

---

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

---

José Pablo Guerra Jordán



UNIVERSIDAD DEL VALLE DE GUATEMALA  
Facultad de Ingeniería



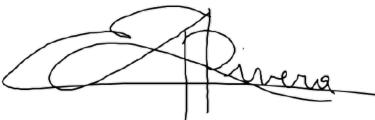
**Algoritmos de Visión por Computadora para el  
Reconocimiento de la Pose de Agentes Empleando  
Programación Orientada a Objetos y Multihilos**

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

Guatemala,

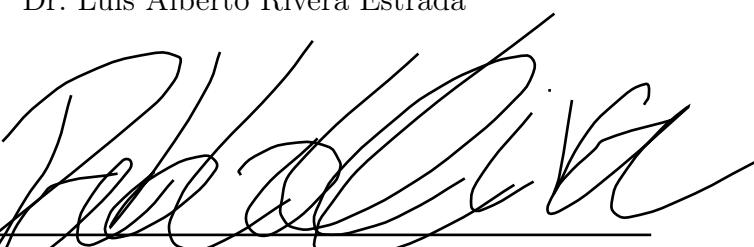
2021

Vo.Bo.:

(f)   
Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f)   
Dr. Luis Alberto Rivera Estrada

(f)   
MSc. Pablo Roberto Oliva Fonseca

(f)   
MSc. Carlos Alberto Esquit Hernández

Fecha de aprobación: Guatemala, 11 de Enero de 2021 .

---

## 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, porque 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>	III
<b>Lista de figuras</b>	VII
<b>Lista de cuadros</b>	VIII
<b>Resumen</b>	IX
<b>Abstract</b>	X
<b>1. Introducción</b>	1
<b>2. Antecedentes</b>	3
2.1. Visión por computadora aplicado a robótica de enjambre . . . . .	3
2.2. Multihilos como una opción para mejorar el rendimiento . . . . .	4
2.3. Continuación de la fase I . . . . .	4
<b>3. Justificación</b>	5
<b>4. Objetivos</b>	7
4.1. Objetivo general . . . . .	7
4.2. Objetivos específicos . . . . .	7
<b>5. Alcance</b>	8
<b>6. Marco teórico</b>	9
6.1. Visión por computadora . . . . .	9
6.1.1. OpenCV . . . . .	10
6.1.2. Calibración de cámaras . . . . .	10
6.2. Programación multihilos . . . . .	12
6.3. Programación orientada a objetos . . . . .	13
6.3.1. Objetos . . . . .	14
6.3.2. Clase . . . . .	14
6.3.3. Encapsulación . . . . .	14

<b>6.3.4. Abstracción . . . . .</b>	<b>14</b>
<b>7. Metodología . . . . .</b>	<b>15</b>
<b>7.1. Investigación . . . . .</b>	<b>15</b>
<b>7.2. Implementación . . . . .</b>	<b>15</b>
<b>7.3. Validación . . . . .</b>	<b>16</b>
<b>8. Prototipos de mesa de pruebas . . . . .</b>	<b>17</b>
<b>8.1. Primer prototipo . . . . .</b>	<b>17</b>
<b>8.2. Segundo prototipo . . . . .</b>	<b>19</b>
<b>9. Pruebas preliminares en Python . . . . .</b>	<b>21</b>
<b>9.1. Pruebas de multihilos . . . . .</b>	<b>21</b>
<b>9.2. Pruebas de OpenCV . . . . .</b>	<b>25</b>
<b>10. Pruebas algoritmo y OpenCV en C++ . . . . .</b>	<b>27</b>
<b>10.1. Calibración de cámara en C++ . . . . .</b>	<b>27</b>
<b>10.2. Identificación de marcadores en C++ . . . . .</b>	<b>30</b>
<b>11. Migración de código de C++ a Python . . . . .</b>	<b>35</b>
<b>11.1. Migración del código para la calibración de cámara . . . . .</b>	<b>37</b>
<b>11.2. Migración del código para generación de marcadores e identificación . . . . .</b>	<b>45</b>
<b>11.2.1. Generación del marcador . . . . .</b>	<b>45</b>
<b>11.2.2. Identificación del ID . . . . .</b>	<b>47</b>
<b>12. Pruebas de detección de marcadores con diferentes tamaños en Python . . . . .</b>	<b>51</b>
<b>13. Comparación entre los algoritmos de C++ y Python . . . . .</b>	<b>55</b>
<b>13.1. Pruebas de detección de pose . . . . .</b>	<b>55</b>
<b>13.2. Comparación de rendimiento y tiempos de ejecución . . . . .</b>	<b>62</b>
<b>13.3. Interfaz gráfica de usuario . . . . .</b>	<b>64</b>
<b>14. Pruebas preliminares con Matlab . . . . .</b>	<b>65</b>
<b>14.1. Captura de imágenes usando Matlab . . . . .</b>	<b>65</b>
<b>14.2. Corrección de perspectiva previo a la calibración . . . . .</b>	<b>66</b>
<b>14.3. Generación de los marcadores . . . . .</b>	<b>67</b>
<b>14.4. Detección de contornos en Matlab . . . . .</b>	<b>67</b>
<b>15. Conclusiones . . . . .</b>	<b>72</b>
<b>16. Recomendaciones . . . . .</b>	<b>73</b>
<b>17. Bibliografía . . . . .</b>	<b>74</b>
<b>18. Glosario . . . . .</b>	<b>76</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.	Diagrama de flujo para el programa con multi-hilos. . . . .	22
12.	Captura del primer archivo de texto para el ejemplo multihilos. . . . .	23
13.	Captura del segundo archivo de texto para el ejemplo multihilos. . . . .	23
14.	Captura del texto reconstruido para el ejemplo multihilos. . . . .	24
15.	Captura de la cámara web con OpenCV. . . . .	25
16.	Intento 1 de calibración utilizando C++. . . . .	28
17.	Intento 2 de calibración utilizando C++. . . . .	29
18.	Identificación de marcador utilizando el algoritmo de C++. . . . .	30
19.	Diagrama de flujo para la identificación de un marcador. . . . .	30
20.	Diagrama de flujo para la generación de un marcador [5]. . . . .	31
21.	Diagrama de flujo para la identificación de un marcador. . . . .	32
22.	Imagen original para la identificación de los marcadores. . . . .	33
23.	Primera prueba de identificación de código en la imagen capturada. . . . .	33
24.	Identificación del marcador. . . . .	34
25.	Ejemplo de imagen a calibrar en la migración del algoritmo. . . . .	37
26.	Proceso de calibración de la cámara en C++ [5]. . . . .	38
27.	Proceso de detección de bordes y definición de esquinas en Python. . . . .	39
28.	Ejemplo de contornos identificados por el algoritmo de Python. . . . .	41

