

<b>8. Prototipos de Mesa de Pruebas</b>	<b>17</b>
8.1. Primer Prototipo . . . . .	17
8.2. Segundo Prototipo . . . . .	19
<b>9. Pruebas preliminares en Python y C++</b>	<b>21</b>
<b>10. Pruebas Algoritmo y OpenCV en C++</b>	<b>25</b>
10.1. Calibración en C++ . . . . .	25
10.2. Identificación de Marcadores en C++ . . . . .	28
<b>11. Migración Código de C++ a Python</b>	<b>33</b>
11.1. Migración del Código para la Calibración de Cámara . . . . .	34
11.2. Migración del Código para Obtención de Pose e identificación . . . . .	41
11.2.1. Identificación del ID . . . . .	41
<b>12. Pruebas Algoritmo y OpenCV en Python</b>	<b>44</b>
<b>13. Pruebas de Generación de Marcadores y Detección en Python</b>	<b>46</b>
<b>14. Comparación entre C++ y Python</b>	<b>50</b>
14.1. Pruebas de detección de pose . . . . .	50
14.2. Comparación de rendimiento y tiempos de ejecución . . . . .	55
14.3. Interfaz Gráfica de Usuario . . . . .	56
<b>15. Conclusiones</b>	<b>57</b>
<b>16. Recomendaciones</b>	<b>58</b>
<b>17. Bibliografía</b>	<b>59</b>
<b>18. Anexos</b>	<b>61</b>

---

## Lista de figuras

---

1.	Representación de la información que extrae una computadora a partir de una imagen [7]. . . . .	10
2.	Proceso general para la extracción de información de una imagen para su uso en Visión por Computadora [8]. . . . .	10
3.	Conversión entre el plano de imagen al mundo real [12]. . . . .	11
4.	Patrón común para la calibración [14]. . . . .	12
5.	Ejemplificación del uso de hilos en un programa o proceso [16]. . . . .	13
6.	Primer prototipo de mesa. . . . .	18
7.	Primer prototipo de mesa y cámara. . . . .	18
8.	Armado de base de cámara para el segundo prototipo. . . . .	19
9.	Segundo prototipo de mesa de pruebas. . . . .	20
10.	Colocación para el segundo armado de la mesa de pruebas. . . . .	20
11.	Captura del primer archivo de texto para el ejemplo Multi-hilos . . . . .	22
12.	Captura del segundo archivo de texto para el ejemplo Multi-hilos . . . . .	22
13.	Captura del texto reconstruido para el ejemplo Multi-hilos . . . . .	23
14.	Captura de la cámara web con OpenCV . . . . .	24
15.	Intento 1 de calibración utilizando C++ . . . . .	26
16.	Intento 2 de calibración utilizando C++ . . . . .	27
17.	Detección y reescalamiento de código generado para tamaño de 2x2 cm . . . .	28
18.	Diagrama de flujo para la generación de un marcador [5]. . . . .	29
19.	Diagrama de flujo para la identificación de un marcador. . . . .	30
20.	Imagen original para la identificación de los marcadores. . . . .	31
21.	Primera prueba de identificación de código en la imagen capturada . . . . .	31
22.	Identificación del marcador . . . . .	32
23.	Ejemplo de imagen a calibrar en la migración del algoritmo. . . . .	34
24.	Proceso de calibración de la cámara en C++ [5]. . . . .	35
25.	Proceso de detección de bordes y definición de esquinas en Python . . . .	36
26.	Ejemplo de contornos identificados por el algoritmo de Python. . . . .	38
27.	Contornos identificados para la calibración en Python . . . . .	39
28.	Ejemplo de imagen calibrada utilizando el algoritmo de Python. . . . .	39

29.	Ejemplo de contornos identificados por el algoritmo de C++ . . . . .	40
30.	Ejemplo de imagen calibrada utilizando el algoritmo de C++ . . . . .	40
31.	Diagrama de flujo para el reconocimiento del ID en el algoritmo de Python. . . . .	42
32.	Prueba calibración de la cámara con Python . . . . .	45
33.	Segunda prueba calibración de la cámara con Python . . . . .	45
34.	Prueba de generación de código utilizando Python . . . . .	47
35.	Detección y reescalamiento de código generado para tamaño de $1 \times 1$ cm. . . . .	47
36.	Detección y reescalamiento de código generado para tamaño de $2 \times 2$ cm. . . . .	48
37.	Detección y reescalamiento de código generado para tamaño de $4 \times 4$ cm. . . . .	49
38.	Detección y reescalamiento de código generado para tamaño de $8 \times 8$ cm. . . . .	49
39.	Mesa de pruebas para la obtención de la posición real de los objetos . . . . .	50
40.	Frecuencia para la diferencia de la posición entre Python y C++ sin aproximación a entero . . . . .	52
41.	Frecuencia para la toma de pose en Python con respecto a la posición real . . . . .	53
42.	Frecuencia para la toma de pose en C++ con respecto a la posición real . . . . .	53
43.	Primer prototipo de interfaz de usuario. . . . .	56

---

## Lista de cuadros

---

1.	Comparación entre la detección de pose del algoritmo en <b>Python</b> y <b>C++</b> y la posición real, con aproximación a enteros . . . . .	51
2.	Comparación entre la detección de pose del algoritmo en <b>Python-C++</b> y la posición real sin aproximación a enteros . . . . .	51
3.	Frecuencia para la toma de pose con aproximación a entero (eje x y eje y) . .	52
4.	Media y moda para la posición con aproximación a enteros . . . . .	54
5.	Media y moda para la posición sin aproximación a enteros . . . . .	54
6.	Media y moda de la diferencia con respecto a la posición real del algoritmo en <b>Python</b> . . . . .	54
7.	Media y moda de la diferencia con respecto a la posición real del algoritmo en <b>C++</b> . . . . .	54
8.	Medidas de tiempo para los algoritmos usando 4 hilos para <b>Python</b> y 2 hilos para <b>C++</b> . . . . .	55
9.	Medidas de tiempo para el algoritmo en <b>Python</b> usando 2 hilos . . . . .	55
10.	Medidas de tiempo para el algoritmo en <b>Python</b> sin hilos . . . . .	55

---

## Resumen

---

En este trabajo se implementó un algoritmo de visión por computadora de una manera eficiente utilizando la Programación Orientada a Objetos (POO) y multi-hilos. Esto se realizó mediante el análisis y mejora del algoritmo desarrollado anteriormente en el lenguaje de programación C++, haciendo una migración hacia el lenguaje de Python. Las validaciones se realizaron mediante pruebas comparativas para obtener parámetros como el tiempos de ejecución, uso del CPU, entre otras, para determinar en qué puntos hubo mejoras, o bien, si ambos algoritmos tenían el mismo rendimiento. Además, se buscó validar que los resultados de las implementaciones fueran similares.

Se desarrolló una herramienta de *software*, cuyo objetivo principal es poder reconocer la pose de agentes (denominados así normalmente en el área de robótica de enjambre). La herramienta se combinó con una mesa de pruebas, la cual fue armada para poder realizar pruebas de la calibración de la cámara y la detección de pose. Existiendo ya una versión anterior de esta herramienta, con estas pruebas se pudo identificar los puntos o elementos de mejora que se tenían.

La herramienta de *software* que se desarrolló es versátil y muy útil para futuros proyectos en la línea de investigación de robótica de enjambre. Esto se debe a que, con el uso de multi-hilos, se aporta una mejora en el rendimiento del algoritmo. Además, el uso de la POO le da modularidad y facilidad al usuario para realizar mejoras futuras o nuevas implementaciones.

---

## Abstract

---

This work allowed to implement a computer vision algorithm in an efficient way using Object Oriented Programming (OOP) and multi-threads. This was done by analyzing and improving the algorithm previously developed in C++ language, migrating to the Python language. The validations were carried out through comparative tests to obtain parameters such as execution times, CPU usage, among others, to be able to determine in which points there were improvements, or if both algorithms had the same performance, as well as to validate that the results of the implementations were similar.

Therefore, a software tool was developed, the main objective of which is to be able to recognize the position of agents (so commonly called in the area of swarm robotics). The tool was combined with a testbed, which was assembled to help to perform tests on the calibration of the camera and that allowed to identify the improvement points of the algorithms.

The software tool that was developed is versatile and very useful for future projects in the swarm robotic research line. This is because, with the use of multithreading, an improvement in the performance of the algorithm is provided. In addition, the use of OOP gives modularity and help the user to make future improvements or new implementations.

# CAPÍTULO 1

---

## Introducción

---

La robótica de enjambre se ha convertido en un área de gran interés en los últimos años, sin embargo, aún tiene grandes retos que se deben abarcar. Dentro de estos retos está una implementación eficiente de algoritmos para su aplicación en visión por computadora, que es una herramienta de gran utilidad en esta área. Por tanto, se requiere tener herramientas versátiles que ayuden al investigador o usuario en el desarrollo de sus investigaciones.

Considerando lo anterior, se buscó encontrar el lenguaje orientado a objetos más adecuado que permita entregarle al usuario versatilidad en las herramientas de *software* utilizadas. Además de esto, se busca que los algoritmos que el investigador utilice (o desarrolle a partir de herramientas ya existentes) sean computacionalmente eficientes (como procesamiento de imágenes y obtención de datos) por lo que se utilizó la programación multi-hilos como un medio para mejorar el rendimiento de estos programas.

Para esto, utilizando el algoritmo desarrollado por André Rodas en la fase anterior, se realizaron mejoras a dicho algoritmo para ofrecer la versatilidad y cumplir con los objetivos planteados. Además, se validó la herramienta desarrollada en una mesa de pruebas similar a la Robotat de la Universidad Del Valle.

Este documento consta de una sección de objetivos, donde se introduce al lector a las metas que se pretenden lograr con este trabajo. Además, en la sección de antecedentes se presentan otros proyectos similares o aplicaciones en esta área, que fueron realizados anteriormente.

La metodología para este trabajo constituyó en una investigación previa para entender el funcionamiento de los lenguajes Python y C++, así como de la librería de OpenCV. Esto, con el objetivo de comprender la implementación de sus diferentes funciones dentro de la herramienta a desarrollar. Se procedió a implementar Programación Orientada a Objetos y con esto, se buscó mostrar el funcionamiento que la POO ofrece. Además, se buscó mostrar que la POO en conjunto con los multi-hilos puede ayudar a mejorar, tanto en rendimiento como en modularidad y escalabilidad, los programas que se desarrollen para el área de robótica de enjambre.

Luego de describir la metodología y los experimentos, se presentan los resultados obtenidos de las pruebas, validando así el alcance que se buscaba obtener en esta tesis. Finalmente, se presentan las conclusiones de este trabajo y se dan recomendaciones para futuras aplicaciones (como puntos de mejoras o sugerencias de uso).

# CAPÍTULO 2

---

## Antecedentes

---

### **Visión por Computadora Aplicado a Robótica de Enjambre**

En el documento de tesis de doctorado escrita por Luis Antón Canalís [1], titulada “Swarm Intelligence in Computer Vision: an Application to Object Tracking”, se describe cómo la visión por computadora se puede aplicar a la robótica de enjambre y a la orientación de sus agentes.

La robótica de enjambre va aplicada a la emulación del comportamiento de las colonias (por ejemplo de hormigas o abejas) con algoritmos por computadora. La mayoría de estos algoritmos han utilizado como herramientas diferentes algoritmos de visión por computadora. Esto significa que, por ejemplo, las imágenes emulan terrenos o espacios donde las colonias virtuales las recorren buscando información significativa basada en el procesamiento obtenido por visión por computadora.

Además de eso, en el documento “Visión artificial y comunicación en robots cooperativos omnidireccionales” [2] se detalla también la importancia del uso de visión por computadora (a lo que los autores se refieren como visión artificial) también para el reconocimiento de pose de agentes. Se denota el uso de Python como un lenguaje útil y simple para la programación, así como el uso de la librería OpenCV para la aplicación de algoritmos para visión por computadora.

## **Multi-hilos como una opción para mejorar el rendimiento**

Hiroshi Inoue y Toshio Nakatani proponen una comparación entre un modelo orientado a multi-hilos y otro orientado a multiprocesos en un procesador SMT (Simultaneous Multi-Threading). En el caso de utilizar todos los núcleos del procesador (8 para este caso) se obtiene que un programa multi-hilos es 3.4 % (con un máximo de 9.2 %) más rápido que el modelo multi-procesos. Esto se debe al paralelismo que ofrece los multi-hilos, lo cual es una ventaja [3].

Otro ejemplo lo provee Edward W. Felten y Dylan McNamee. Ellos realizaron un algoritmo para un modelo de comunicación de mensajes utilizando multi-hilos. Los principales resultados de estos experimentos fueron que el uso de multi-hilos puede reducir la carga del CPU, así como una mejora en el rendimiento del programa realizado [4].

Como es posible observar en los ejemplos anteriores, el uso de multi-hilos en las aplicaciones que lo permitan, ayuda a mejorar el rendimiento de un programa o algoritmo, ya que la reducción de tiempo puede llegar a ser considerablemente alta según sea el nivel de complejidad que tengan los algoritmos que se escriban llevando a obtener una respuesta o resultado de maneras más rápida y (dependiendo de la implementación) eficiente.

## **Continuación de la Fase I**

Este trabajo es la fase 2 del Proyecto de Graduación propuesto por André Rodas titulado “Desarrollo e implementación de algoritmo de visión por computador en una mesa de pruebas para la experimentación con micro-robots móviles en robótica de enjambre” [5]. La fase anterior consistió en la realización de un algoritmo que permitiera reconocer objetos dentro de una cama o mesa de pruebas para ser aplicada en robótica de enjambre. Dicha implementación se realizó utilizando OpenCV y el lenguaje C++, con el objetivo de obtener el mayor rendimiento de procesamiento, aunque también se realizó un análisis de otros posibles candidatos de lenguajes para su uso.

Para validar lo anterior, se realizaron una serie de pruebas para determinar el mejor identificador para el reconocimiento de objetos en dichas mesas. Agregado a esto, se realizaron pruebas con diferentes métodos de procesamiento de la librería OpenCV para obtener los mejores resultados y poder comparar con cuáles de estos se obtenía el mejor margen de reconocimiento y procesamiento de la imagen en la mesa de pruebas.

# CAPÍTULO 3

---

## Justificación

---

Hoy en día, las herramientas computacionales son de gran ayuda para las distintas actividades que se realizan en diferentes áreas, tanto de la industria como en las ramas científicas. Una de estas herramientas es la visión por computadora, que está enfocada en el uso de algoritmos para el procesamiento de imágenes, dándole la capacidad a la computadora de reconocer datos significativos que pueden ser orientados a diferentes aplicaciones. Su principal uso esta basado en la resolución de problemas o toma de decisiones que requieran gran poder de computo.

Sin embargo, estas tareas pueden representar un costo alto en recursos computacionales. Para esto, la programación multi-hilos puede ser una herramienta muy útil. Al tener varios hilos de procesamiento, es posible capturar y procesar datos de forma más eficiente. Esto permite que las computadoras (y las diferentes aplicaciones que se puedan realizar en estas) puedan llegar a resultados de manera más rápida y eficiente.

Al combinar la programación multi-hilos y la visión por computadora, es posible obtener reconocimientos de imágenes o entornos (como mesas de pruebas, mapas, entre otros) de manera mucho más adecuada, permitiendo utilizar estos datos en posteriores proyectos, como en la robótica de enjambre, por ejemplo.

La programación orientada a objetos agrega muchas otras ventajas a los procesos de computo. Primero, ofrece una reutilización del código, es decir, permite utilizar distintos métodos en diversas partes de un código y en variedad de proyectos y aplicaciones. Segundo, permite modificaciones de manera sencilla y práctica ya que se puede añadir o eliminar objetos según sea la necesidad de la aplicación, así como también fiabilidad, ya que es posible reducir códigos grandes en partes más pequeñas, permitiendo encontrar errores de manera más rápida y precisa.

Como se ha dicho, el área de investigación de robótica de enjambre ha crecido y tomado relevancia en la actualidad, por tanto, con este trabajo se busca tener un conjunto versátil de herramientas para que puedan ser aplicadas en diferentes tipos de proyectos o áreas de investigación. Su importancia radica en ofrecer mejoras a algoritmos propuestos anteriormente, buscando hacer más eficiente los diferentes procesos que se utilizan en la visión por computadora y la robótica de enjambre. Con esto, se busca ayudar a los investigadores a obtener resultados de manera más rápida y precisa en las diferentes ramas de aplicación.

# CAPÍTULO 4

---

## Objetivos

---

### Objetivo General

Mejorar el algoritmo de visión por computadora desarrollado para la mesa de pruebas Robotat para experimentación de robótica de enjambre, usando programación orientada a objetos, la librería OpenCV, y programación multi-hilos.

### Objetivos Específicos

- Seleccionar el lenguaje de programación orientado a objetos más adecuado para la implementación de algoritmos de visión por computador para la mesa de pruebas Robotat.
- Desarrollar algoritmos computacionalmente eficientes por medio de programación multi-hilos.
- Diseñar e implementar una herramienta de software para aplicaciones de robótica de enjambre, usando los algoritmos desarrollados.
- Validar la herramienta de software en la mesa de pruebas Robotat.

## CAPÍTULO 5

---

### Alcance

---

Para esta tesis se utilizó el algoritmo desarrollado previamente por André Rodas, y como resultado de este trabajo, se adaptó la versión desarrollada en el lenguaje C++ al lenguaje Python para tener una herramienta versátil (aplicando el algoritmo desarrollado), logrando tener distintas opciones según sea la aplicación o requerimiento del usuario. Además, dicho algoritmo permite ubicar la posición de los robots en un espacio físico, como por ejemplo la mesa de pruebas Robotat[5], esto con la ayuda de una cámara, por tanto, dentro de las funciones del algoritmo, está el poder calibrar la cámara. Además de la calibración, el algoritmo permite la toma de imágenes, procesamiento e identificación de los robots de manera paralela mediante el uso de multi-hilos.

El algoritmo tiene además la posibilidad de generar identificadores visuales para poder ubicar de mejor manera cada robot dentro de la mesa de pruebas. Con esto se logra saber (además de la posición) a que robot se está haciendo referencia cuando se obtengan su información.

Se realizaron diferentes pruebas para identificar robots en distintas posiciones, cada uno de estos con un identificador visual distinto y tamaños diferentes. Con estas pruebas se determinó que el tamaño de los identificadores va desde tamaños de  $3 \times 3$  cm hasta  $10 \times 10$  cm mientras las condiciones de luz sean las adecuadas (que la mesa este bien iluminada, por ejemplo) para alto contraste de los objetos sobre la mesa.

El alcance de este trabajo se vio limitado por la pandemia del Covid-19. Debido a esto, no se pudieron realizar las pruebas en la mesa de Robotat de la Universidad del Valle, como era el plan original. Sin embargo, estas pruebas se lograron realizar gracias a que se desarrolló una mesa alternativa para probar las funciones del algoritmo mencionadas anteriormente. Quedará para futuras fases hacer la validación en la mesa de la universidad.

# CAPÍTULO 6

---

## Marco teórico

---

### 6.1. Visión por Computadora

Visión por computadora se refiere al uso de cámaras o cualquier otro dispositivo de toma de fotografías o vídeos, para recolectar información para su posterior análisis, desarrollando algoritmos para hacer entender a la computadora que es lo que hay (en cuanto a datos o información significativa) en este tipo de archivos [6].

En otras palabras, consiste en obtener la información relevante, realizando un procesamiento a imágenes y vídeos, para que los seres humanos puedan entender de mejor manera que es lo que hay en ellos. Es decir, poder visualizar lo que una computadora hace en este tipo de procesamientos. Normalmente, este tipo de procesamiento de datos es utilizado para obtener información del medio o entorno (mapas, carreteras, imágenes de todo tipo, etc.) y poder ser utilizado en resolución de problemas o toma de decisiones por parte de una computadora, basado en su entorno o aplicación [6].

Un ejemplo claro de aplicación de este campo es en el área de Aprendizaje Automático (*Machine Learning*) o Inteligencia Artificial. En dichos campos de estudio la visión por computadora es utilizada para el procesamiento de imágenes y reconocer objetos, personas u otros elementos y con esto tomar una decisión o proceder a realizar alguna acción o tarea. Este tipo de tareas es un ejemplo aplicado en áreas como la conducción de autos autónomos

La figura 1 es un ejemplo de como una computadora procesa y entiende una imagen capturada y extrae de ella la información que le es útil. La figura 2 ilustra el procedimiento general que una computadora realiza para obtener dicha información.

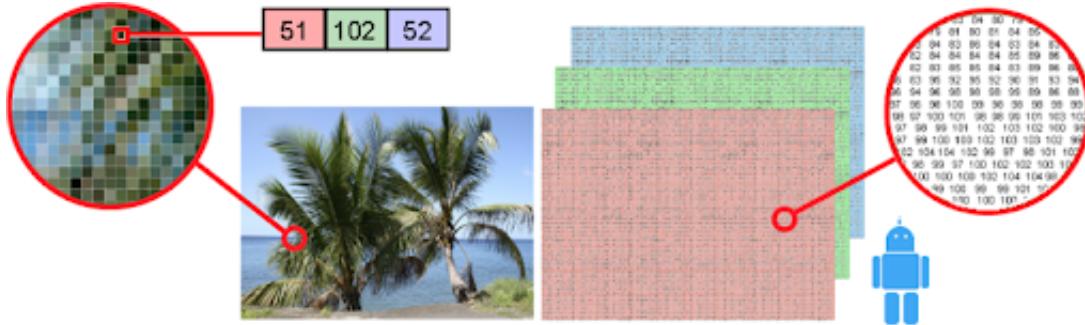


Figura 1: Representación de la información que extrae una computadora a partir de una imagen [7].

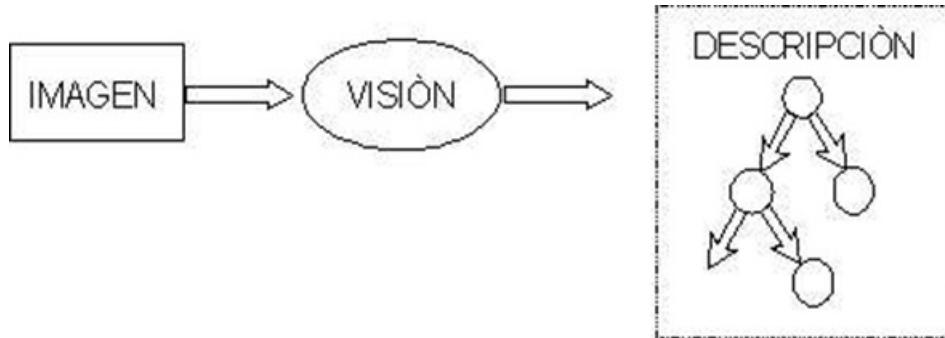


Figura 2: Proceso general para la extracción de información de una imagen para su uso en Visión por Computadora [8].

### 6.1.1. OpenCV

Esta es la librería de **Open Source Computer Vision Library** y está disponible para Windows, MacOS y Linux. Es una librería para visión por computadora y machine learning. Incluye mas de 2500 algoritmos que permiten ser utilizados para reconocimiento de rostros, identificación de objetos, clasificación del comportamiento humano, rastreo del movimiento de los ojos entre otros. Su implementación puede realizarse en C++, Python, Java y Matlab para las diferentes aplicaciones mencionadas. [9]

### 6.1.2. Calibración de Cámaras

La calibración de cámaras es una herramienta útil al momento de querer obtener información con respecto a una imagen. La calibración permite que la cámara pueda ser utilizada en aplicaciones como Realidad Aumentada, seguimiento y reconstrucción 3D entre otros[10]. Además, entre mejor se logre realizar dicha calibración, mucho más precisos serán las mediciones que se realicen a partir de esa imagen [11].

Básicamente, consiste en obtener los parámetros internos de la cámara como distancia focal, factores de distorsión y puntos centrales del plano imagen, así como también otros parámetros como píxeles, tamaño de imagen, por mencionar algunos. Con estos parámetros, se pretende establecer un marco referencial con respecto al mundo real, esto con el objetivo de unir el marco de referencia del mundo real con el de la imagen. Para esto, existen técnicas de calibración como la calibración fotogramétrica, que es la observación de objetos 3D con geometría conocida y buena precisión, utilizando los planos ortogonales es posible, mediante configuraciones elaboradas, obtener una calibración y resultados eficientes; así como la Autocalibración, que consiste en tomar una serie de fotos fijas para obtener los parámetros intrínsecos y extrínsecos de la cámara [10].

La figura 3 ilustra los parámetros intrínsecos de la cámara. Dentro de ellos se pueden observar el punto C, conocido como el punto principal. Este es la intersección entre el plano de la imagen y el eje óptico [12]. La imagen también muestra como por medio de este plano de imagen y el eje óptico es posible pasar a los ejes del mundo real, lo que ayuda a convertir lo que se ve en la imagen a lo que vemos físicamente y sus medidas.

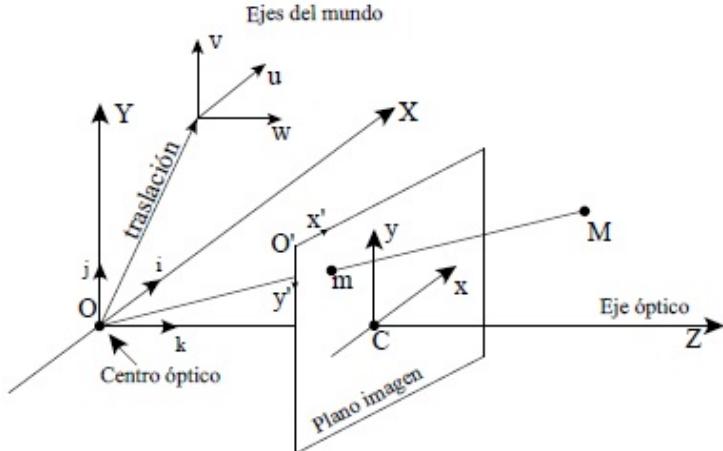


Figura 3: Conversión entre el plano de imagen al mundo real [12].

La figura 4 es un patrón común en la calibración de cámaras. Esto se debe a que el objetivo de este patrón es ubicar los puntos de interés (las esquinas de los cuadros) y con estos estimar los parámetros de la cámara [13]. Por tanto, el tablero de ajedrez es utilizado ya que debido a su diseño ayuda a determinar no solo posiciones o medidas de la imagen al mundo real, sino ayuda a obtener otros factores, como la distorsión de imagen, entre otros.

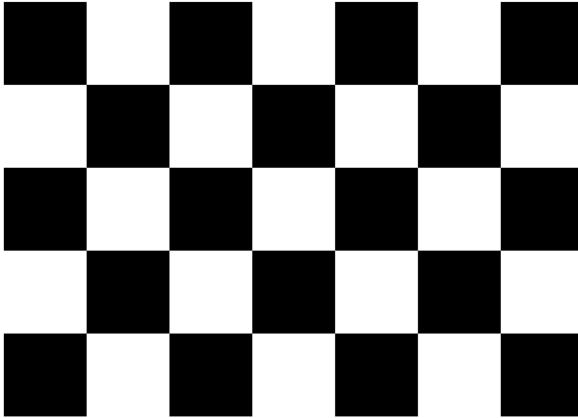


Figura 4: Patrón común para la calibración [14].

## 6.2. Programación Multi-hilos

Los procesadores orientados a multiprocesos, permiten realizar diferentes tareas al mismo tiempo. Estos a su vez, son responsables de manejar los recursos que se le asignen a cada uno de estos.

Más específicamente, cuando un programa es ejecutado, la computadora crea algo llamado **proceso** que contiene toda la información relevante del programa, como su identificador por ejemplo, hasta que dichos los programas terminan su ejecución.

Los sistemas operativos multiprocesos, permiten la ejecución de varios programas de manera simultanea y es la computadora la encargada de asignar los recursos a los diferentes programas que los necesiten. El objetivo principal es obtener un uso adecuado de los recursos del CPU [15].

Un hilo, por tanto, es una unidad de control dentro de un proceso. El programa corre un hilo principal o **main thread** encargado de crear otros hilos según sea su programación, siendo este es el principio básico del multi-hilos. Básicamente, orientar los programas ejecutados para realizar varias tareas y en muchos casos realizar procesos más eficientes [15].

Existen ventajas en el uso de multi-hilos. El primero, es la ventaja de tener un programa realizando captura o procesamiento de información, mientras otro está esperando estas salidas. Es decir, en lugar de tener un programa que espera una imagen o dato, para luego procesarla y dar un resultado, se pueden tener dos procesos corriendo, uno capturando datos y el otro procesando, por tanto, el resultado de esto será mucho más rápido. Además, la comunicación multi-hilos es mucho más eficiente que la comunicación entre procesos [15].

El figura 5 muestra como los hilos se pueden observar en un proceso o programa computacional. Como se ve, hay 3 hilos corriendo en paralelo, cada uno realizando funciones diferentes (procesamiento, extracción de datos, etc.) y corren a partir de un hilo principal. Esto ilustra como el uso de hilos en algoritmos que requieran varias tareas de procesamiento, puede ayudar a tener un mejor rendimiento, ya que no se tiene que esperar a que una tarea finalice para empezar otra, sino que permite ejecutarlas al mismo tiempo (dependiendo algunos factores propios del *hardware* y el *software* donde se implemente).

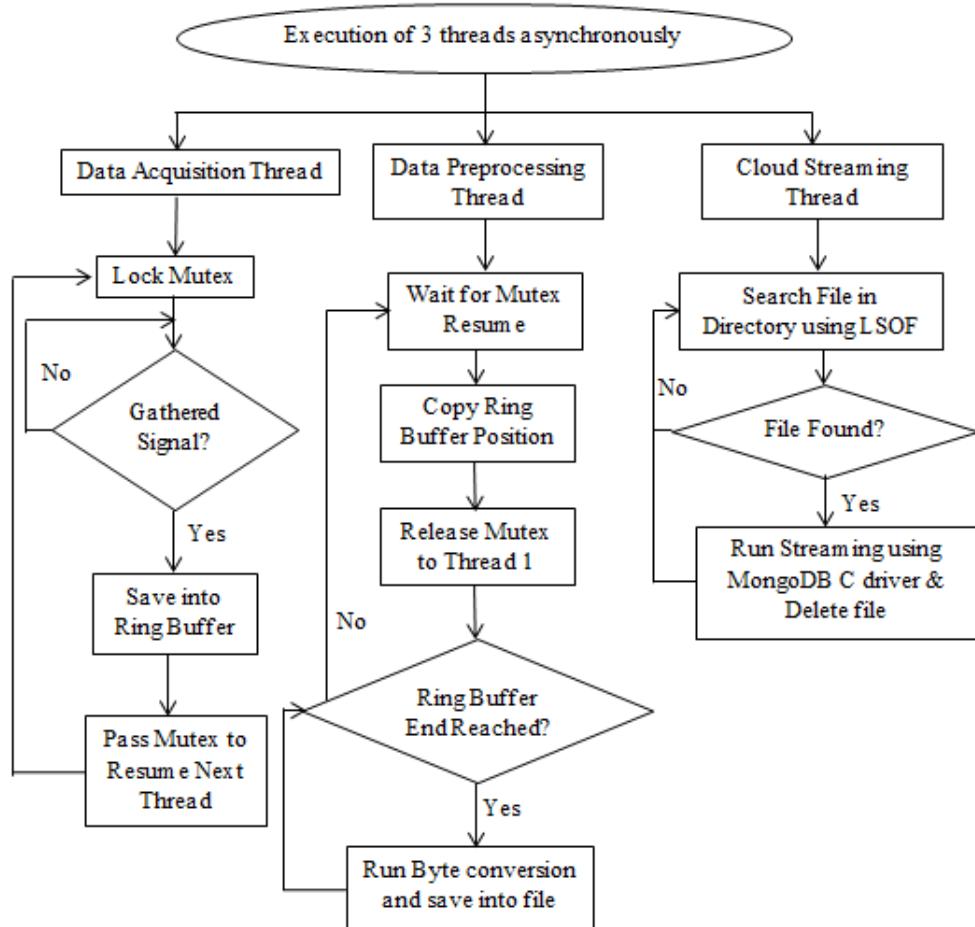


Figura 5: Ejemplificación del uso de hilos en un programa o proceso [16].

### 6.3. Programación Orientada a Objetos

Es un enfoque para la organización y el desarrollo de programas que intenta incorporar características más potentes y estructuradas para la programación. Es una nueva forma de organizar y desarrollar programas y no tiene nada que ver con ningún lenguaje de programación específico. Sin embargo, no todos los lenguajes son adecuados para implementar fácilmente los conceptos de la Programación orientada a objetos (POO) [17].

### **6.3.1. Objetos**

Un objeto en POO cumple básicamente las mismas funciones que un objeto de la vida real. Un objeto puede guardar atributos o características y ser utilizadas mediante métodos. Esto es útil porque permite de alguna forma esconder esta parte interna para solo enfocarse en su función principal (aquella para la cual fueron diseñados). A esto se le conoce como *encapsulación* [18].

### **6.3.2. Clase**

Una clase es el plano a partir del cual se crean métodos o atributos para los diferentes objetos individuales que pertenecen a esta clase. Es, lo que se podría llamar, como un molde o modelo para la creación de objetos [19] [18].

### **6.3.3. Encapsulación**

Esta es una de las características más importantes dentro de la POO. Básicamente se refiere a que todo lo referente a un objeto quede dentro de él, es decir, que solo se pueda acceder a sus propiedades mediante el uso de los métodos y propiedades que se proporcionen en la clase [20].

### **6.3.4. Abstracción**

Otra propiedad importante dentro de la POO es la abstracción. Como su nombre lo indica, es la capacidad de poder mostrar las características del objeto hacia el mundo exterior (en un programa, por ejemplo) pero quitándole la complejidad de dichos métodos. Es decir, que el usuario se ocupe únicamente de entender que funciones tiene más allá de entender como funcionen a lo interno [20].

# CAPÍTULO 7

---

## Metodología

---

### Investigación

Se buscó información referente a los lenguajes de programación para alcanzar los objetivos. El primer punto fue investigar el funcionamiento de la programación multi-hilos y la sintaxis para los lenguajes seleccionados (funciones, restricciones de uso, diferencias entre ellos, etc.). Además, se investigó todo lo referente a la librería de OpenCV para comprender su uso y realizar las pruebas que se requieren con la cámara.