29.	Contornos identificados para la calibración en Python. . . . .	42
30.	Ejemplo de imagen calibrada utilizando el algoritmo de Python. . . . .	42
31.	Ejemplo de contornos identificados por el algoritmo de C++. . . . .	43
32.	Ejemplo de imagen calibrada utilizando el algoritmo de C++. . . . .	43
33.	Primera prueba calibración de la cámara con contorno en la mesa utilizando Python. . . . .	44
34.	Segunda prueba calibración de la cámara con objetos en la mesa utilizando Python. . . . .	44
35.	Ejemplo de marcador generado en Python con el ID 189. . . . .	46
36.	Ejemplo de marcador generado en Python con el ID 40. . . . .	46
37.	Marcador generado con el ID 178 . . . . .	46
38.	Diagrama de flujo para el reconocimiento del ID en el algoritmo de Python. . . . .	48
39.	Ejemplo de imagen para identificar el marcador. . . . .	49
40.	Marcador reconocido en C++. . . . .	50
41.	Resultado luego de la identificación del marcador en Python. . . . .	50
42.	Detección y reescalamiento de código generado para tamaño de $1 \times 1$ cm. . . . .	52
43.	Detección y reescalamiento de código generado para tamaño de $2 \times 2$ cm. . . . .	53
44.	Detección y reescalamiento de código generado para tamaño de $4 \times 4$ cm. . . . .	54
45.	Detección y reescalamiento de código generado para tamaño de $8 \times 8$ cm. . . . .	54
46.	Mesa de pruebas para la obtención de la posición real de los objetos . . . . .	55
47.	Marco de referencia para la toma de pose. . . . .	56
48.	Frecuencia para la diferencia de la posición entre Python y C++ sin aproximación a entero . . . . .	59
49.	Frecuencia para la toma de pose en Python con respecto a la posición real . . . . .	60
50.	Frecuencia para la toma de pose en C++ con respecto a la posición real . . . . .	60
51.	Primer prototipo de interfaz de usuario. . . . .	64
52.	Imagen capturada usando Matlab y la cámara web. . . . .	66
53.	Corrección de perspectiva de la imagen usando Matlab. . . . .	66
54.	Ejemplo de marcador generado usando Matlab. . . . .	67
55.	Imagen para pruebas en Matlab. . . . .	68
56.	Primeras pruebas de detección de contornos usando Canny en Matlab. . . . .	69
57.	Prueba de detección de contornos en Matlab. . . . .	69
58.	Prueba de detección de centro del contorno en Matlab. . . . .	70
59.	Ubicación de los contornos utilizando . . . . .	71
60.	Prueba de detección de centro del contorno en Matlab. . . . .	71

---

## Lista de cuadros

---

1.	Posición real para el caso con aproximación a enteros. . . . .	56
2.	Posición detectada por los algoritmos de Python y C++ con aproximación a enteros. . . . .	57
3.	Posición real para el caso sin aproximación a enteros. . . . .	57
4.	Posición detectada por los algoritmos de Python y C++ sin aproximación a enteros. . . . .	58
5.	Frecuencia para la toma de pose con aproximación a entero (eje x y eje y) . . .	58
6.	Media y moda para la posición con aproximación a enteros entre los algoritmos de Python y C++ . . . . .	61
7.	Media y moda para la posición sin aproximación a enteros entre los algoritmos de Python y C++ . . . . .	61
8.	Media y moda de la diferencia con respecto a la posición real del algoritmo en Python (ambos ejes). . . . .	61
9.	Media y moda de la diferencia con respecto a la posición real del algoritmo en C++ (ambos ejes). . . . .	61
10.	Diferencia por eje (X,Y) entre la posición detectada por el algoritmo de Python y la posición real. . . . .	62
11.	Diferencia por eje (X,Y) entre la posición detectada por el algoritmo de C++ y la posición real. . . . .	62
12.	Medidas de tiempo para los algoritmos usando 4 hilos para Python y 2 hilos para C++ . . . . .	63
13.	Medidas de tiempo para el algoritmo en Python usando 2 hilos . . . . .	63
14.	Medidas de tiempo para el algoritmo en Python y C++ sin hilos . . . . .	63

---

## Resumen

---

En este trabajo se implementó un algoritmo de visión por computadora de una manera eficiente utilizando la Programación Orientada a Objetos (POO) y multihilos. 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, obtención de mediciones por ambos algoritmos, entre otras, y así determinar en qué puntos hubo mejoras, o bien, si los algoritmos implementados en los dos lenguajes tenían el mismo rendimiento y presentaban resultados 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 multihilos, 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

---

In this work, a computer vision algorithm was implemented in an efficient way using Object Oriented Programming (OOP) and multithreading programming. This was done by analyzing and improving the algorithm previously developed in C++ language, migrating it to the Python language. The validations were carried out through comparative tests to obtain parameters such as execution times, among others. These parameters helped to determine in which points there were improvements, or if both algorithms had the same performance. In addition, helped to validate that the results of the implementations were similar.

A software tool was developed. The main functionality of the tool is to recognize the position of agents (term commonly used in swarm robotics). The tool was combined with a testbed, which was assembled to test the calibration of the camera. These tests were important to identify possible improvements 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 due to the performance improvement achieved through the multithreading programming. Furthermore, the use of OOP provides modularity and may be useful for future applications.

# 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 de 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 multihilos 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 consistió 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

---

### 2.1. 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 simulaciones 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.

## 2.2. Multihilos como una opción para mejorar el rendimiento

Hiroshi Inoue y Toshio Nakatani proponen una comparación entre un modelo orientado a multihilos 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 multihilos. Los principales resultados de estos experimentos fueron que el uso de multihilos 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.

## 2.3. 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 o videos, 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 que requieran gran poder de computo, debido a la cantidad de procesamiento y análisis que se realiza en dichas imágenes o videos.

Sin embargo, estas tareas pueden representar un costo alto en recursos computacionales. Para esto, la programación multihilos 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 multihilos 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

---

### 4.1. 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.

### 4.2. 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. Como resultado de este trabajo, se adaptó la versión desarrollada en el lenguaje C++ al lenguaje Python para tener una herramienta versátil, logrando tener distintas opciones según sea la aplicación o requerimiento del usuario. Además, el algoritmo permite la identificación de los robots en un espacio físico, como por ejemplo la mesa de pruebas Robotat<sup>[5]</sup>, mediante el procesamiento de imágenes que se realiza de manera paralela mediante el uso de multihilos. Por tanto, además de la identificación de los robots, dentro de las funciones del algoritmo, está el poder calibrar la cámara para enfocarse en el área de interés o trabajo.

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 qué 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 videos, para recolectar información para su posterior análisis. En base a esto, se desarrollaron 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 de imágenes y videos, para que las computadoras puedan entender de mejor manera que es lo que hay en ellos. Es decir, la computadora puede visualizar lo que hay en una imagen como si estuviera observándola, como lo haría un ser humano.

Normalmente, este tipo de procesamiento de datos es utilizado para obtener información del medio o entorno (mapas, carreteras, imágenes de todo tipo, etc.). De esta forma, una vez procesada la información, utilizarla 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.

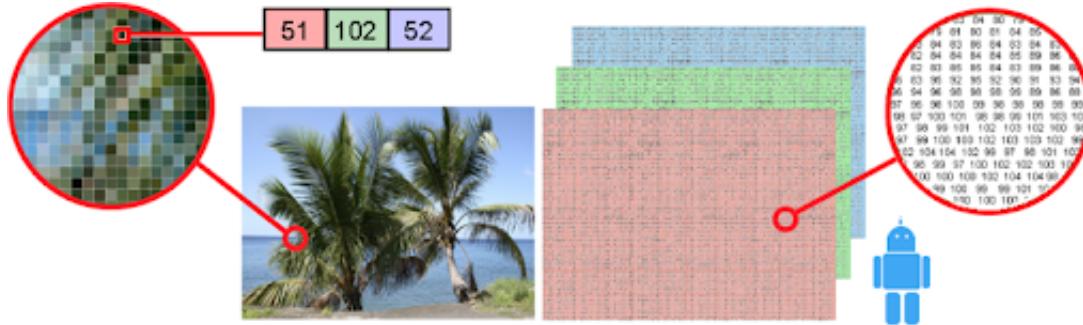


Figura 1: Representación de la información que extrae una computadora a partir de una imagen [7].