Luego de investigar referente a la sintaxis y uso de multi-hilos, se buscó información sobre la programación orientada a objetos como una herramienta para el desarrollo de un algoritmo eficiente para aplicaciones de robótica de enjambre. De igual forma, con esto se logró tender su implementación para la aplicación en C++ y Python.

### Implementación

Una vez investigado lo referente a la programación multi-hilos y OpenCV, se procedió a realizar códigos de prueba para validar el funcionamiento de las diferentes aplicaciones que se necesiten implementar. Esto implica programas de ejemplo de aplicación multi-hilos, así como el uso de una cámara y OpenCV. Esto se realizó utilizando los lenguajes de C++ y Python

Luego de haber realizado y comprendido el funcionamiento, se procedió a analizar los programas existentes (implementados en el lenguaje C++) para encontrar puntos de mejora y poder aplicar lo investigado. Finalmente, se realizó una migración hacia el lenguaje Python como una primera opción de ofrecer versatilidad para el uso de la herramienta que se desarrolló. Una vez realizado esta migración, se buscó tener puntos de comparación para

determinar cual es el mejor lenguaje, si fuese el caso.

Además de esto, debido a la pandemia, se implementó una mesa de pruebas que buscaba ser utilizada como una herramienta para validar el uso de OpenCV y sus aplicaciones en el procesamiento de imagen. Por lo que, con la finalidad de reconocer el entorno para la pose de agentes y realizar calibraciones a la cámara, se utilizó esta mesa para realizar las pruebas necesarias. Dicha mesa tiene una dimensión de 28 cm × 14 cm.

## Validación

Finalmente, luego de tener dos elementos de comparación, se procedió a realizar pruebas para obtener parámetros que permitan medir y validar cual de las dos opciones es la mejor, o si ambas aportan de igual forma.

Uno de las primeras pruebas fue realizar comparaciones entre la programación multihilos de ambos lenguajes definidos. Esto con el fin de medir su rendimiento y comparar el resultado de las tareas que se dispongan en esta primera prueba.

La segunda prueba consistió en la calibración de la cámara y el procesamiento de imágenes. Utilizando OpenCV y la mesa de pruebas diseñada, se logró que de manera autónoma la computadora pueda reconocer la mesa y tomar acciones de calibración con la cámara. Para esto se tiene ya implementado un código en C++ (fase anterior de este trabajo) el cual se comparó con los resultados en Python realizando el mismo procedimiento. Esto permitió tener punto de comparación entre estos dos lenguajes.

Además de las pruebas anteriores, se realizaron comparaciones entre la posición que identificaban ambos algoritmos en los dos diferentes lenguajes y se compararon con la posición real. Esto permitió analizar la precisión y exactitud de los programas, así como ver el funcionamiento entre ellos. Dicha prueba se realizó en una mesa de tamaño de 26 × 16 cm.

# CAPÍTULO 8

---

## Prototipos de Mesa de Pruebas

---

### 8.1. Primer Prototipo

Con el objetivo de poder simular a escala la mesa de pruebas que se encuentra en el laboratorio de la UVG, se realizaron dos prototipos de dicha mesa. El primero se realizó con un tablero o pizarrón pequeño y una base robusta para colocar la cámara, como se observa en las figuras 6 y 7. Como se puede observar, se colocaron asteriscos en las esquinas de la mesa que representan los puntos para la calibración. La base, en conjunto con la cámara, estaba montada sobre la mesa de pruebas y así tener la visión superior de la mesa. Sin embargo, esto presentaba ciertos problemas en cuanto a la iluminación y captura correcta de las imágenes. Analizando esto, se realizó un segundo prototipo de dicha mesa, mejorando la base, como se muestra en la figura 8. Esto ayudó a tener una mejor perspectiva de la mesa, eliminando problemas de iluminación, como se ve en la figura 9



Figura 6: Primer prototipo de mesa.



Figura 7: Primer prototipo de mesa y cámara.

## 8.2. Segundo Prototipo

Como se mencionó, el primer prototipo de mesa tenía problemas, ya que la base de la cámara creaba una sombra sobre el tablero, lo cual era un problema para las pruebas, ya que en Visión por Computadora, la iluminación juega un papel importante. Por lo que, para este segundo prototipo, se procedió a conseguir un trípode en el cual montar la cámara (figura 8). Esto daba una mejor colocación de la cámara, además, evitaba el problema de la sombra sobre la mesa. Finalmente, se utilizó un cartón de dimensiones de 28 cm × 14 cm para simular la mesa y de igual forma se le colocaron puntos en las esquinas para referencia en la calibración (figura 9).



Figura 8: Armado de base de cámara para el segundo prototipo.



Figura 9: Segundo prototipo de mesa de pruebas.



Figura 10: Colocación para el segundo armado de la mesa de pruebas.

## CAPÍTULO 9

---

### Pruebas preliminares en Python y C++

---

Con el objetivo de validar el correcto funcionamiento de los lenguajes a utilizar (Python y C++) se realizaron algunas pruebas para entender su sintaxis y la aplicación de sus diferentes funciones. La primera prueba fue realizada en Python para verificar el funcionamiento de los multi-hilos, así como mostrar los resultados.

El objetivo era desarrollar un programa en lenguaje Python con multi-hilos, que tomara ordenadamente cada línea de cada archivo y lo ordenara en uno nuevo, como se muestra en la figura 13. Para esto se contaban con 2 archivos de texto. En la figura 11 se observa un primer archivo de texto del himno nacional de Guatemala pero con las líneas impares. Luego, en la figura 12, se encuentra la otra parte del himno, pero con las líneas pares.

Es posible observar como el programa logra exitosamente reordenar las líneas y formar el himno nacional adecuadamente. A pesar de ser un ejemplo sencillo, ilustra de buena manera como los hilos pueden ayudar a realizar múltiples tareas en un mismo proceso. Esto debido a que, en este programa, era necesario leer y escribir en un *buffer* común que posteriormente el tercer hilo iba a leer para ir escribiendo ordenadamente el archivo.

```
● ● ● Lab6_primer.txt
¡Guatemala feliz...! que tus aras
ni haya esclavos que laman el yugo

lo amenaza invasión extranjera,
a vencer o a morir llamará.
Libre al viento tu hermosa bandera
que tu pueblo con ánima fiera

tú forjaste con mano iracunda,
y la espada que salva el honor.
Nuestros padres lucharon un día
y lograron sin choque sangriento

colocarte en un trono de amor,
dieron vida al ideal redentor.
Es tu enseña pedazo de cielo
y iay! de aquel que con ciega locura

que veneran la paz cual presea,
si defienden su tierra y su hogar.
Nunca esquivan la ruda pelea
que es tan sólo el honor su alma idea

de dos mares al ruido sonoro,
te adormeces del bello Quetzal.
Ave india que vive en tu escudo,
¡ojalá que remonte su vuelo,

más que el cóndor y el águila real!
GUATEMALA, tu nombre inmortal!
```

Figura 11: Captura del primer archivo de texto para el ejemplo Multi-hilos

```
● ● ● Lab6_segundo.txt
no profane jamás el verdugo;
ni tiranos que escupan tu faz.
Si mañana tu suelo sagrado
libre al viento tu hermosa bandera

a vencer o a morir llamará;
antes muerto que esclavo será.
De tus viejas y duras cadenas
el arado que el suelo fecunda

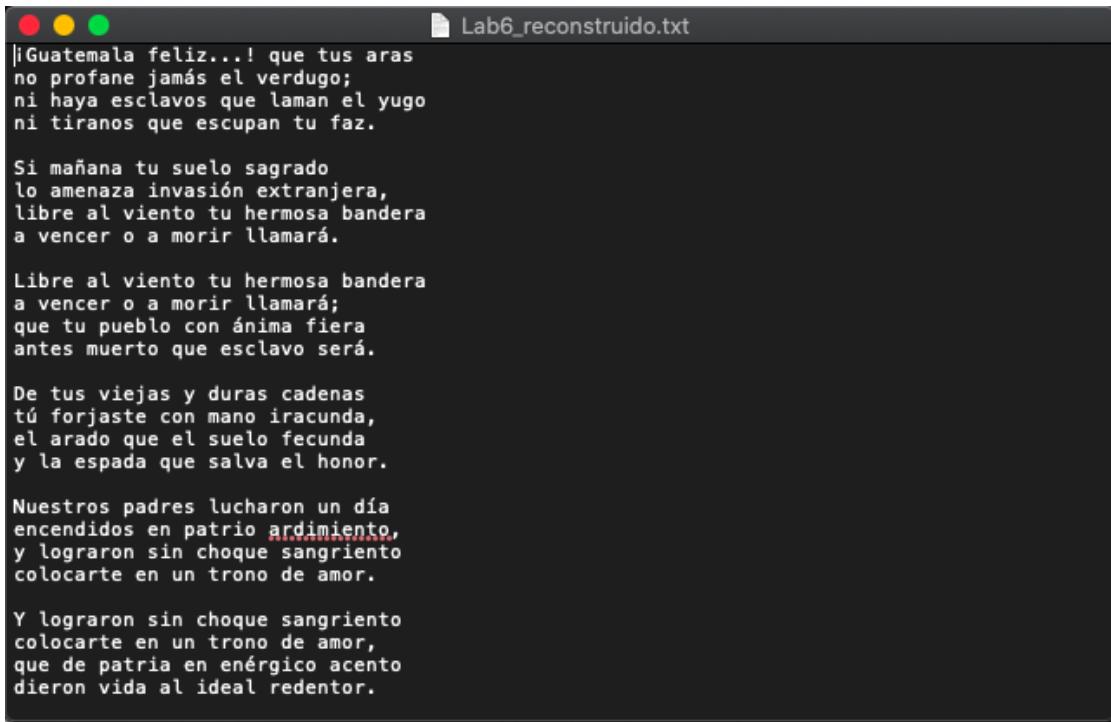
encendidos en patrio ardimiento,
colocarte en un trono de amor.
Y lograron sin choque sangriento
que de patria en energético acento

en que prende una nube su albura,
sus colores pretenda manchar.
Pues tus hijos valientes y altivos,
nunca esquivan la ruda pelea

si defienden su tierra y su hogar,
y el altar de la patria su altar.
Recostada en el ande soberbio,
bajo el ala de grana y de oro

paladín que protege tu suelo;
más que el cóndor y el águila real!
¡Ojalá que remonte su vuelo,
y en sus alas levante hasta el cielo,
```

Figura 12: Captura del segundo archivo de texto para el ejemplo Multi-hilos



The screenshot shows a terminal window titled "Lab6\_reconstruido.txt". The window contains a poem in Spanish, which is a reconstruction of the "Canción Nacional de Guatemala". The poem consists of several stanzas, each starting with a capital letter and followed by a colon. The text is in a monospaced font.

```
● ● ● Lab6_reconstruido.txt
¡Guatemala feliz...! que tus aras
no profane jamás el verdugo;
ni haya esclavos que laman el yugo
ni tiranos que escupan tu faz.

Si mañana tu suelo sagrado
lo amenaza invasión extranjera,
libre al viento tu hermosa bandera
a vencer o a morir llamará.

Libre al viento tu hermosa bandera
a vencer o a morir llamará;
que tu pueblo con ánima fiera
antes muerto que esclavo será.

De tus viejas y duras cadenas
tú forjaste con mano iracunda,
el arado que el suelo fecunda
y la espada que salva el honor.

Nuestros padres lucharon un día
encendidos en patrio ardor...,  
y lograron sin choque sangriento
colocarte en un trono de amor.

Y lograron sin choque sangriento
colocarte en un trono de amor,  
que de patria en energico acento
dieron vida al ideal redentor.
```

Figura 13: Captura del texto reconstruido para el ejemplo Multi-hilos

Otras de las pruebas realizadas en Python fue probar la librería de OpenCV para entender su funcionamiento y aplicar ciertas funciones que se usaron en el algoritmo propuesto. Para esta prueba se realizó un vídeo, es decir, la cámara estaba tomando un vídeo en tiempo real y se aplicaba un filtro de grises para luego mostrar la imagen en blanco y negro, como lo ilustra la imagen siguiente.



Figura 14: Captura de la cámara web con OpenCV

La gran ventaja de OpenCV es que permite realizar procesamiento de imagen y vídeo de manera simple, ahorrando tiempo al programador en el desarrollo de sus algoritmos. Este ejemplo ilustra significativamente el poder de OpenCV de manipular imágenes, además de vídeos en tiempo real, y el procesamiento y facilidad que ofrece la librería. La prueba se realizó en Python, como se mencionó, ya que el objetivo era realizar la migración desde C++ hacia este lenguaje, por tanto, era necesario ver la sintaxis de ciertas funciones de esta librería.

# CAPÍTULO 10

---

## Pruebas Algoritmo y OpenCV en C++

---

### 10.1. Calibración en C++

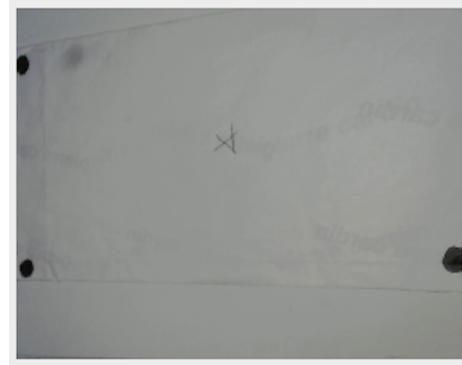
En primer lugar, se realizaron pruebas del algoritmo de calibración en C++. El objetivo principal de esta prueba es tomar como referencia cuatro figuras (en este caso los círculos marcados) en cada esquina y de la fotografía completa, tomar estos puntos como las nuevas esquinas de la imagen. Por tanto, una calibración correcta consistiría en la ubicación de estos puntos y tomarlos como nuevas esquinas.

Sin embargo, en la figura 15 se muestra una calibración fallida de la mesa (los puntos identificados no fueron colocados en la esquina de la imagen). Esto debido a unos parámetros de identificación basados en píxeles (como se muestra en el código 10.1). El objetivo principal de esta condición *if* que se describe en dicho código, consistía en ubicar el ancho y alto de las figuras que se ubicaran como posibles nuevas esquinas y comparar que estuvieran dentro de un rango de tamaño (como se muestra en la linea 7 con la variable *PixCircleValue*). La variable *PixCircleValue* estaba definida como un tamaño en píxeles que aproximadamente podía tener un contorno circular identificado.

El objetivo de esta condicional basada en los píxeles era evitar que figuras o contornos no deseados dentro o fuera de la mesa fueran tomadas como puntos de calibración para las esquinas de la imagen. Sin embargo, los píxeles pueden variar de manera diversa según sea la resolución de la cámara, la posición, iluminación, entre otros factores.



(a) Imagen original para ubicar las esquinas.



(b) Intento 1 de calibración, fallido.

Figura 15: Intento 1 de calibración utilizando C++

```

1
2 for (int i = 0; i < contours.size(); i++)
3 {
4
5 RotatedRect cod;
6 cod = minAreaRect(contours[i]);
7
8 if ((abs(cod.size.width - cod.size.height) < 2) && ((cod.size.height >
9 PixCircleValue - 3 && cod.size.height < PixCircleValue + 3) && (cod.size.
10 width > PixCircleValue - 3 && cod.size.width < PixCircleValue + 3))) {
11
12     if (a) {
13         for (int i = 0; i < 4; i++)
14             esquina[i] = cod.center;
15         a = false;
16     }
17     else {
18         for (int i = 0; i < 4; i++)
19             if (distancia2puntos(bordesMAX[i], cod.center) < distancia2puntos(
20                 bordesMAX[i], esquina[i]))
21                 esquina[i] = cod.center;
22     }
23 }
```

Código 10.1: Condición para la ubicación de las esquinas en la imagen.

Por lo que, para hacerlo de manera más generalizada a cualquier escenario, se procedió a eliminar esta condición y que se basará únicamente en la ubicación de los puntos en la imagen, como se ve en el código 10.2. Esta nueva condición recorría los contornos identificados y comparaba con las esquinas actuales de la imagen (línea 8 del código 10.2) hasta encontrar el contorno más cercano a los bordes originales e iba actualizando las esquinas como se observa en las líneas 14 y 15 del código 10.2. Así, luego de que el ciclo se completara, se tendrían los contornos mejor ubicados en las esquinas que se desearán tener.

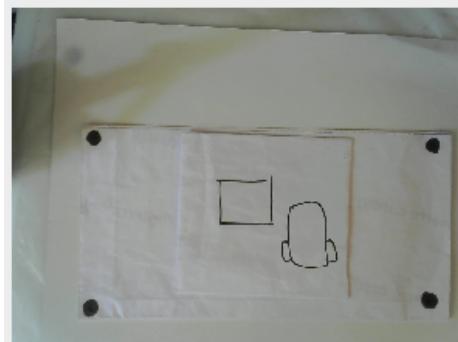
```

1
2 for (int i = 0; i < contours.size(); i++)
3 {
4     RotatedRect cod;
5     cod = minAreaRect(contours[i]);
6
7     if (a) {
8         for (int i = 0; i < 4; i++)
9             esquina[i] = cod.center;
10        a = false;
11    }
12    else {
13        for (int i = 0; i < 4; i++)
14    if(distancia2puntos(bordesMAX[i],cod.center)<distancia2puntos(bordesMAX[i],
15        esquina[i]))
16            esquina[i] = cod.center;
17    }
}

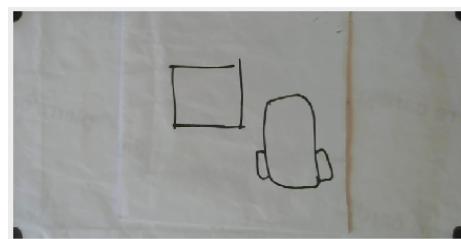
```

Código 10.2: Nueva condición para la ubicación de las esquinas en la imagen.

Finalmente, al implementar esta nueva condición, los puntos más cercanos (los marcados en negro en la figura 15a y 16a) serán tomados como las esquinas de la mesa y por tanto, la imagen se calibrara correctamente, como se puede observar en la figura 16b. Además de esto, se puede ver que hay objetos en la figura 16, pero gracias a que el algoritmo identifica los contornos y ubica los más cercanos a las esquinas de la imagen original, toma en cuenta los puntos negros marcados en las esquinas para la calibración.



(a) Imagen original a calibrar



(b) Imagen correctamente calibrada en los puntos deseados.

Figura 16: Intento 2 de calibración utilizando C++

## 10.2. Identificación de Marcadores en C++

Los marcadores en este capítulo hacen referencia a una imagen compuesta de una cuadrícula de  $3 \times 3$  cuadros, donde el cuadro superior izquierdo es un pivote marcado en color blanco, y los otros 8 cuadros pueden variar entre color negro y gris. El pivote sirve para poder posicionar la imagen correctamente para identificar el marcador, como se muestra en la figura 17, que el pivote inicia en la esquina superior derecha, pero luego de realizar la rotación, se coloca en su posición original en la esquina superior izquierda. Los restante 8 cuadros forman una combinación según el ID que se desee generar dependiendo el valor que se coloque (que puede ir desde 0 hasta 255). El flujo de creación de estos marcadores se muestra en la figura 18 y el proceso completo para la identificación del marcador se detalla en la figura 19.

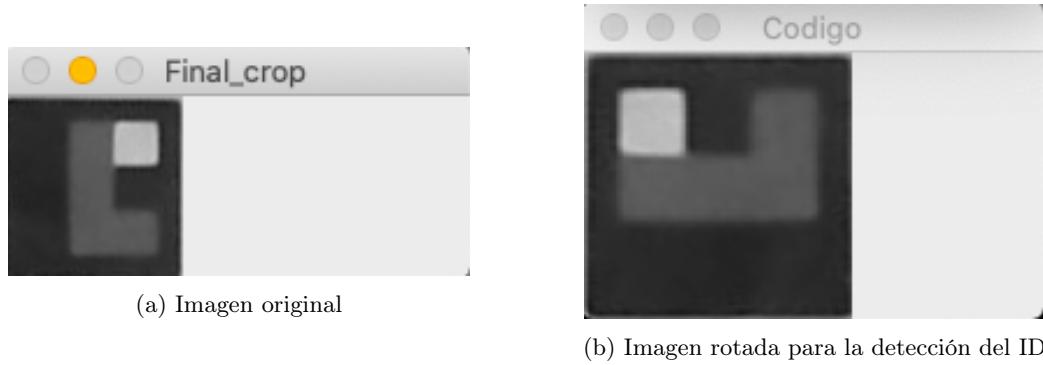


Figura 17: Detección y reescalamiento de código generado para tamaño de 2x2 cm

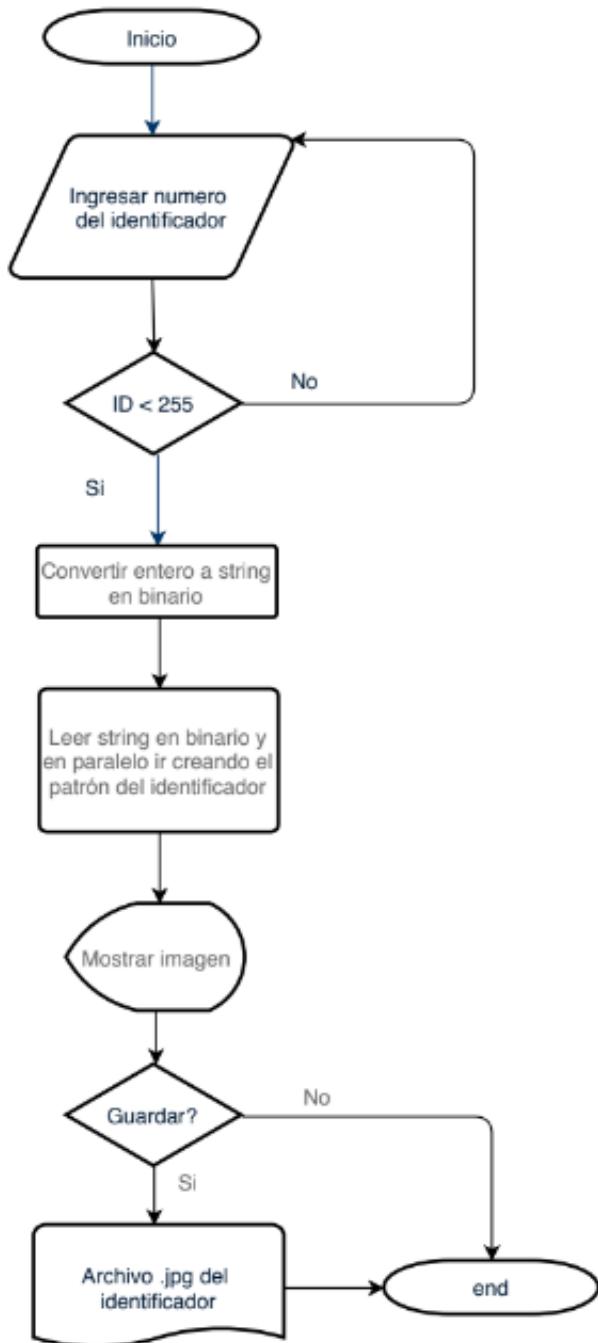


Figura 18: Diagrama de flujo para la generación de un marcador [5].

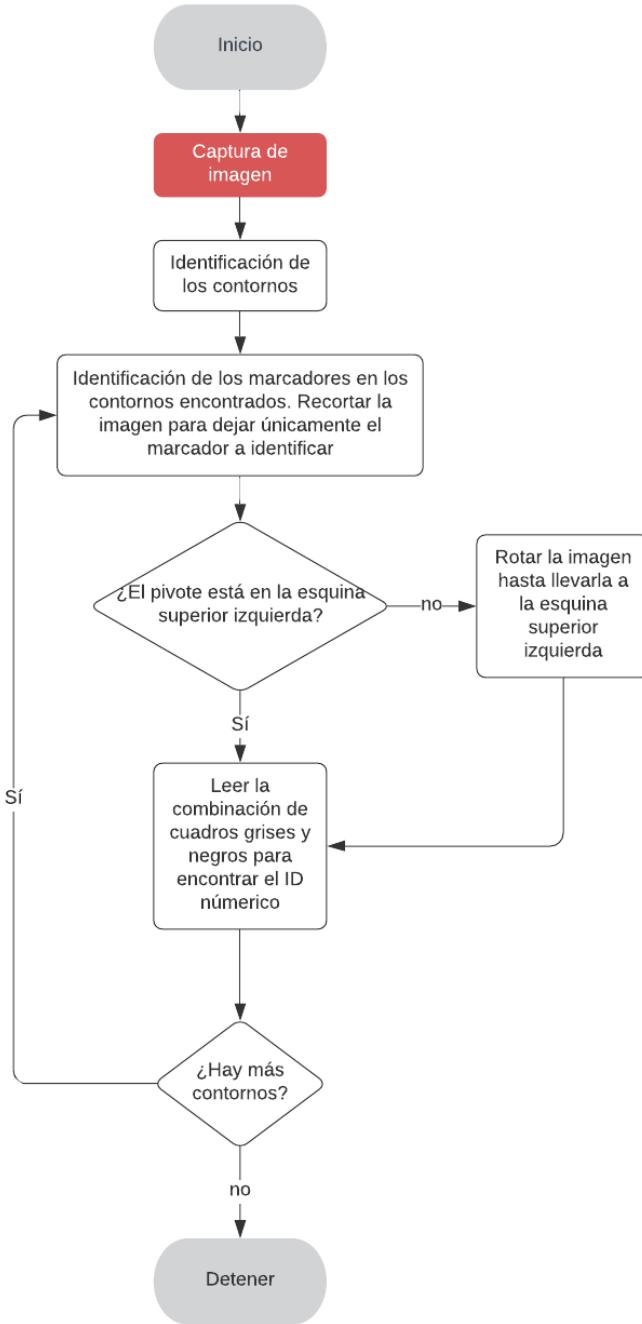


Figura 19: Diagrama de flujo para la identificación de un marcador.

Otras de las pruebas realizadas (utilizando el proceso descrito anteriormente) además de las pruebas de calibración, fueron las pruebas de identificación de marcadores. Estas pruebas consistían en capturar imágenes de la mesa con los marcadores para poder saber que ID representan y la posición en la mesa. La figura 20 muestra la imagen original con la que se realizó uno de los casos de pruebas. Como se observa en dicha figura, hay dos marcadores posicionados, uno representa 40 y el otro 170 (mostrados también en las figuras 22a, 22b y 21).

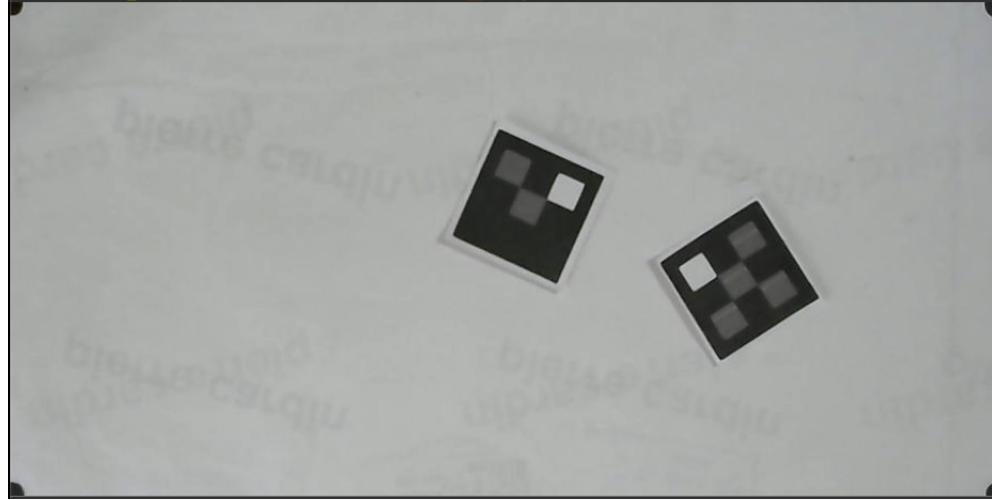


Figura 20: Imagen original para la identificación de los marcadores.

Como se muestra en la figura 21, luego de identificar al respectivo marcador, se procede a recortarlo de la imagen completa, rotarlo y poner el pivote (el cuadro blanco) en la esquina superior izquierda. Posterior a esto, se realiza la identificación obteniendo el valor de los cuadros negros y grises. El cuadro gris representa 1 y el negro representa un 0. Esto genera un número de 8 bits del cual se extrae el valor del ID.



Figura 21: Primera prueba de identificación de código en la imagen capturada

Finalmente, el programa muestra en una interfaz el último código que se identificó a manera de mostrar que el resultado fue el correcto. El resultado es un cuadro de imagen que se llama **ejemplo** como se muestra en la figura 22b, y en la figura 22a se muestra el valor del ID. Al ser, en este caso, el marcador con ID igual a 170 el último que se identificó de los dos, es el que se muestra en la figura 22 como ejemplo en la interfaz gráfica.

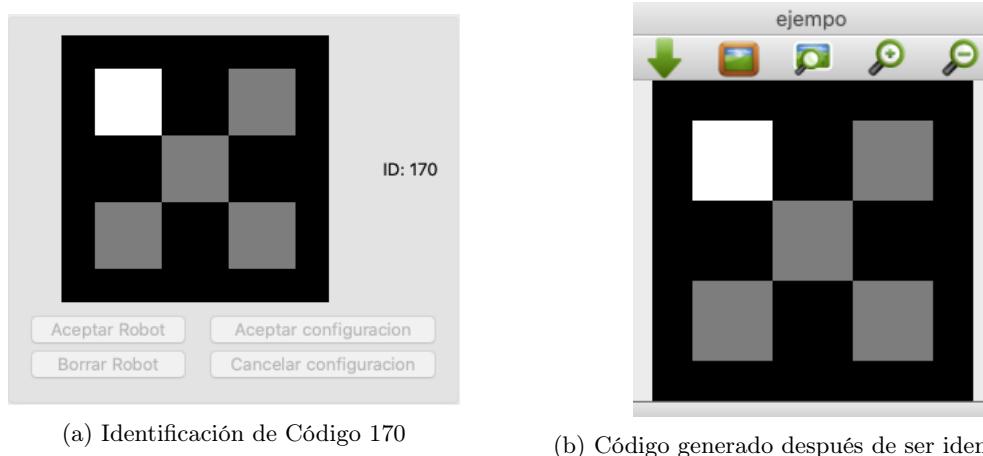


Figura 22: Identificación del marcador

Cabe mencionar que para validar estos resultados se hicieron varias pruebas (mostrando uno de los casos a manera de ejemplo). De todas las pruebas realizadas (variando entre ellas distintos valores de IDs) el porcentaje de identificación fue del 100 %. En las siguientes secciones se muestran otras pruebas variando el tamaño de los marcadores, y con esto, mostrar otros casos del funcionamiento de este programa.

# CAPÍTULO 11

---

## Migración Código de C++ a Python

---

En este capítulo se presenta resultados que mostrarán la migración que se realizó del lenguaje C++ a Python. El objetivo principal de esta migración es diversificar la herramienta que se desarrolló para la toma de pose de agentes. Se utiliza Python ya que es un lenguaje que muestra ser simple en la mayoría de sus funciones, y con esto, se logra presentarle al usuario de esta herramienta, diversas opciones según sea su necesidad o el lenguaje en el que desarrolle sus proyectos.

La implementación original en C++ consta de 3 partes: Calibración de la cámara, generación de marcadores y obtención de pose e identificación de los agentes. Por lo tanto, el objetivo de esta migración era trasladar estos programas a Python y tener una herramienta completa que tuviera todas estas funciones. Los detalles de esta migración se detallan en las siguientes secciones.

## 11.1. Migración del Código para la Calibración de Cámara

Como se ha venido presentando en capítulos anteriores, el objetivo principal del programa es la identificación de pose y calibración de la cámara. Por tanto, uno de los primeros pasos a realizar en esta migración fue la calibración de la cámara, utilizando el mismo método que ya se explico: Identificar contornos circulares en la imagen y a partir de estos, definir las nuevas esquinas de la imagen, por ejemplo, en la figura 23 y el proceso general de como se calibra la cámara en C++ se muestra en la figura 24 y el proceso para la calibración en Python se muestra en la figura 25.

Tanto para Python como para C++, el proceso es igual al que muestra la figura 24. Sin embargo, el proceso que se muestra en la figura 25 es únicamente de la detección de borde circulares y posición de las esquinas, ya que con esto se busca ilustrar los códigos que se mostrarán más adelante.



Figura 23: Ejemplo de imagen a calibrar en la migración del algoritmo.

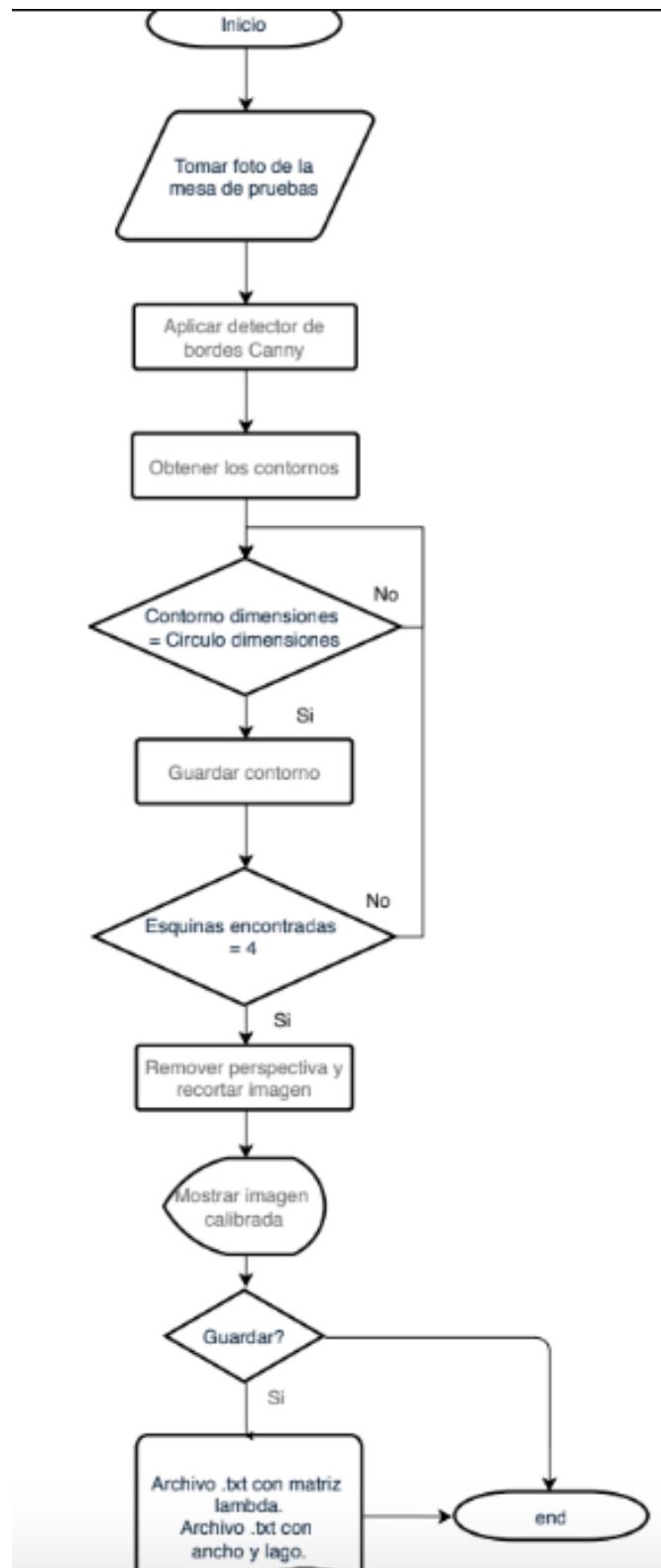


Figura 24: Proceso de calibración de la cámara en C++ [5].