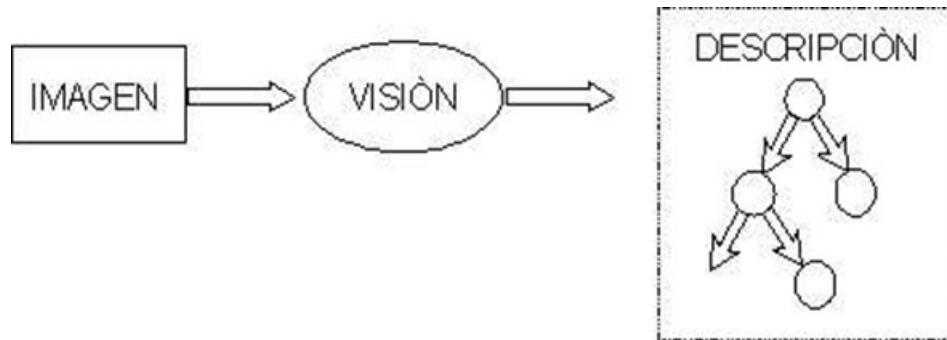


Figura 2: Proceso general para la extracción de información de una imagen para su uso en Visión por computadora [8].

### 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 más 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 de 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 Figura también muestra cómo 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.

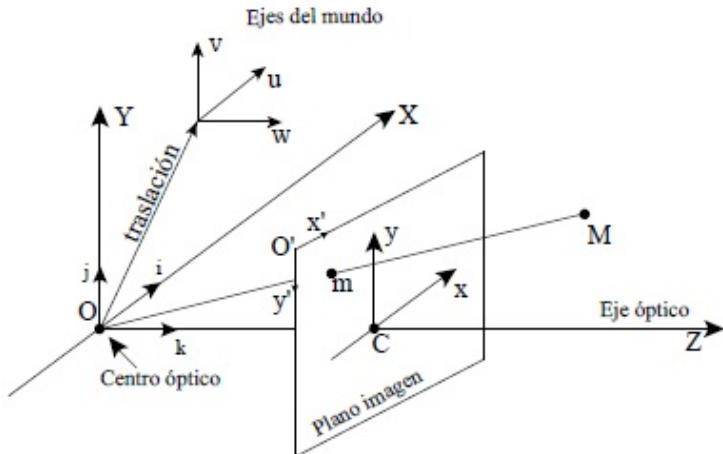


Figura 3: Conversión entre el plano de imagen al mundo real [12].

La Figura 4 es un patrón común en la calibración de cámaras. Esto se debe a que con ayuda de este patrón, se ubican los puntos de interés (las esquinas de los cuadros) y con estos se puede estimar los parámetros de la cámara [13], como por ejemplo posiciones o medidas de la imagen al mundo real, además de obtener otros factores, como la distorsión de imagen, por ejemplo.

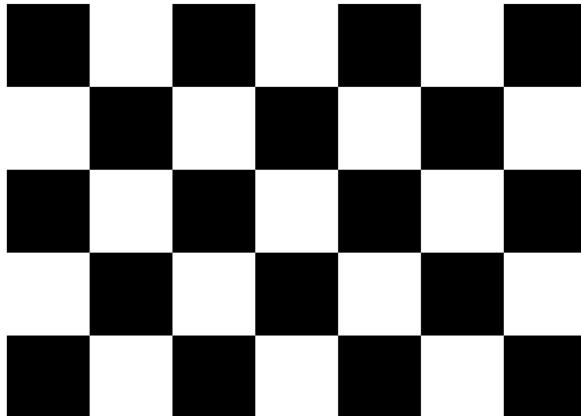


Figura 4: Patrón común para la calibración [14].

## 6.2. Programación multihilos

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 simultánea 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].

La Figura 5 muestra como los hilos se pueden observar en un proceso o programa computacional. Como se ve, hay tres 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).

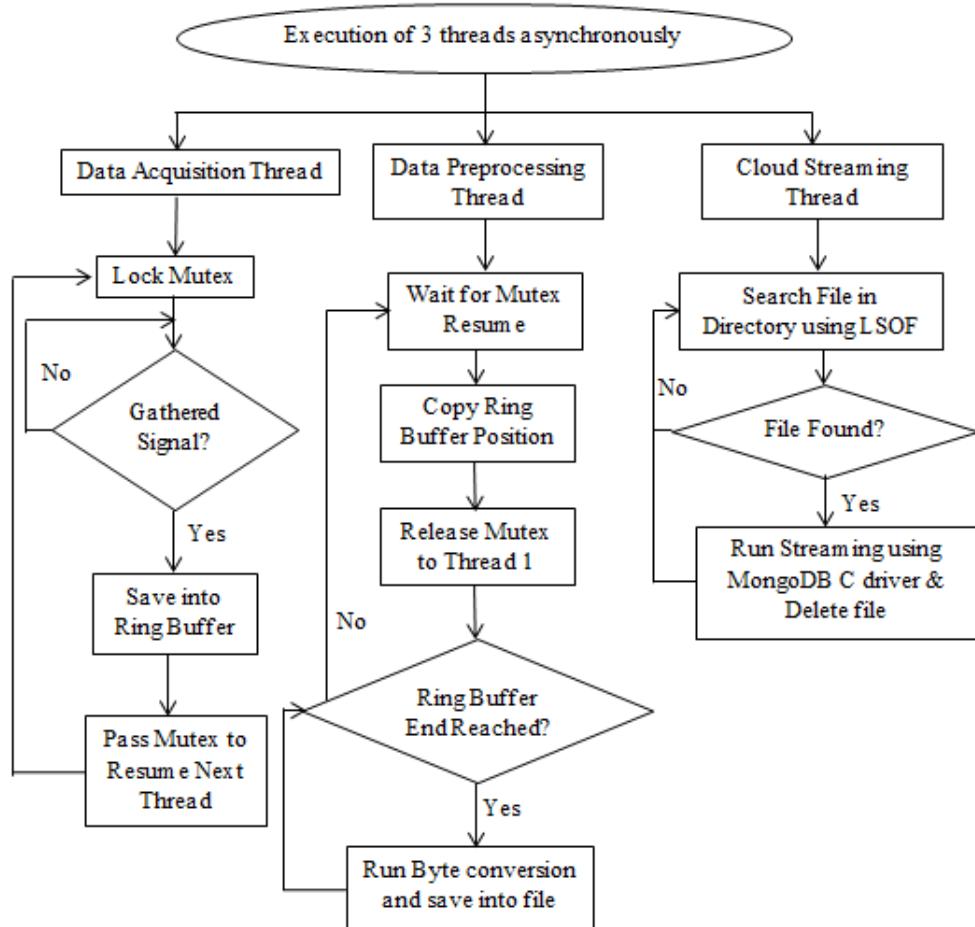


Figura 5: Ejemplificación del uso de hilos en un programa o proceso [16].

### 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

---

### 7.1. 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 multihilos, 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.

### 7.2. Implementación

Una vez investigado lo referente a la programación multihilos, OpenCV, y programación orientada a objetos, se procedió a realizar diferentes implementaciones de códigos para validar el funcionamiento de las diferentes funciones que iban a ser utilizadas. Estos fueron programas de ejemplo de aplicación multihilos, así como el uso de una cámara y OpenCV. Esto se realizó utilizando el lenguaje de Python, que es el lenguaje hacia donde se migró los algoritmos implementados en la fase anterior.

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 de *software* que utiliza los algoritmos de la fase anterior. Una vez realizada 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  $28 \times 14$  cm.

### 7.3. Validación

Finalmente, luego de tener ambas implementaciones, 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 multi-hilos de ambos lenguajes definidos, aplicados al algoritmo de detección y toma de pose. 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. Estas acciones consistieron en ubicar las nuevas esquinas deseadas en la imagen. 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  $24 \times 16$  cm.