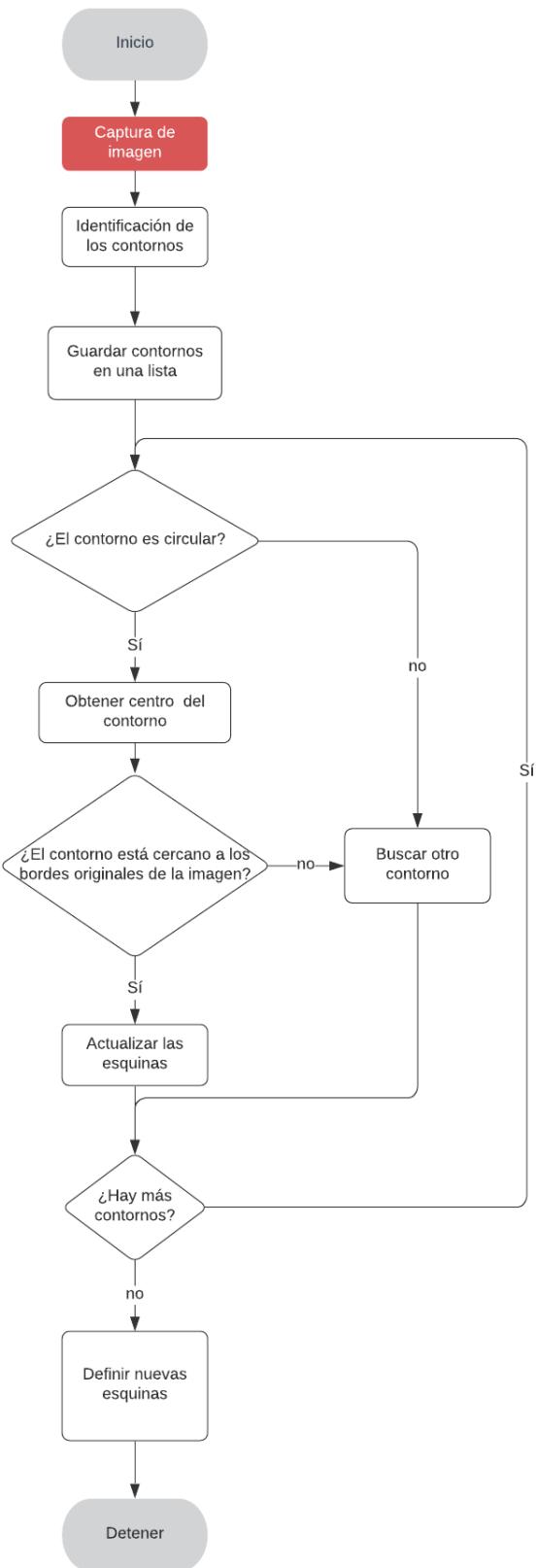


Figura 25: Proceso de detección de bordes y definición de esquinas en Python

El código 11.1 muestra una parte del algoritmo de calibración implementado en C++. En este código se muestra como se obtienen las esquinas mediante la función `minAreaRect` (línea 6). Esto encierra las figuras encontradas en pequeños cuadros (los más pequeños que los puedan encerrar) y con esto, se calcula el centro de dicho cuadro para compararlo con las esquinas iniciales y determinar si esta esquina está más cerca del borde de la imagen. Sin embargo, para Python se modificó dicho procedimiento.

```

1 Point bordesMAX[4] = { Point(0, 0) , Point(0, drawing.size().height) , Point
2   (drawing.size().width, 0) , Point(drawing.size().width, drawing.size().
3   height) };
4
5   for (int i = 0; i< contours.size(); i++)
6   {
7     RotatedRect cod;
8     cod = minAreaRect(contours[i]);
9     if (a) {
10       for (int i = 0; i < 4; i++)
11         esquina[i] = cod.center;
12       a = false;
13     }
14     else {
15       for (int i = 0; i < 4; i++)
16         if (distancia2puntos(bordesMAX[i], cod.center)<
17           distancia2puntos(bordesMAX[i], esquina[i]))
18           esquina[i] = cod.center;
19     }
20 }
```

Código 11.1: Detección de esquinas para calibrar cámara en C++.

El código 11.2 muestra una parte del programa realizado en Python como parte de la migración del algoritmo de calibración de la cámara. Para esta implementación lo que se hizo fue utilizar la función `cv.approxPolyDP()`. Esta función busca obtener una aproximación de los contornos identificados. Un ejemplo de contornos identificados se muestra en la figura 26.

Posteriormente, para detectar contornos lo más circular posible, se procedió a comparar el resultado de la aproximación y el área del contorno, si este estaba dentro del rango entonces se añadía a una lista que guardaba dichos contornos (líneas 2 a la 6 del código 11.2). Con esto, se obtienen los centros de cada contorno en la lista (línea 17 y 18 del código 11.2) y se comparan de igual forma con los esquinas predefinidas (línea 11) para finalmente determinar cual esta más cerca del borde de la imagen y tomarlo como la nueva esquina de la imagen calibrada.

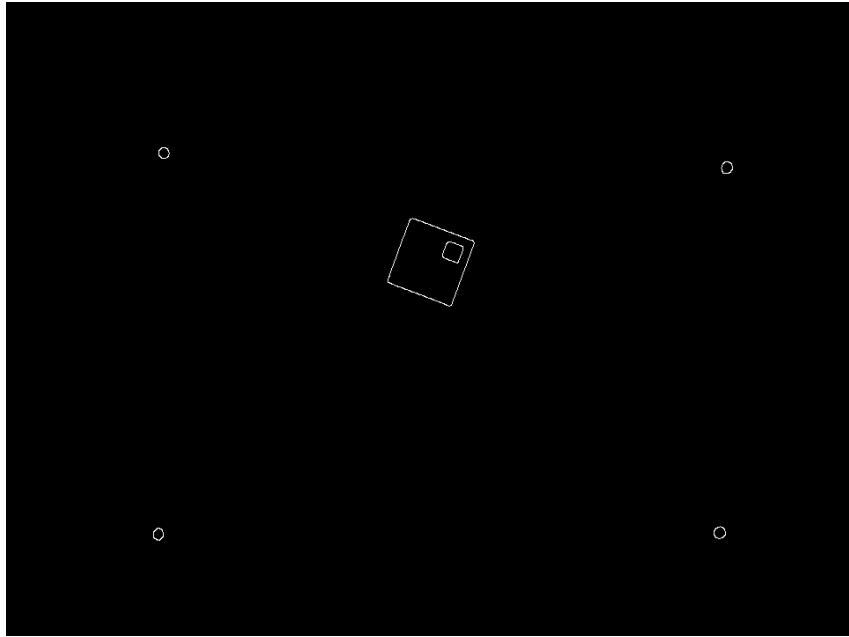


Figura 26: Ejemplo de contornos identificados por el algoritmo de Python.

```

1 for con in contour:
2     approx = cv.approxPolyDP(con,0.01*cv.arcLength(con,True),True) #utiliza
3         el metodo de aproximacion approxPolyDP para encontrar los contornos
4         circulares.
5     area = cv.contourArea(con) #calcula el area de este contorno.
6
7     if ((len(approx) > 8) & (area > 3) ): #Threshold aproximado, puede variar
8         si se desea para detectar circulos
9         contour_list.append(con) #si cumple, lo agrega a la lista
10
11 Cx = 0 #coordenada en x
12 Cy = 0 #coordenada en y
13
14 esquinas_final = [[1,1], [1,2], [2,1], [2,2]] #un valor inicial para la
15     comparativa
16 a = True
17 for c in contour_list: #recorre la lista de contornos para buscar el centro
18     # calcula el centro
19     M = cv.moments(c)
20 #ver documentacion para obtener mas informacion de como se calcula la
21     coordenada (x,y)
22     Cx = int(M["m10"] / M["m00"])
23     Cy = int(M["m01"] / M["m00"])
24 #agrega la primera esquina en la posicion inicial.
25     if (a):
26         esquinas_final[0] = [Cx,Cy]
27         a = False
28
29     for i in range (0,4):
30         if (distancia2puntos(boardMax[i], (Cx,Cy))<distancia2puntos(boardMax
31             [i], esquinas_final[i])):
32             esquinas_final[i] = [Cx,Cy]
```

Código 11.2: Detección de esquinas para calibrar cámara en Python

Las figuras 27, 28, 29 y 30 muestran los resultados de los algoritmos de calibración en Python y C++. En ambos casos, la calibración es correcta (como muestran las figuras), dando validez a ambos métodos para la detección de las esquinas. El objetivo de hacerlo así en Python era poder enfocarse en los contornos circulares únicamente, y además, ubicar los centros de estos contornos de manera más adecuada, para tener una mejor calibración de la cámara.

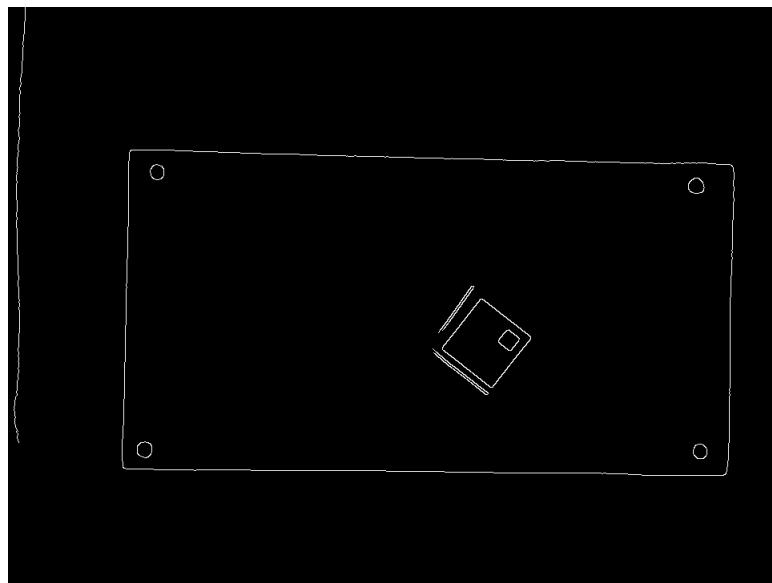


Figura 27: Contornos identificados para la calibración en Python

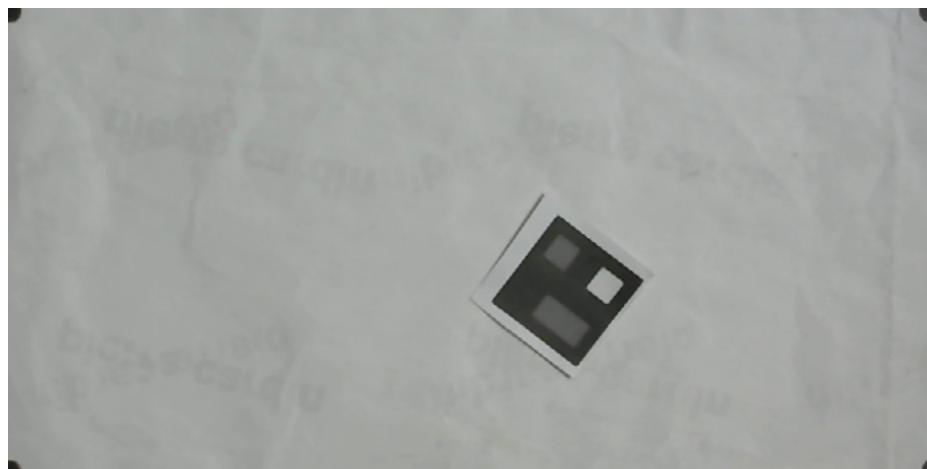


Figura 28: Ejemplo de imagen calibrada utilizando el algoritmo de Python.

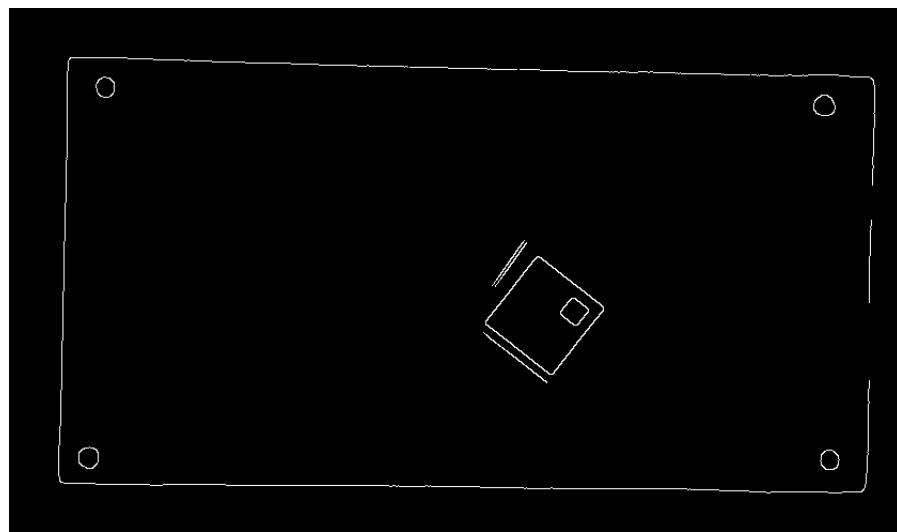


Figura 29: Ejemplo de contornos identificados por el algoritmo de C++

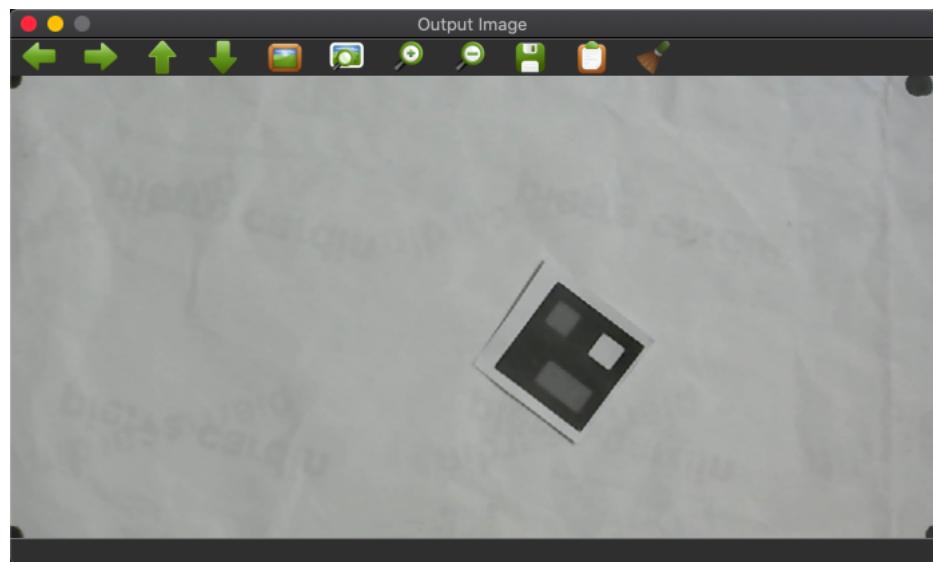


Figura 30: Ejemplo de imagen calibrada utilizando el algoritmo de C++.

## 11.2. Migración del Código para Obtención de Pose e identificación

### 11.2.1. Identificación del ID

La detección del ID se hace siguiendo el diagrama que se muestra en la figura 31, que muestra el proceso para el algoritmo en Python. El código 11.3 muestra como se obtiene el valor del ID utilizando el algoritmo en C++. La migración se realizó hacia Python utilizando de base este algoritmo.

Como se observa en el código 11.3, se utiliza un ciclo para recorrer la imagen para identificar el valor de cada cuadro correspondiente al ID. Sin embargo, en el algoritmo de Python (código 11.4) se divide la imagen. Lo que se hace es recortar la imagen del marcador para obtener cada cuadro que conforma el ID. El valor de este cuadro (que puede ir desde 0 hasta 255 por las condiciones de la imagen) se compara con una cota inferior y superior para determinar si esta dentro del rango que se requiere. Si esto es así, se coloca como 1, sino, es un 0. Estos finalmente se añaden a un array de 8 posiciones que forma un número binario.