# CAPÍTULO 8

---

## Prototipos de mesa de pruebas

---

### 8.1. Primer prototipo

Debido a que no se pudo utilizar la mesa de pruebas Robotat de la Universidad del Valle, se diseñó una mesa similar para realizar las pruebas de validación y comparación. Un primer prototipo 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 muestran las figuras, 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 para tener la vista 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 la Figura 8. Esto ayudo 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 inconveniente 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  $28 \times 14$  cm para simular la mesa y de igual forma se le colocaron puntos en las esquinas para referencia en la calibración (Figura 9). La Figura 10 muestra más de cerca la colocación de la cámara en el tripode para este segundo prototipo.



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

---

Con el objetivo de comprender el uso lenguaje de programación a utilizar para la migración (en este caso Python), se realizaron algunas pruebas para entender su sintaxis y la aplicación de las diferentes funciones y la librería OpenCV. Como primera prueba se realizó un programa utilizando multihilos, y así, entender la sintaxis y el modo de emplearlos en Python. Además de este programa, se hizo otro para utilizar la librería OpenCV, y con esto, comprender su funcionamiento y las aplicaciones que puede tener.

### 9.1. Pruebas de multihilos

El objetivo era desarrollar un programa en lenguaje Python con multihilos, que tomara ordenadamente cada línea de cada archivo y lo ordenara en uno nuevo. El proceso general de este programa se muestra en la Figura 11. Para esto se contaban con dos archivos de texto. En la Figura 12 se observa un primer archivo de texto del himno nacional de Guatemala pero con las líneas impares. Luego, en la Figura 13, 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, como lo muestra la Figura 14. 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.

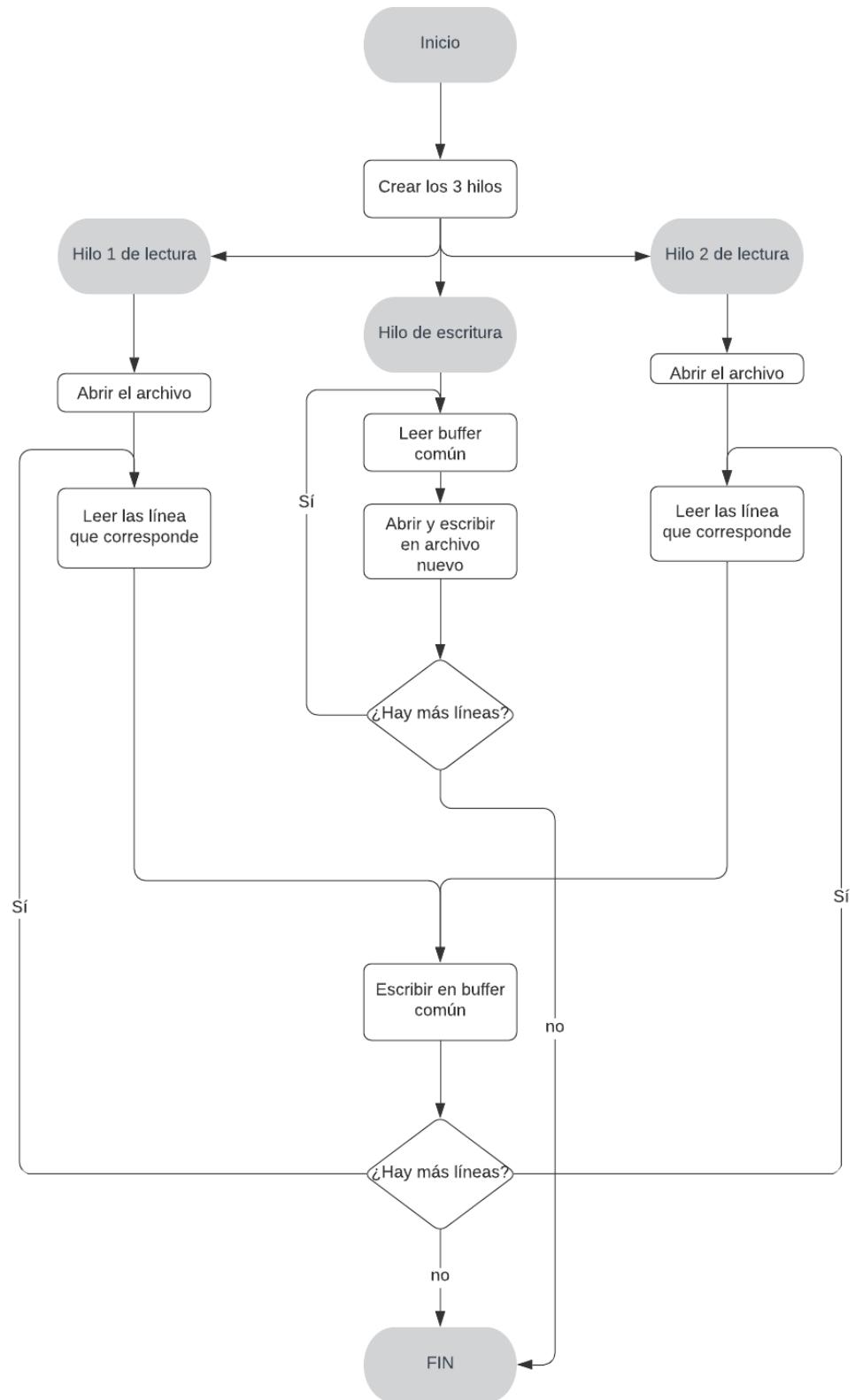


Figura 11: Diagrama de flujo para el programa con multi-hilos.

```
● ● ● 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 12: Captura del primer archivo de texto para el ejemplo multihilos.

```
● ● ● 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 13: Captura del segundo archivo de texto para el ejemplo multihilos.



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 ardimiento,  
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 14: Captura del texto reconstruido para el ejemplo multihilos.

## 9.2. Pruebas de OpenCV

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 desarrollado en la fase anterior de este trabajo. 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 Figura 15.

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. También ilustra 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. El código 9.1 muestra como se genera la Figura 15.



Figura 15: Captura de la cámara web con OpenCV.

Como se puede observar en la línea 26, se obtiene los *frames* o cuadros del vídeo utilizando `cap.read()`. Luego en la línea 37, se les aplica el filtro de blanco y negro mediante la función `cv.cvtColor()` y se muestra ese nuevo cuadro como si el vídeo se estuviera tomando originalmente en blanco y negro. Este código ilustra, como se mencionó, las aplicaciones de OpenCV y el manejo para el procesamiento de imágenes y vídeos.

```

1 """
2 Jose Pablo Guerra
3 Programa de ejemplo para captura de video y procesamiento de imagen usando
4 OpenCV
5 Version final.
6 """
7 """
8 """
9 Anotaciones iniciales:
10 De preferencia utilizar la suite de anaconda, esto permite instalar los
11 paquetes
12 de manera mas adecuada y evitar errores entre versiones o que las librerias
13 esten
14 correctamente linkeadas al compilador.
15 """
16 """
17 """
18
19 import cv2 as cv #importando libreria para opencv
20
21 cap = cv.VideoCapture(0) #VideoCapture(n) n = 0 para otras que no sean la
22 camara principal.
23
24 if not cap.isOpened(): #detectando si la camara esta disponible.
25     print("Cannot open camera")
26     exit()
27 while True:
28
29     ret, frame = cap.read() #lectura y caputra del frame
30
31     #print(ret)
32     if not ret: #debugin del frame, si hay problemas sale del programa
33         #Al inicio siempre puede dar un problema para detectar los
34 frames de video
35
36     print("Can't receive frame (stream end?). Exiting ...")
37     break
38
39     gray = cv.cvtColor(frame, cv.COLOR_BGR2GRAY) #captura de video y
40     procesamiento de imagen a tono de grises.
41     print(gray)
42     print(gray.dtype)
43     cv.imshow('frame', gray) #despliega los cambios hechos en el frame
44     if cv.waitKey(1) == ord('q'): #para salir presione q.
45         break
46
47 #salir correctamente.
48 cap.release()
49 cv.destroyAllWindows()

```

Código 9.1: Código para generar vídeo en blanco y negro.

# CAPÍTULO 10

---

## Pruebas algoritmo y OpenCV en C++

---

En este capítulo se presentan algunas de las pruebas realizadas utilizando el algoritmo de C++, con el objetivo de mostrar su funcionamiento y validarla para las posteriores pruebas comparativas con la migración hacia Python. Estas son unas primeras pruebas que ayudaron a comprender el funcionamiento del algoritmo. Las pruebas van desde la calibración de la cámara, hasta la detección de los **marcadores** colocados en la mesa de pruebas.

### 10.1. Calibración de cámara en C++

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

Sin embargo, en la Figura **[16]** se muestra una calibración fallida de la mesa (los puntos identificados no fueron colocados en la esquina de la imagen). Esto se debe 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 (línea 8), consistía en ubicar el ancho y alto de las figuras que se usarían como posibles nuevas esquinas, y con esto, comparar que estuvieran dentro de un rango de tamaño (como se muestra en la línea 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, lo que hace que este método de calibración pueda no ser adecuado bajo otras condiciones.



(a) Imagen original para ubicar las esquinas.

(b) Intento 1 de calibración, fallido.

Figura 16: 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 anteriormente mencionada, y que se basará únicamente en la ubicación de los puntos en la imagen, como se ve en el código 10.2. Es decir, que ya no se compara si la figura identificada está dentro de un rango en píxeles que lo catalogue como un círculo o una figura circular, sino que únicamente lo identifica y compara su posición para ver si es el más cercano al borde. Por lo tanto, esta nueva condición recorría los contornos identifi-

cados 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.