Finalmente se convierte en un número entero para obtener el ID respectivo (línea 31). Como se puede ver, convertir el binario a un entero resulta más simple en la implementación de Python que en C++.

```
1 cout << "Encontrando los valores de la matriz" << endl;
2 for (int u = 1; u <= 3; ++u) {
3     for (int v = 1; v <= 3; ++v) {
4         int ValColorTemp = finalCropCodRotated.at<uchar>((finalCropCodRotated.size().height * u / 4), (finalCropCodRotated.size().width * v / 4));
5         MatrizValColores[(u - 1)][(v - 1)] = ValColorTemp;
6         if ((ValColorTemp < EscalaColores[2] - GlobalColorDifThreshold) && (ValColorTemp > EscalaColores[0] + GlobalColorDifThreshold)) {
7             EscalaColores[1] = ValColorTemp;
8         }
9     }
10 }
11
12 //Extraemos el codigo binario
13 cout << "Extraemos el codigo binario" << endl;
14 string CodigoBinString = "";
15 for (int u = 0; u < 3; ++u) {
16     for (int v = 0; v < 3; ++v) {
17         if ((u == 0) && (v == 0))
18             CodigoBinString = CodigoBinString;
19         else if ((MatrizValColores[u][v] > EscalaColores[1] - GlobalColorDifThreshold) && (MatrizValColores[u][v] < EscalaColores[1] + GlobalColorDifThreshold))
20             CodigoBinString = CodigoBinString + "1";
21         else
22             CodigoBinString = CodigoBinString + "0";
23     }
24 }
```

Código 11.3: Reconocimiento del ID en C++

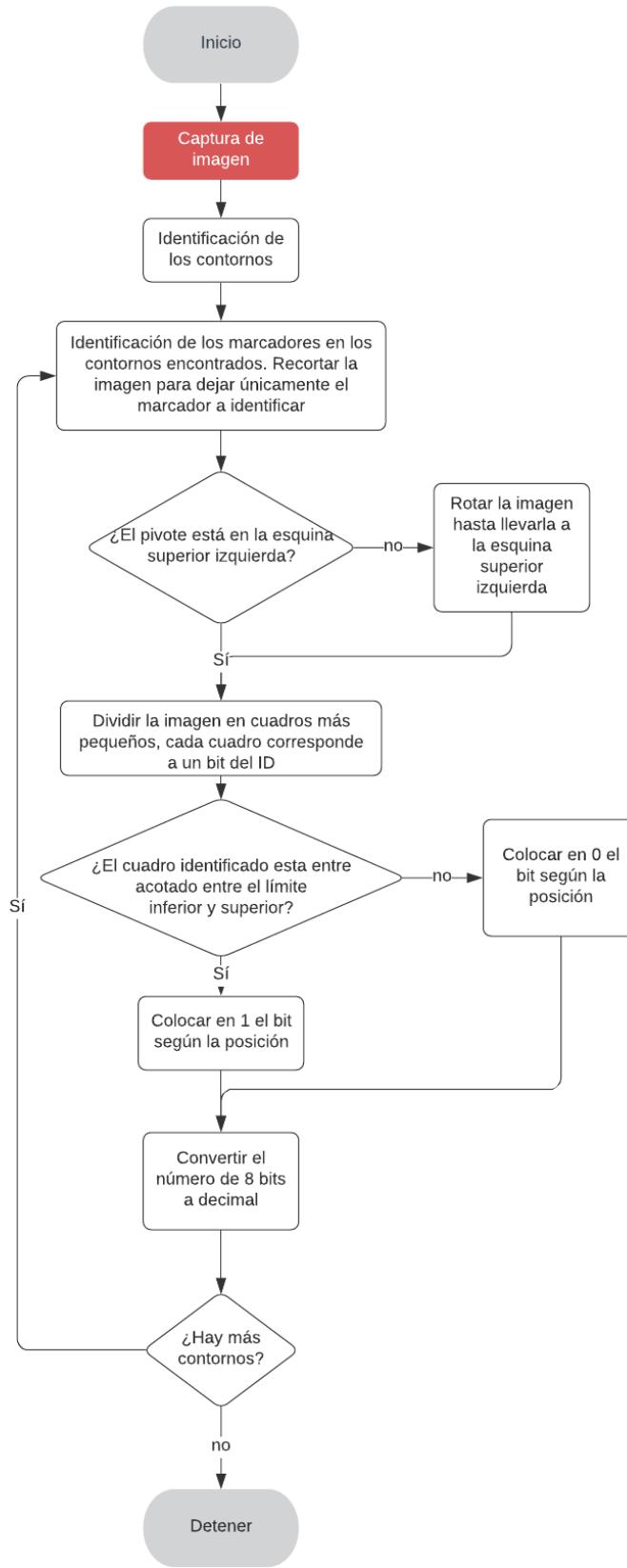


Figura 31: Diagrama de flujo para el reconocimiento del ID en el algoritmo de Python.

```

1 """
2 Luego, se ubica cada cuadro manualmente, es decir, desde a0 hasta a7, se
3 busca dentro de la imagen. Esto se hace manual para tener un
4 mejor control de la ubicacion y que sea mas facil cambiarlo si se desea.
5 Al igual que con las esquinas, se toma el valor medio del cuadro (tratando
6 de buscar justo el centro) para tener un valor
7 de referencia de que color se esta identificando (en este caso, negro o gris
8 para 0 o 1 respectivamente).
9 """
10
11 temp_a1 = resized[int(height_Final_Rotated*1/8):40, 65:100]
12 temp_a5 = resized[int(height_Final_Rotated*1/4 + 42):int(
13     height_Final_Rotated*1/2 + 40), int(height_Final_Rotated*1/8):int(
14     height_Final_Rotated*1/8 + 23)]
15 temp_a7 = resized[int(height_Final_Rotated*1/2) + 15:int(
16     height_Final_Rotated*1/2) + 45, int(width_Final_Rotated*1/2)+20 :int(
17     width_Final_Rotated*1/2) + 45]
18
19 ...
20
21 """
22 La extraccion del codigo se hace mas facil utilizando las funciones dentro
23 de python para el manejo de numeros binarios
24 y decimales. Se define un TRESHOLD_DETECT_MAX y TRESHOLD_DETECT_MIN para que
25 el valor que tome como gris se adecuado.
26 Estos valores pueden variar segun la luz (normalmente varia
27 TRESHOLD_DETECT_MIN)
28 """
29
30 i = 0 #para evitar alguna sobreescritura de esta variable.
31 for i in range (0, len(code)):
32
33     #con los thresholds establecidos, busca que valores sean grises y los
34     #cataloga como 1, sino, los cataloga como 0.
35     if code[i] > TRESHOLD_DETECT_MIN and code[i]< TRESHOLD_DETECT_MAX:
36
37         CodigoBinString = CodigoBinString + "1"
38     else:
39         CodigoBinString = CodigoBinString + "0"
40
41     ...
42
43
44     #esta funcion pasa el string de bits a formato de numero int.
45     tempID =int(CodigoBinString , 2)

```

Código 11.4: Reconocimiento del ID en Python

## CAPÍTULO 12

---

### Pruebas Algoritmo y OpenCV en Python

---

Para las pruebas en **Python**, primero se realizó la migración del código (o algoritmo) planteado en **C++**. Una vez hecha dicha migración, se realizaron pruebas para validar su funcionamiento.

Una de las primeras pruebas son las que se muestran en las figuras siguientes. El objetivo principal del algoritmo en **Python** era identificar figuras o contornos circulares (como lo muestra la figura 32) y al final, cuando estos contornos fueran identificados, se procedía a ubicar los más cercanos a los bordes de la imagen y estos eran tomados como las nuevas esquinas (como se ejemplifica en la figura mencionada)

Otra de las pruebas fue realizar la misma calibración simulando tener objetos en la mesa. La figura 33 muestra un dibujo que simula ser un robot en la mesa. Como se observa, el algoritmo lo identifica, pero al no estar cerca de los bordes de la imagen, no es tomado como una esquina para la calibración.

Como se ha mencionado en ca

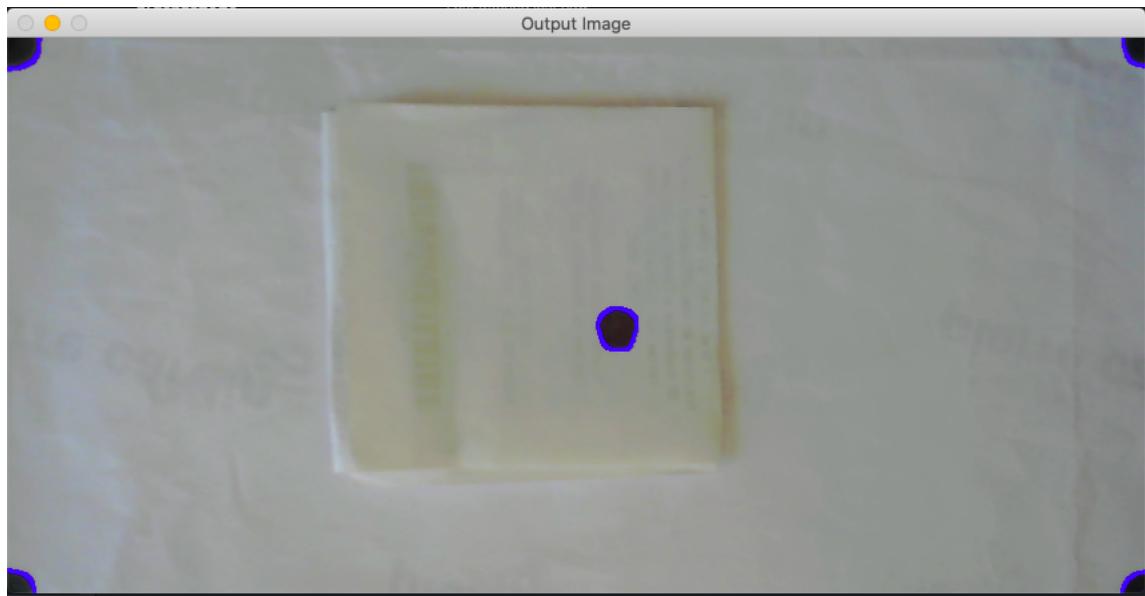


Figura 32: Prueba calibración de la cámara con Python

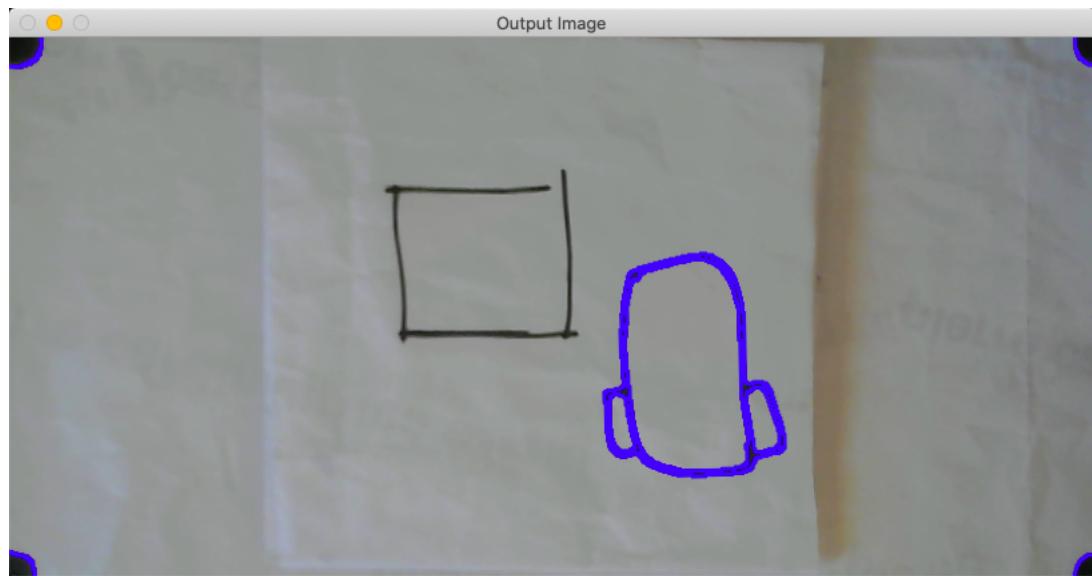


Figura 33: Segunda prueba calibración de la cámara con Python

## CAPÍTULO 13

---

### Pruebas de Generación de Marcadores y Detección en Python

---

La creación de los marcadores fue realizado también migrando el código en C++. Este genera un código entre 0 y 255 que posteriormente es mapeado a una imagen como se muestra en la figura 34, que representa el número 40. Luego de eso, la impresión y colocación de ese marcador queda a discreción del usuario (tanto en tamaño como en posición).

Finalmente, las figuras 35, 36, 37 y 38 muestran cómo el algoritmo modificado en Python, toma a la imagen original y la reescalía a un tamaño estándar para su identificación. Dicho tamaño es de  $116 \times 116$ . Este cambio de tamaño se hace, ya que, luego de realizadas las pruebas, se obtuvo que en promedio un marcador de  $3 \times 3$  cm tiene este tamaño, por lo que, debido a la necesidad de partir el marcador en los pequeños cuadros que lo conforman, y ya que la división se hace manualmente ubicando los cuadros en la imagen, resulta más simple llevarlo a este tamaño estándar para que se aplique a un único caso sin importar el tamaño del marcador.

Cabe mencionar que, la identificación en códigos menores a tamaño de  $3 \times 3$  cm, se hace complicada debido a que al reescalarlo se pierde definición de la imagen. Aunque es posible que en condiciones de luz adecuada (luz directa sobre la mesa y los marcadores, de preferencia una luz color blanca) las imágenes de menor tamaño pueden ser mejor identificadas. Aunque estas mismas condiciones también ayudan a una mejor identificación de los marcadores en tamaños mayores o iguales a  $3 \times 3$  cm.

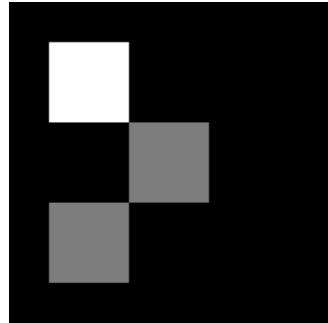


Figura 34: Prueba de generación de código utilizando Python

Como se puede observar en la figura 35, el marcador original tiene un tamaño de  $1 \times 1$  cm. Como se ha mencionado, el procedimiento era reescalarlo al tamaño ya definido. A pesar de lograr el objetivo de rotar el pivote y hacer el escalamiento de las dimensiones, las figuras 35c y 35b se ven distorsionadas y esto representa un problema al momento de intentar identificar los códigos ya que los valores que se desean obtener de estos pueden variar significativamente. La figura 36 muestra un marcador de tamaño  $2 \times 2$  cm. Como se puede observar, luego de hacer el escalamiento, la imagen ya no se observa tan distorsionada en comparación a la figura 35c o 35b. Aún así, si las condiciones de luz no son adecuadas puede que la identificación del marcador no sea correcta.

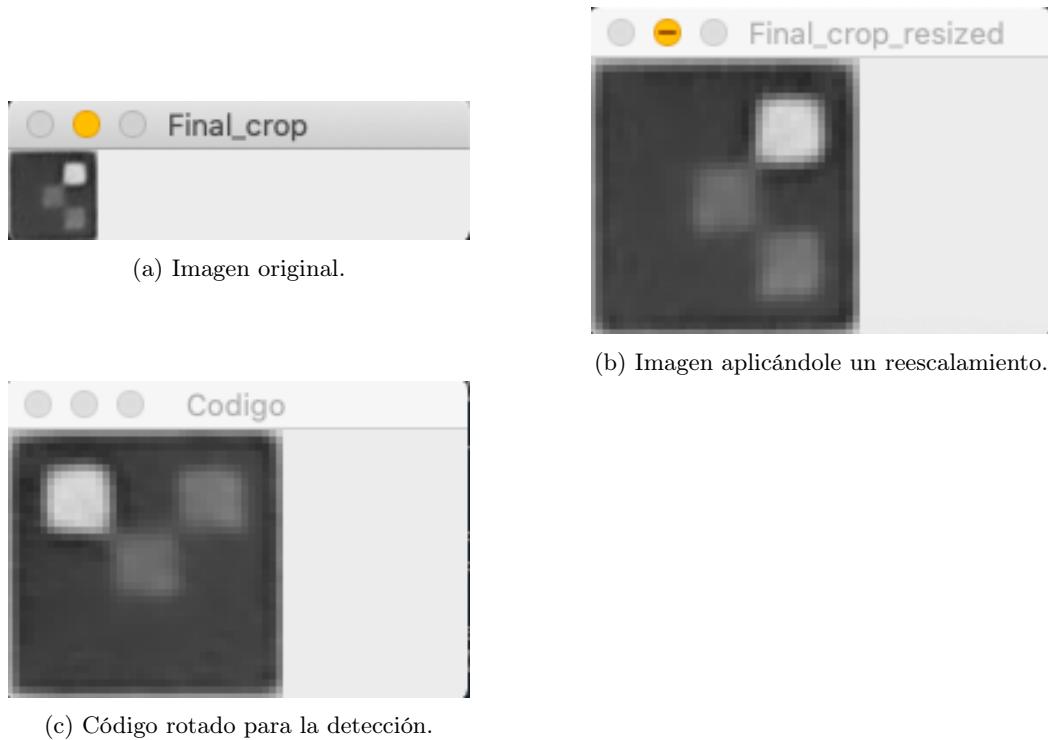


Figura 35: Detección y reescalamiento de código generado para tamaño de  $1 \times 1$  cm.