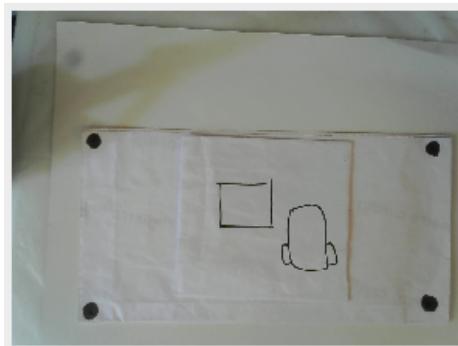
Así, luego de que el ciclo se completara, se tendrían los contornos mejor ubicados en las esquinas que se desearán tener.

```

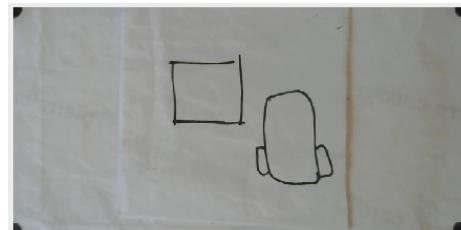
1  for (int i = 0; i < contours.size(); i++)
2  {
3      RotatedRect cod;
4      cod = minAreaRect(contours[i]);
5
6      if (a) {
7          for (int i = 0; i < 4; i++)
8              esquina[i] = cod.center;
9              a = false;
10     }
11     else {
12         for (int i = 0; i < 4; i++)
13     if(distancia2puntos(bordesMAX[i],cod.center)<distancia2puntos(bordesMAX[i],
14     esquina[i]))
15             esquina[i] = cod.center;
16     }
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 mostrados en las figuras 16a y 17a) serán tomados como las esquinas de la mesa y por tanto, la imagen se calibrara correctamente, como se puede observar en la Figura 17b. Además de esto, se puede ver que hay objetos en la Figura 17, 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 17: Intento 2 de calibración utilizando C++.

## 10.2. Identificación de marcadores en C++

Los **marcadores** en este capítulo (y en capítulos posteriores) hacen referencia a una imagen compuesta de una cuadrícula de  $3 \times 3$  cuadros, donde el cuadro blanco sirve como pivote de la imagen, es decir, en caso de que la imagen esté rotada, este cuadro blanco siempre debe estar en la esquina superior izquierda. Los cuadros negros y grises forman el código. Al final, la imagen es una representación binaria del ID. Cada cuadro representa una potencia de 2, hasta 8 bits (255). Los cuadros grises son 1 (o bit encendido) y los cuadros negros son 0 (o bits apagados).

El pivote sirve para poder posicionar la imagen correctamente para identificar el marcador, como se muestra en la Figura 18. 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 dos primeros cuadros de la fila superior (donde está ubicado el cuadro blanco) representan los primeros bits ( $2^0$  y  $2^1$ ), y subsecuentemente cada uno crece de izquierda a derecha hasta llegar a al último bit, ubicado en la esquina inferior derecha. La Figura 19 representa un 170 e ilustra de mejor manera la explicación de los cuadros, ya que para este caso el código binario es 01010101. Además, la Figura 20 ilustra como se genera un marcador utilizando el algoritmo de C++, y el proceso completo para la identificación del marcador se detalla en la Figura 21.

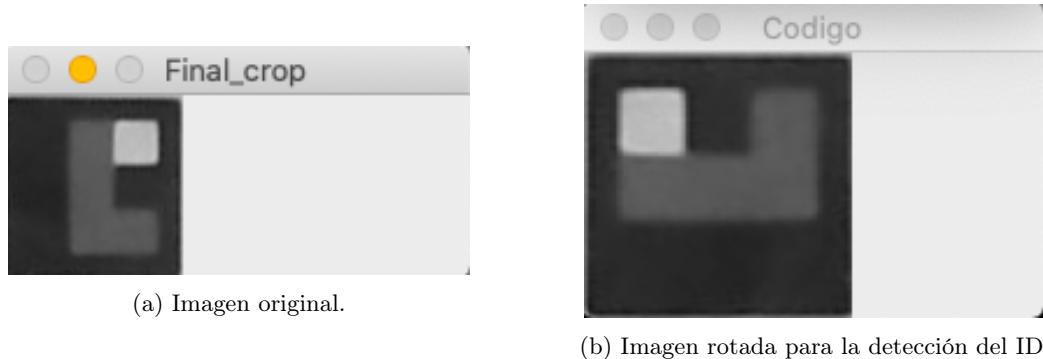


Figura 18: Identificación de marcador utilizando el algoritmo de C++.

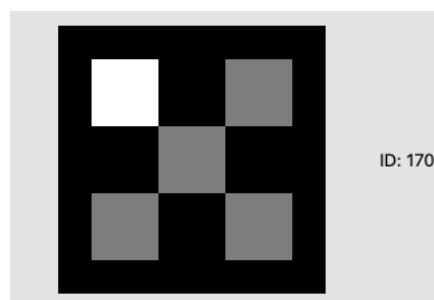


Figura 19: Diagrama de flujo para la identificación de un marcador.

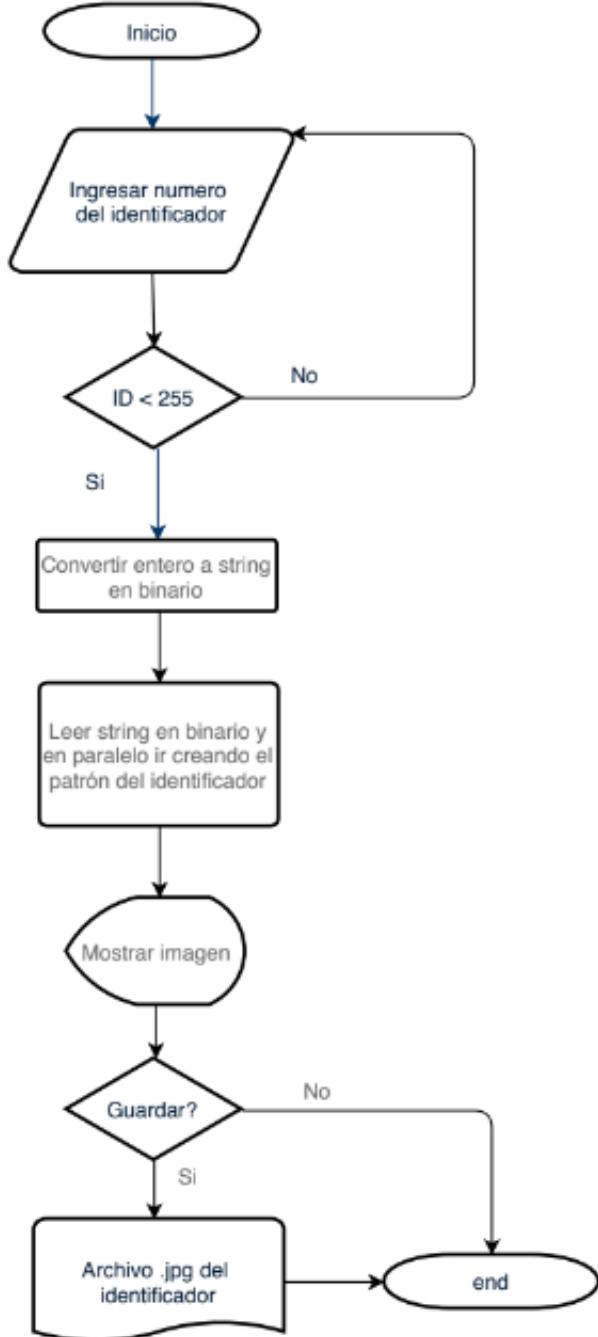


Figura 20: Diagrama de flujo para la generación de un marcador [5].

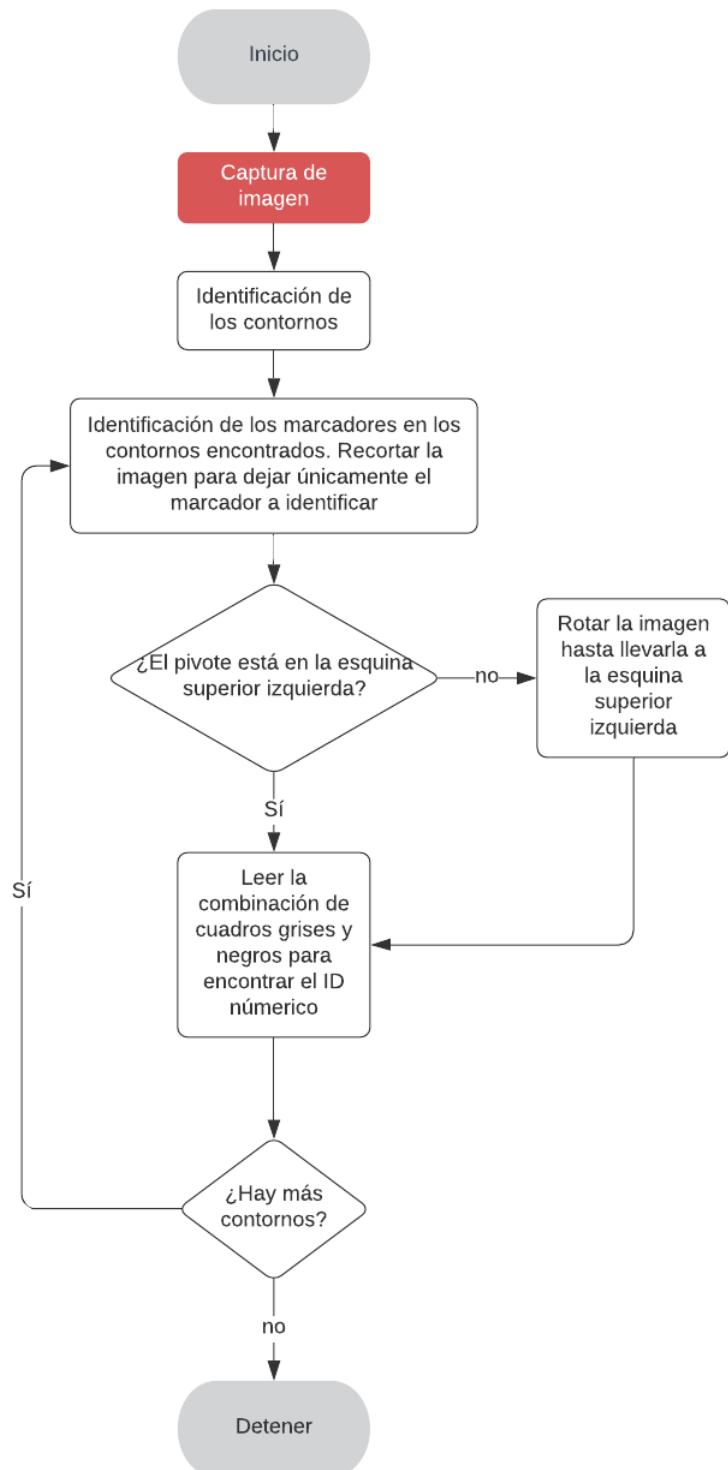


Figura 21: Diagrama de flujo para la identificación de un marcador.

Otras de las pruebas realizadas (utilizando el proceso descrito en las figuras 20 y 21), además de las pruebas de calibración, fueron en la parte de identificación de marcadores. Estas pruebas consistieron en capturar imágenes de la mesa, con los marcadores colocados sobre ella, para identificar el ID representado y la posición que tenían en la mesa. La Figura 22 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 24a, 24b y 23).

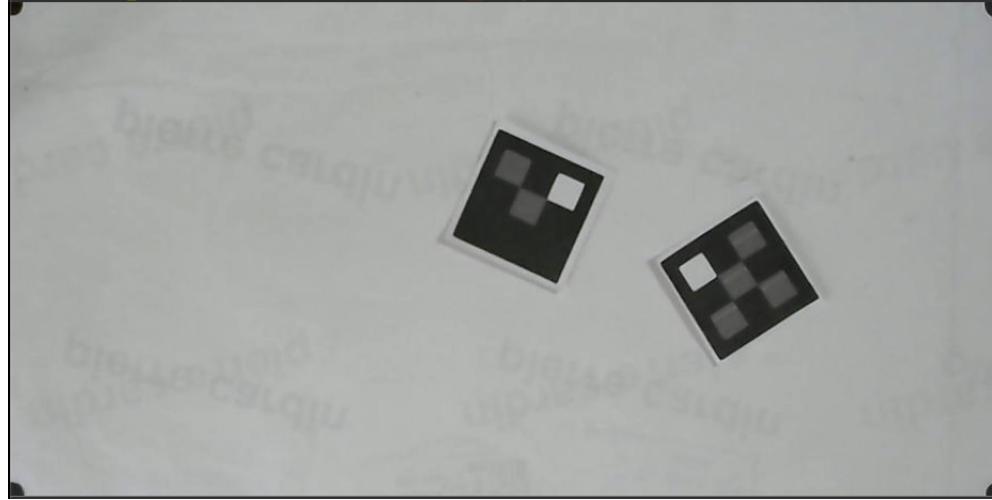


Figura 22: Imagen original para la identificación de los marcadores.

Como se muestra en la Figura 23, 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 23: 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 24b, y en la Figura 24a 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 24 como ejemplo en la interfaz gráfica.

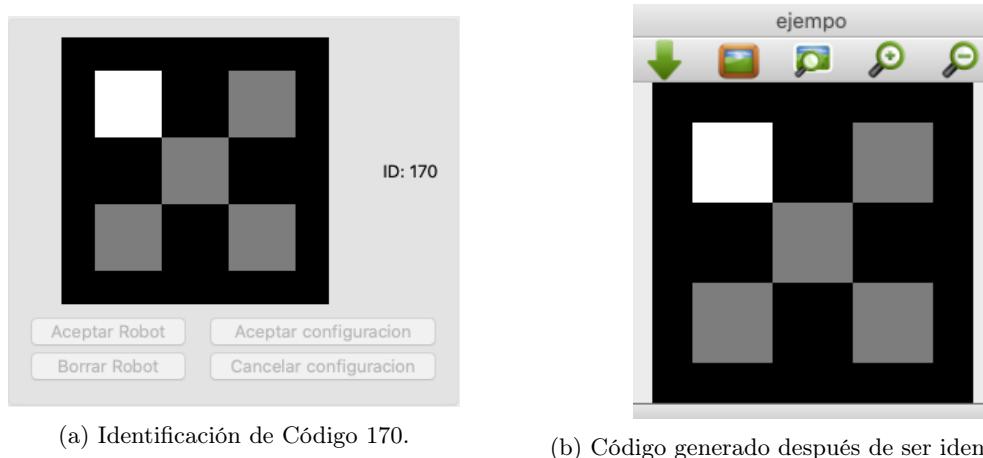


Figura 24: Identificación del marcador.

Para validar el funcionamiento correcto de este algoritmo se realizaron 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 %.

# CAPÍTULO 11

---

## Migración de 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 tres 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. Además, como parte de la migración, se desarrollaron librerías con las funciones ya existentes del algoritmo de la fase anterior. Estas librerías fueron implementadas utilizando clases y objetos, con el objetivo de utilizar la programación orientada a objetos (POO). El código 11.1 muestra una parte de las clases creadas utilizando POO en Python.

Además, el código 11.2 muestra un ejemplo del uso de estas clases en la migración del algoritmo. Este código sirve muestra como se manda a llamar a la clase `camara()` y se usa el método para capturar imagen y calibrarla (realizando el procedimiento que se explicará más adelante).