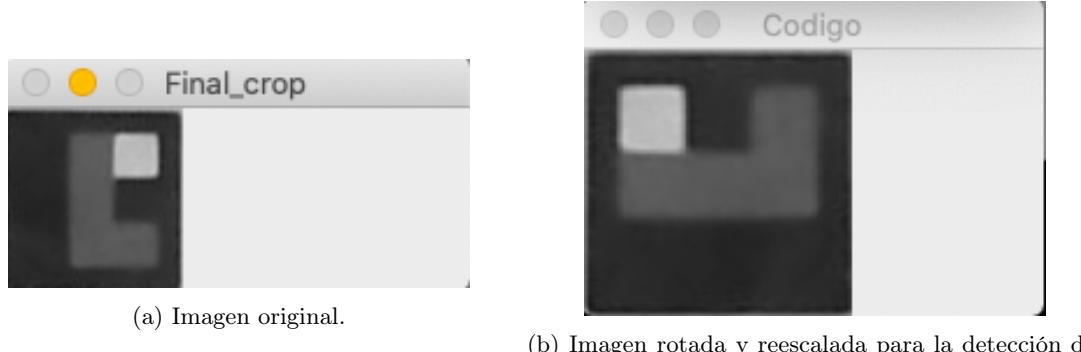
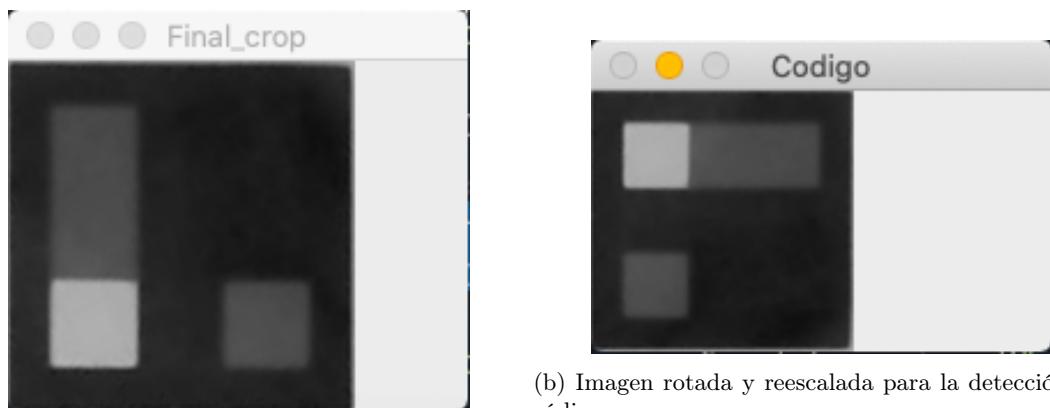


Figura 36: Detección y reescalamiento de código generado para tamaño de  $2 \times 2$  cm.

También se muestran las figuras con tamaños mayores a  $2 \times 2$  centímetros. Como se observa en las figuras 37 y 38, cuyos tamaños originales eran de  $4 \times 4$  y  $8 \times 8$  centímetros respectivamente. A pesar de que son imágenes mucho más grandes, el algoritmo presenta buenos resultados en cuanto a la definición de la imagen al aplicarle el redimensionamiento. Esto indica que es mucho más práctico pasar de dimensiones más grandes hacia dimensiones pequeñas y no tanto viceversa.

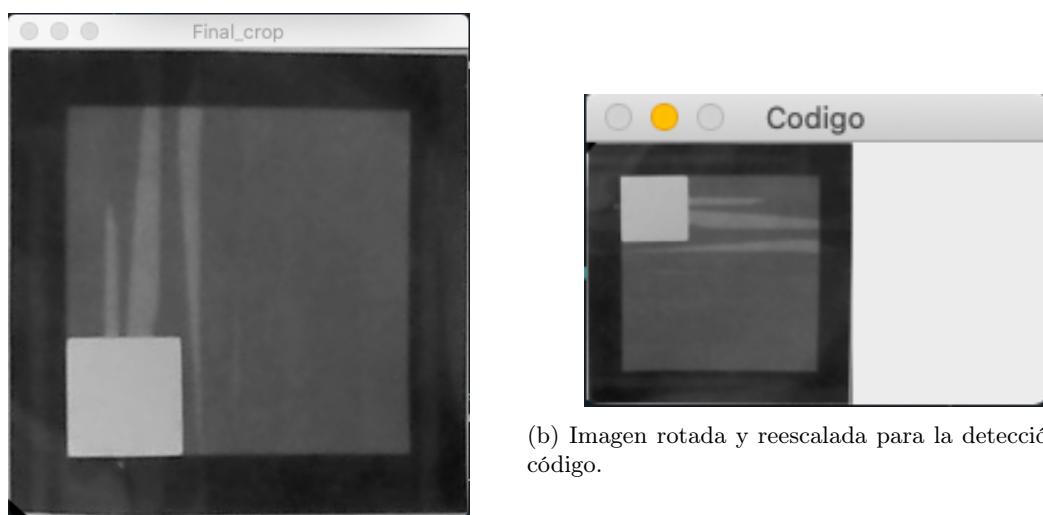
La ventaja de pasar a tamaños más pequeños es, como se ha mencionado, que el algoritmo ubica los diferentes cuadros que conforman los marcadores para obtener el ID que representa. Al ser esto una ubicación ya definida, es decir, dentro del código se define las posiciones que debe buscar, resulta más fácil pasar de cualquier tamaño (más grande o más pequeño) hacia el tamaño de  $3 \times 3$  cm y una vez en este tamaño, realizar todo el procedimiento de identificación.



(a) Imagen original.

(b) Imagen rotada y reescalada para la detección del código.

Figura 37: Detección y reescalamiento de código generado para tamaño de  $4 \times 4$  cm.



(a) Imagen original.

(b) Imagen rotada y reescalada para la detección del código.

Figura 38: Detección y reescalamiento de código generado para tamaño de  $8 \times 8$  cm.

# CAPÍTULO 14

---

## Comparación entre C++ y Python

---

### 14.1. Pruebas de detección de pose

Las primeras pruebas de comparación entre ambos algoritmos fue la de validar la posición detectada para un robot en la mesa. Para esto, se utilizó un papel milimetrado (con divisiones de hasta 1 milímetro) para saber cual era la posición exacta físicamente en la mesa (Posición Real). Las dimensiones de dicha mesa eran de 24x16 cm, como se muestra en la figura 39 (imagen ya calibrada por el algoritmo). El algoritmo se corrió 30 veces en ambos lenguajes y como es posible observar en el cuadro 1, la posición es igual en ambos casos (tanto Python como C++) de manera similar con las posiciones físicas en la mesa.

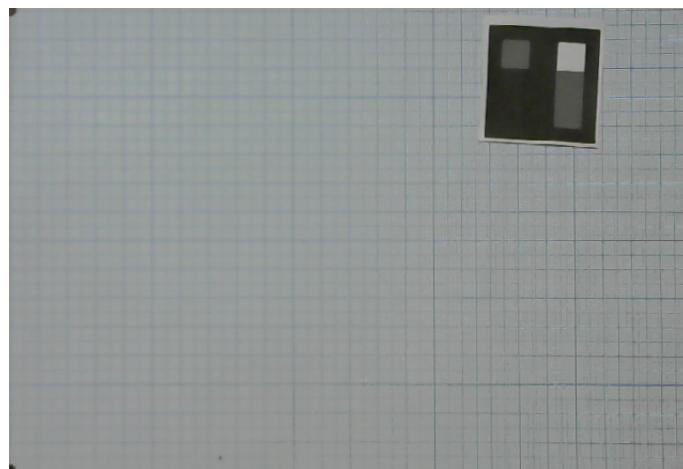


Figura 39: Mesa de pruebas para la obtención de la posición real de los objetos

Python			C++			Posición Real	
X (cm)	Y (cm)	$\theta$ (grad.)	X (cm)	Y (cm)	$\theta$ (grad.)	X (cm)	Y (cm)
9	7	85	9	7	85	9.5	7.5
5	3	90	5	3	91	5	3
19	13	90	19	13	90	20	13
11	4	91	11	4	90	12	4.7
14	8	147	14	8	147	14.5	8
19	13	25	19	13	25	19.5	13.5
3	13	-121	3	13	-121	3.5	13.5
10	8	-68	10	8	-68	10.5	8
19	7	114	19	7	114	20	7.5
9	7	177	9	7	176	9	7.5

Cuadro 1: Comparación entre la detección de pose del algoritmo en Python y C++ y la posición real, con aproximación a enteros

Como se mencionó, el cuadro 1 muestra los resultados del algoritmo luego de aproximar las medidas a números enteros, sin embargo, para tener una mejor visualización, se realizaron las mismas pruebas pero dejando valores decimales. Estos resultados se detallan en el cuadro 2. El objetivo de estas pruebas es tener una referencia de que tan preciso resulta el algoritmo con respecto a las medidas reales en la mesa.

Python			C++			Posición Real	
X (cm)	Y (cm)	$\theta$ (grad.)	X (cm)	Y (cm)	$\theta$ (grad.)	X (cm)	Y (cm)
9.93	11.7	14	10.03	11.6	14	10.5	10.5
19.44	3.0	90	19.62	2.94	90	19.7	3
5.28	3.48	32	5.2	3.51	32	5.5	3.5
9.94	12.79	-79	10.05	12.63	-79	10.3	12.5
3.43	7.76	-15	3.59	7.9	-15	3.5	8
10.05	7.91	84	10.12	7.84	84	10.1	7.8
9.53	3.33	-57	9.47	3.3	-59	10	3
15.41	9.33	-149	15.59	9.21	-149	16	9.5
19.62	13.2	-144	19.85	12.97	-144	20.2	13
19.47	7.8	127	19.71	7.73	127	19.5	7.6

Cuadro 2: Comparación entre la detección de pose del algoritmo en Python-C++ y la posición real sin aproximación a enteros

Finalmente se hizo un análisis estadístico de los datos obtenidos en los cuadros 1 y 2. Cabe mencionar que estos datos fueron calculados utilizando el promedio de la diferencia de los valores absolutos en ambos ejes (X e Y) para tener una perspectiva global de la posición.

En los cuadros 3 y figura 40, se observan las frecuencias de la diferencia (en centímetros) entre el algoritmo de Python y C++. Como se puede observar, la frecuencia cuando se realiza una aproximación a números enteros es prácticamente 0 centímetros (cuadro 3), es decir que no hay diferencia entre la implementación en Python con respecto a la de C++.

Ahora, si se dejan los datos con cifras decimales, al momento de obtener la posición en **Python** con respecto a la de **C++**, las diferencias que más se repiten son 0.06 centímetros, 0.09 centímetros y 0.16 centímetros (con 2 cada uno) mostrados en la figura 40. Además de esto, basado en las pruebas realizadas, se puede decir que la diferencia entre los algoritmos oscila entre 0.05 centímetros y 0.23 centímetros, por arriba o por debajo de esos números la frecuencia fue 0, es decir, que no había ningún registro que correspondiera a esos números.

Diferencia (en cm)	Frecuencia
0	15
1	0
2	0
3	0

Cuadro 3: Frecuencia para la toma de pose con aproximación a entero (eje x y eje y)

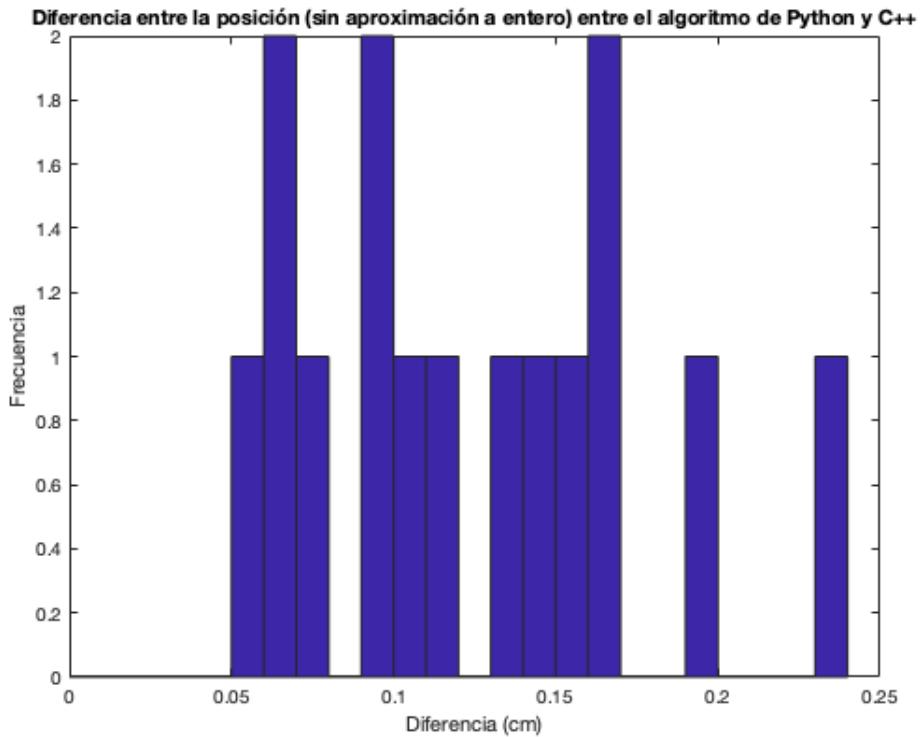


Figura 40: Frecuencia para la diferencia de la posición entre Python y C++ sin aproximación a entero

Finalmente, con respecto a la posición real, la figura 41 muestra que la diferencia que más se repite es aproximadamente 0.5 centímetros, que fue calculada con la posición obtenida en **Python**. La figura 42 muestra la diferencia entre la posición real y la posición del algoritmo de **C++**. Como se observa, los datos de **C++** están más cercanos entre ellos y por debajo de los 0.6 centímetros, a diferencia de los datos de **Python** que están un poco más dispersos, aunque de igual forma se concentran más por debajo de los 0.6 cm.

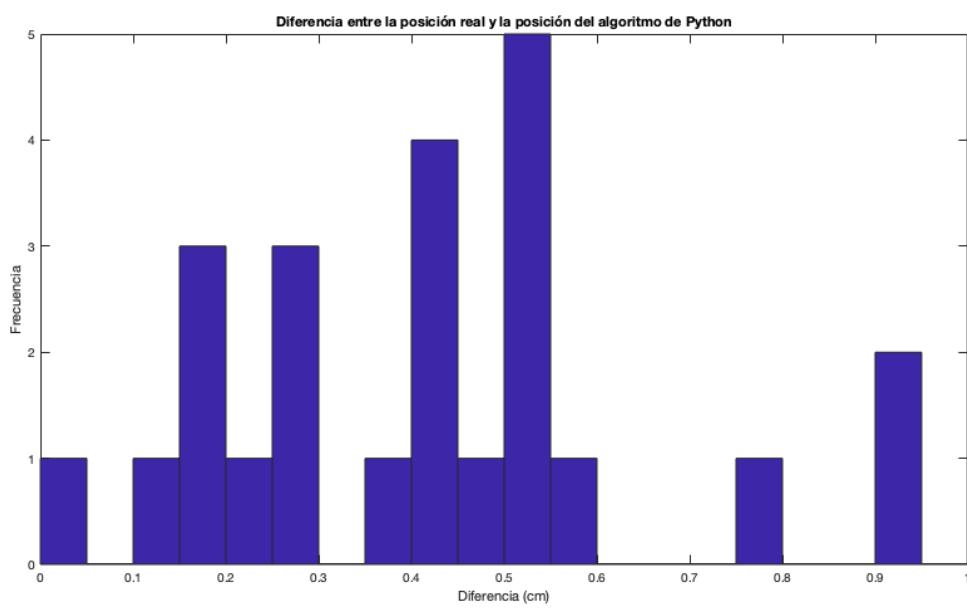


Figura 41: Frecuencia para la toma de pose en Python con respecto a la posición real

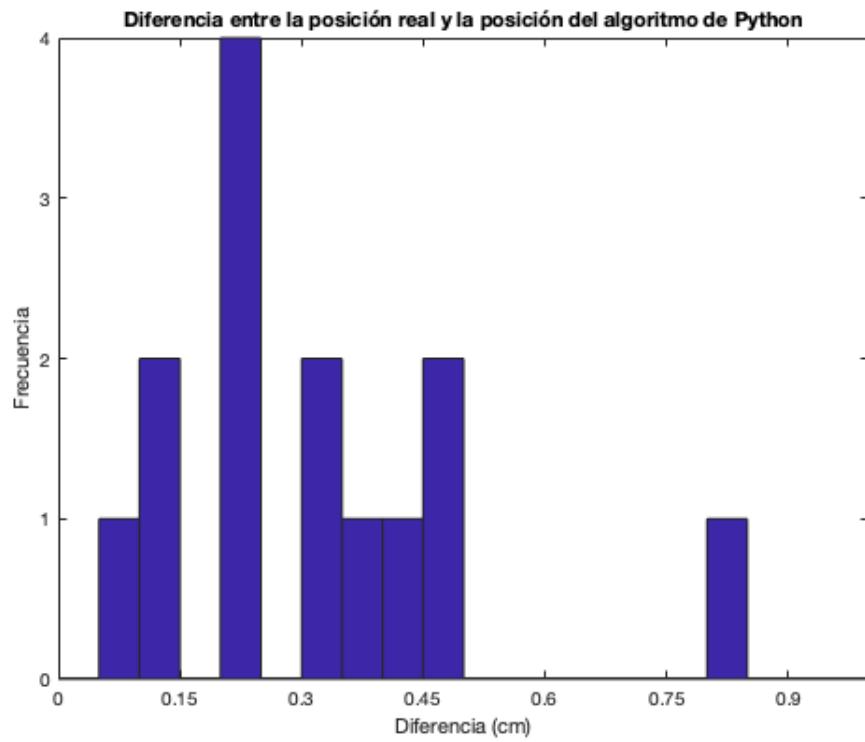


Figura 42: Frecuencia para la toma de pose en C++ con respecto a la posición real

Además de calcular la frecuencia, se calculó la media y la moda para cada caso mostrado en las figuras 40, 41 y 42 . Los resultados se muestran en los cuadros 4, 5, 6 y 7. Como se puede observar, al momento de realizar una aproximación a enteros, la diferencia entre los resultados obtenidos con **Python** y con **C++** es de 0 centímetros, mientras que para el ángulo hay una media de 0.467 grados. En el cuadro 5 se puede observar que la media ya es distinta, ya que aquí se dejaron los datos con cifras decimales, y para este caso es de 0.1157 centímetros con una moda de 0.07 respectivamente. Para el ángulo se obtiene que la media es de 0.2 grados, siendo la moda para este caso de 0 grados como se muestra.

<b>Parámetro</b>	<b>Media</b>	<b>Moda</b>	<b>Desviación Estándar</b>
Posición (cm)	0	0	0
Ángulo (grados)	0.467	0	0.6399

Cuadro 4: Media y moda para la posición con aproximación a enteros

<b>Parámetro</b>	<b>Media</b>	<b>Moda</b>	<b>Desviación Estándar</b>
Posición (cm)	0.1157	0.085	0.0526
Ángulo (grados)	0.2	0	0.5606

Cuadro 5: Media y moda para la posición sin aproximación a enteros

Ahora, con respecto a la posición real, el cuadro 6 muestra los resultados comparativos de **Python**. Para este, se tiene que la media es de 0.3758 centímetros y una moda de 0.5 centímetros. A diferencia de las posiciones calculadas con el algoritmo implementado en **C++**. Los resultados del cuadro 7 indican que la media de la diferencia es de 0.27 centímetros que corresponde a casi un 50 % menos que la diferencia del algoritmo de **Python**. Estos resultados nos indican que, entre algoritmos, la diferencia de las posiciones es baja, pero con respecto a la posición real, sí hay una diferencia un poco más notable. En cualquiera de los casos, el que se tenga una diferencia menor a 0.5 centímetros es un buen indicativo de la operación del algoritmo (tomando en cuenta las dimensiones de la mesa en donde se realizaron las pruebas).

<b>Parámetro</b>	<b>Media</b>	<b>Moda</b>	<b>Desviación Estándar</b>
Posición (cm)	0.3758	0.5	0.2314

Cuadro 6: Media y moda de la diferencia con respecto a la posición real del algoritmo en **Python**

<b>Parámetro</b>	<b>Media</b>	<b>Moda</b>	<b>Desviación Estándar</b>
Posición (cm)	0.27	0.19	0.1968

Cuadro 7: Media y moda de la diferencia con respecto a la posición real del algoritmo en **C++**

## 14.2. Comparación de rendimiento y tiempos de ejecución

El cuadro 8 muestra los tiempos de ejecución entre algoritmos con su implementación multi-hilos. Como se ha expuesto en secciones anteriores, los multi-hilos ayudan a realizar tareas en forma paralela, lo que puede impactar en el tiempo de ejecución.

El algoritmo de **Python** fue implementado con 2 hilos de procesamiento, 1 de actualizar los datos de los robots y el hilo principal. El algoritmo de **C++** tiene únicamente 2 hilos, el principal y un 1 hilo de procesamiento. Esta diferencia recae en poder aportar modularidad al código de **Python**, ya que separando el procesamiento en 2, se hacen esas partes independientes entre sí, por lo que una parte se dedica únicamente a la extracción de información de la foto y la otra a procesar la información. A diferencia de **C++** donde están en uno solo haciendo que el procesamiento y la extracción se ejecuten en un mismo proceso y no en dos independientes.

Aún así, como se observa en el cuadro, el tiempo de ejecución en **C++** es mejor que el tiempo en **Python**. Esto puede deberse a la existencia de 2 hilos adicionales. Aunque como se mencionó, un hilo es para actualización de datos por lo que podría ser eliminado. Sin embargo, el cuadro 9 muestra los tiempos del algoritmo en **Python** con 2 hilos (dejando el código muy similar a la implementación en **C++**). Como se observa, existe una mejora en comparación con su versión teniendo 4 hilos, y por ende, una mejora considerable en cuanto a los tiempos del algoritmo en **C++**.

Estos tiempos se obtuvieron con modo a captura única. Esto quiere decir que el usuario determina cuando realizar la captura de la mesa. Es una opción agregar un hilo de captura continua que ejemplifica de mejor manera el uso de los hilos en este algoritmo.

Lenguaje	Media	Moda	Desv. Estándar	T. Máximo(s)	Repeticiones
Python	0.2519 s	0.29 s	0.031	0.31	45
C++	0.1083 s	0.1073 s	0.00597	0.14	45

Cuadro 8: Medidas de tiempo para los algoritmos usando 4 hilos para Python y 2 hilos para C++

Lenguaje	Media	Moda	Desv. Estándar	T. Máximo(s)	Repeticiones
Python	0.06358 s	0.047 s	0.02659	0.119	45

Cuadro 9: Medidas de tiempo para el algoritmo en Python usando 2 hilos

También se realizó un calculo del tiempo de ejecución sin hilos. El cuadro 10 muestra dichos resultados. Como se puede observar, el tiempo de ejecución en **Python** sin hilos es 1.1 veces más rápido que la ejecución con 2 hilos y 4.5 veces más rápido que su versión con 4 hilos.

Lenguaje	Media	Moda	Desv. Estándar	T. Máximo(s)	Repeticiones
Python	0.05513 s	0.085 s	0.02808	0.100	45

Cuadro 10: Medidas de tiempo para el algoritmo en Python sin hilos

Es importante recalcar que, a pesar de que los tiempos no presentan una mejora con más hilos, el uso de estos ayuda mucho a la versatilidad y modularidad del código. Esto es debido a que permite dividirlo en partes más pequeñas y que cada una de ellas realice una tarea más específica. Por tanto, ayuda mucho en la búsqueda de errores ya que es más fácil identificar donde puede estar una posible falla y solucionarla, lo cual es una gran ventaja en códigos que son grandes.

### 14.3. Interfaz Gráfica de Usuario

Para facilitar al usuario el uso de esta herramienta y los algoritmos que se han discutido a lo largo de este trabajo, se desarrolló una interfaz gráfica. Esta interfaz unifica los algoritmos propuestos en una sola para darle un fácil acceso al usuario, como se muestra en la figura 43.

Este es un primer prototipo de interfaz. Consta de unos botones que permiten calibrar la cámara, o volver a calibrar en caso de ser necesario. Además, se puede ingresar el valor del ID que se desee generar, el cual será mostrado al usuario y guardado automáticamente en la computadora. Finalmente, consta de los botones para la detección de pose del robot, así como un visualizador de las imágenes que se vayan capturando (por ejemplo la imagen ya calibrada o la imagen de la mesa con los robots en ella).

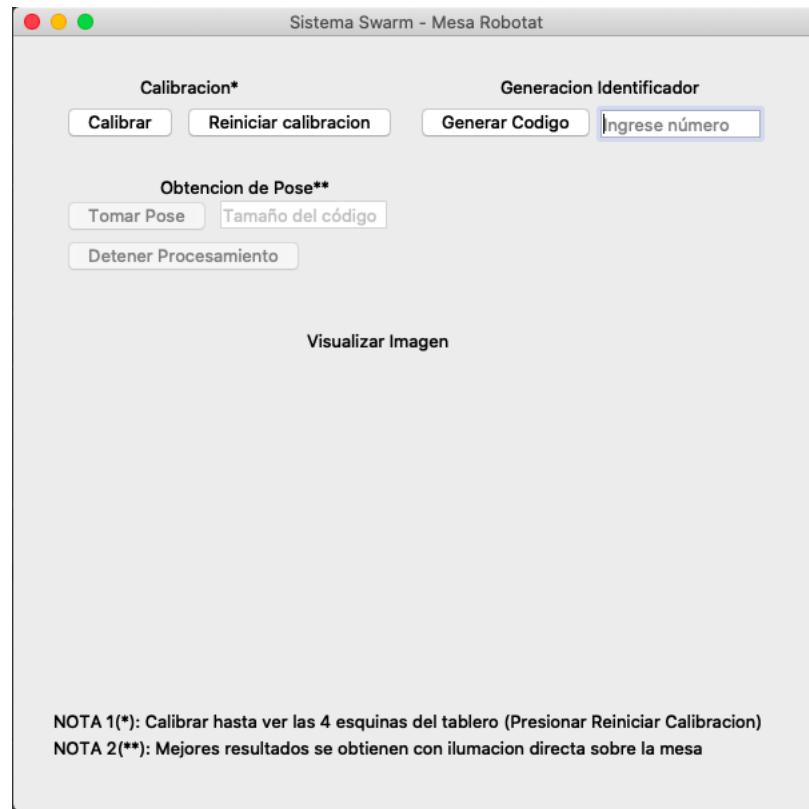


Figura 43: Primer prototipo de interfaz de usuario.

# CAPÍTULO 15

---

## Conclusiones

---

1. Trabajar con una resolución de  $920 \times 760$  permite identificar códigos mínimo de hasta  $3 \times 3$  cm y máximo hasta  $10 \times 10$  cm, de lo contrario, la detección podría ser errónea.
2. Se encontró que hay una diferencia aproximada de 0.5 centímetros entre el algoritmo de **Python** y las medidas con físicas reales y de hasta 0.19 centímetros con la implementación en **C++**. Aún así, la diferencia es considerablemente baja tomando en cuenta las dimensiones de la mesa donde se realizaron las pruebas.
3. Se encontró que si las medidas de la posición son trabajadas en números enteros, estadísticamente no hay diferencia entre la versión de **C++** y la versión de **Python**, pero si no se aproxima a números enteros, la diferencia entre los algoritmos es en promedio 1 milímetro. En ambos casos se denota exactitud y precisión entre los algoritmos.
4. El tiempo de ejecución en la implementación con mult-hilos, es mejor en el lenguaje **C++** que en el lenguaje **Python**. Esto puede deberse a la cantidad de hilos implementados en **Python**.

# CAPÍTULO 16

---

## Recomendaciones

---

1. Existe un editor de interfaces gráficas para el lenguaje C++ llamado Qt. Este tiene su propia versión de hilos, la cual se puede explorar para realizar una comparación entre los hilos que se usaron en los algoritmos.
2. Es posible identificar correctamente los marcadores en condiciones de adecuadas de luz, pero es recomendable siempre tener buena iluminación y una luz directa sobre la mesa para tener alto contraste y obtener mejores resultados.
3. Para tener una correcta calibración en Python, se requiere que las esquinas estén correctamente identificadas con formas lo más circulares posible. Además de esto, evitar que algún otro objeto con la misma forma este por fuera del área de interés ya que puede ser tomado erróneamente como una esquina.
4. Es posible utilizar verdadera parallelización en los multi-hilos. Esto se puede lograr utilizando varios *cores* de una computadora. Para futuras fases se puede explorar esta opción y ver como mejora el rendimientos de estos programas.
5. Es posible encontrar una mejor forma en como el algoritmo de Python identifica el ID. En este trabajo se plantea recortar la imagen en cada cuadro que conforma el marcador. Sin embargo, se puede explorar otras opciones que presenten una mayor eficiencia.

## CAPÍTULO 17

---

### Bibliografía

---

- [1] L. A. Canalís, “Swarm Intelligence in Computer Vision: an Application to Object Tracking”, Univesidad de las Palmas de Gran Canaria, mar. de 2010.
- [2] J. A. Lizarazo Zambrano y M. A. Ramos Velandia, “Visión artificial y comunicación en robots cooperativos omnidireccionales”, Universidad Central, Facultad de Ingeniería y Ciencias Básicas, ene. de 2016.
- [3] H. Inoue y T. Nakatani, “Performance of multi-process and multi-thread processing on multi-core SMT processors”, págs. 1-10, 2010.
- [4] E. W. Felten y D. J. McNamee, *Improving the performance of message-passing applications by multithreading*. Citeseer, 1992.
- [5] A. J. Rodas Hernández, “Desarrollo e implementación de algoritmo de visión por computador en una mesa de pruebas para la experimentación con micro-robots móviles en robótica de enjambre”, Departamento de ingeniería electrónica, mecatrónica y biomédica, Universidad del Valle de Guatemala, ene. de 2019.
- [6] R. L. Briega, *Visión por computadora*, <https://iaarbook.github.io/vision-por-computadora/>, visitado el 10/05/2020.
- [7] Universidad Oberta de Catalunya, *La visión por computador: Una disciplina en auge*, <http://informatica.blogs.uoc.edu/2012/04/19/la-vision-por-computador-una-disciplina-en-auge/>, visitado el 14/09/2020.
- [8] J. C. Hernández, J. Rodríguez y M. F. Santiago, *VISIÓN POR COMPUTADORA*, <http://graficacionporcomputadora.blogspot.com/2013/05/52-vision-por-computadora.html>, visitado el 14/09/2020.
- [9] OpenCV, *About*, <https://opencv.org/about/>, visitado el 05/04/2020.
- [10] D. A. Pizarro, P. Campos y C. L. Tozzi, “COMPARACIÓN DE TÉCNICAS DE CALIBRACIÓN DE CÁMARAS DIGITALES”, *Universidad Tarapacá*, vol. 13, n.º 1, págs. 58-67, 2005. DOI: <http://dx.doi.org/10.4067/S0718-13372005000100007>.

- [11] C. Ricolfe Viala y A. J. Sánchez Salmerón, “PROCEDIMIENTO COMPLETO PARA EL CALIBRADO DE CÁMARAS UTILIZANDO UNA PLANTILLA PLANA”, *Revista Iberoamericana de Automática e Información Industrial*, vol. 5, n.º 1, págs. 93-101, 2008. DOI: [http://dx.doi.org/10.1016/S1697-7912\(08\)70126-2](http://dx.doi.org/10.1016/S1697-7912(08)70126-2).
- [12] F. Lanzaro y M. Fuentes, *Calibración y Posicionamiento 3D*, <https://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2014/candombe/calibracion.html>, visitado el 14/09/2020.
- [13] L. Cabrera Quirós, R. Campos Gómez y J. Castro Godínez, “Pasos críticos en la estimación de pose en cámara: una evaluación usando la biblioteca LTI-LIB2”, págs. 60-69, sep. de 2013.
- [14] E. Coronado, *Patrones de calibración para OpenCV*, <https://mecatronicauaslp.wordpress.com/2014/03/08/patrones-de-calibracion-para-opencv/>, visitado el 14/09/2020.
- [15] R. H. Carver y K.-C. Tai, *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs*. John Wiley & Sons, 2005.
- [16] P. Yong y E. T. W. Ho, “Streaming brain and physiological signal acquisition system for IoT neuroscience application”, págs. 752-757, dic. de 2016. DOI: 10.1109/IECBES.2016.7843551.
- [17] P. Chawla, *OOP with C++*, <http://www.ddegjust.ac.in/studymaterial/mca-3/ms-17.pdf>, visitado el 05/06/2020.
- [18] Oracle, *Lesson: Object-Oriented Programming Concepts*, <https://docs.oracle.com/javase/tutorial/java/concepts/index.html>, visitado el 05/06/2020.
- [19] E. Doherty, *What is Object Oriented Programming? OOP Explained in Depth*, [https://www.educative.io/blog/object-oriented-programming?aid=5082902844932096&utm\\_source=google&utm\\_medium=cpc&utm\\_campaign=blog-dynamic&gclid=Cj0KCQjwoPL2BRDxARIAsAEMm9y\\_jZ7Mu3oH1wRJc5uHdWIeMdhGHbLeR22kkNxokV1-YZS-isYFjZK1aArzfEALw\\_wcB](https://www.educative.io/blog/object-oriented-programming?aid=5082902844932096&utm_source=google&utm_medium=cpc&utm_campaign=blog-dynamic&gclid=Cj0KCQjwoPL2BRDxARIAsAEMm9y_jZ7Mu3oH1wRJc5uHdWIeMdhGHbLeR22kkNxokV1-YZS-isYFjZK1aArzfEALw_wcB), visitado el 05/06/2020.
- [20] J. M. Alarcón, *Los conceptos fundamentales sobre Programación Orientada Objetos explicados de manera simple*, <https://www.campusmvp.es/recursos/post/los-conceptos-fundamentales-sobre-programacion-orientada-objetos-explicados-de-manera-simple.aspx>, visitado el 19/08/2020.

## CAPÍTULO 18

---

Anexos

---