```

1  class camara():
2      def __init__(self, cam_num = 0, WIDTH = 960, HEIGHT = 720):
3          """
4              Parameters
5              -----
6              cam_num : TYPE, optional
7                  DESCRIPTION. The default is 0. Setea el valor de la camara en 0,
8                  normalmente una camara
9                  adicional (no la que la computadora trae) por default es 0
10             WIDTH : TYPE, optional
11                 DESCRIPTION. The default is 960. El ancho del marco de captura
12             HEIGHT : TYPE, optional
13                 DESCRIPTION. The default is 720. El alto del marco de captura
14
15             self.cap = cv.VideoCapture(cam_num)
16             self.cap.set(cv.CAP_PROP_FRAME_WIDTH, WIDTH)
17             self.cap.set(cv.CAP_PROP_FRAME_HEIGHT, HEIGHT)
18             self.cam_num = cam_num
19             self.img_counter_code = 0
20
21     def get_frame(self, capture_mode = "CONTINUE"):
22         """
23             Parameters
24             -----
25             capture_mode : TYPE string, optional, modo de captura
26                 DESCRIPTION. The default is "CONTINUE".
27                 Permite capturar un unico frame o mostrar un video en una pantalla
28                 y luego con la tecla escape capturar la foto. El modo CONTINUE
29                 es muy similar a lo que se ve en la camara de un telefono.
30             Returns
31             -----
32             TYPE numpy.array
33                 DESCRIPTION
34                     Retorna el frame capturado al momento de presionar la tecla ESC.
35
36             while True:
37                 if capture_mode == "SINGLE":
38                     ret, self.last_frame = self.cap.read()
39                     break
40                 elif capture_mode == "CONTINUE":
41                     ret, self.last_frame = self.cap.read()
42                     cv.imshow("test", self.last_frame) #muestra el video.
43                     k = cv.waitKey(1) #k = 1 es para espacio
44                     if k%256 == 27:
45                         cv.destroyWindow("test")
46                         cv.waitKey(1)
47                         break
48             return self.last_frame

```

Código 11.1: Ejemplo de implementación de POO en Python para la migración del algoritmo.

```

1 def capturar(self):
2     foto = camara.get_frame #metodo para capturar foto de la clase
3     camara()
4     CaliSnapshot = camara.Calibrar(foto,Calib_param,Treshold) #metodo
5     para la calibracion

```

Código 11.2: Ejemplo de captura de imagen y calibración de cámara usando POO en Python.

## 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 explicó: Identificar contornos circulares en la imagen y a partir de estos, definir las nuevas esquinas de la imagen. La Figura 25 muestra un ejemplo de una imagen que se utilizará para probar el algoritmo. El proceso general de como se calibra la cámara en C++ se muestra en la Figura 26 y el proceso para la calibración en Python se muestra en la Figura 27.

Tanto para Python como para C++, el proceso es igual al que muestra la Figura 26. Sin embargo, el proceso que se muestra en la Figura 27 es únicamente de la detección de borde circulares y posición de las esquinas utilizado en la migración. Este diagrama busca ilustrar los códigos que se mostrarán más adelante, para mostrarle al lector los pasos seguidos en el algoritmo de Python.



Figura 25: Ejemplo de imagen a calibrar en la migración del algoritmo.

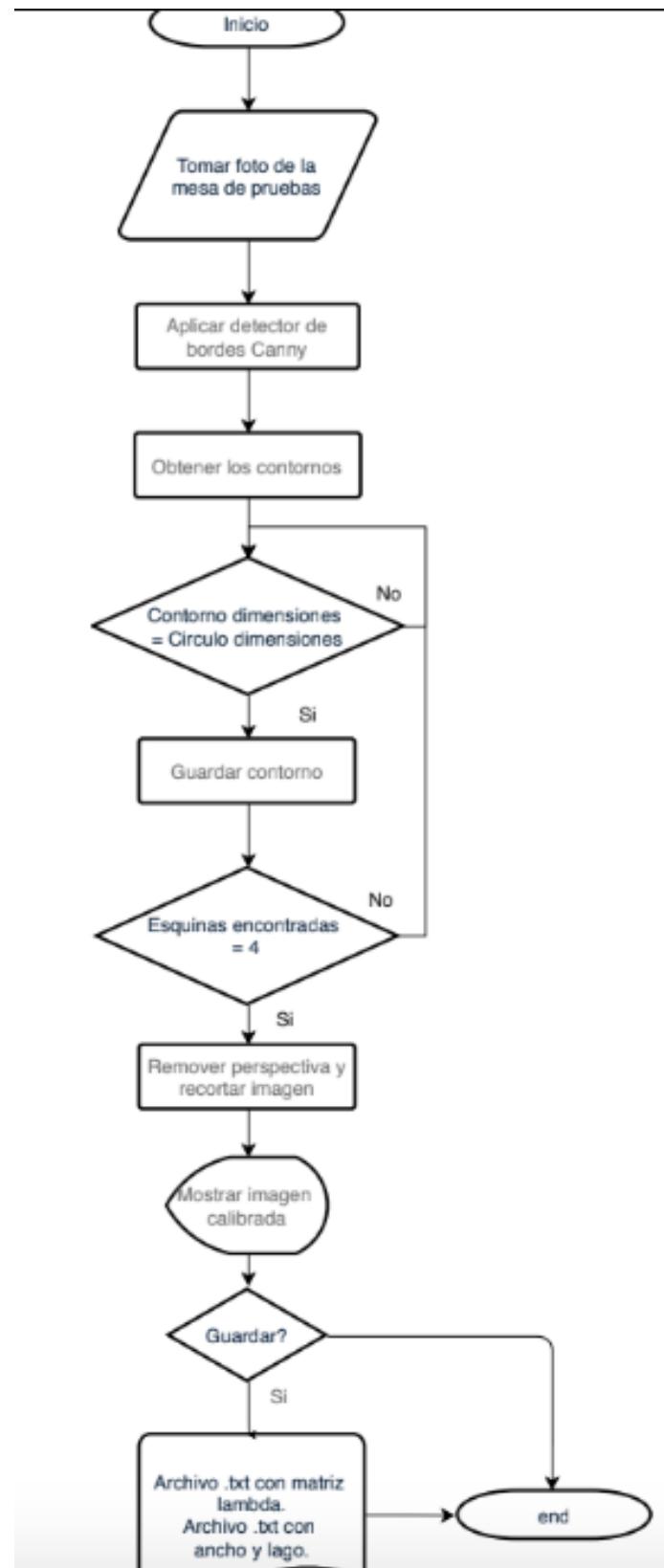


Figura 26: Proceso de calibración de la cámara en C++ [5].

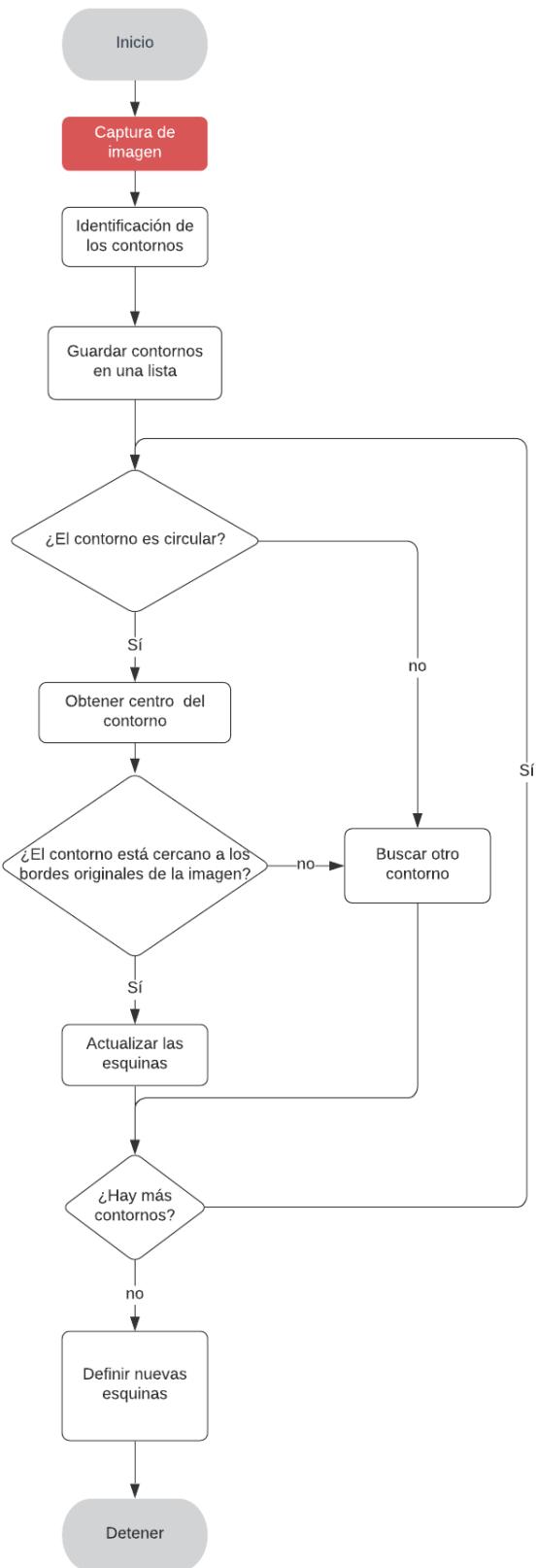


Figura 27: Proceso de detección de bordes y definición de esquinas en Python.

El código [11.3] 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 11.3: Detección de esquinas para calibrar cámara en C++.

El código [11.4] 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 mediante un cálculo matemático propio de la función. Esto permite tener un parámetro numérico de comparación para verificar si efectivamente se trata de un contorno circular. La identificación de contornos se hace utilizando la detección de bordes de `Canny` de la librería de OpenCV. Este algoritmo de `Canny` permite resaltar bordes (basados en ciertas cotas y cálculos propios de dicho método) para identificar de mejor manera los objetos de interés. Un ejemplo de contornos identificados se muestra en la Figura [28]. Como se dijo, el diagrama que se muestra en la Figura [27] describe el proceso que se utiliza en Python para la detección y ubicación de las nuevas esquinas.

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 [11.4]). Con esto, se obtienen los centros de cada contorno en la lista (línea 17 y 18 del código [11.4]) 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.

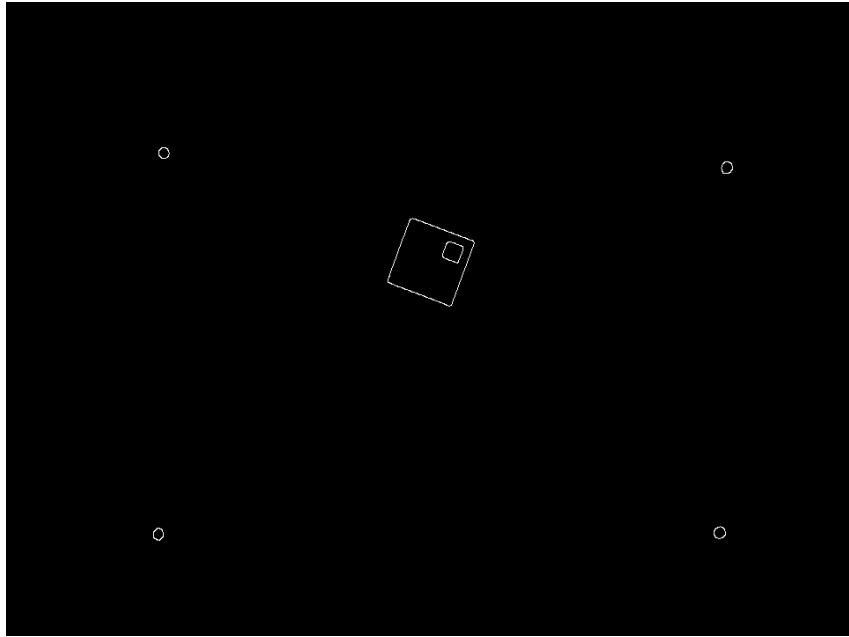


Figura 28: Ejemplo de contornos identificados por el algoritmo de Python.

```

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 11.4: Detección de esquinas para calibrar cámara en Python

Las figuras 29, 30, 31 y 32 muestran los resultados de los algoritmos de calibración en Python y C++. En ambos casos, la calibración es correcta (como muestran las figuras 30 y 32), dando validez a ambos métodos para la detección de las esquinas. El objetivo de esta implementación en Python, es poder enfocarse en los contornos circulares únicamente basados en propiedades numéricas (dadas por la aproximación de `cv.approxPolyDP()`) y áreas, y no tanto en rectángulos exteriores que encierran los contornos de interés. Además, permite ubicar los centros de estos contornos de manera más adecuada, para tener una mejor calibración de la cámara.

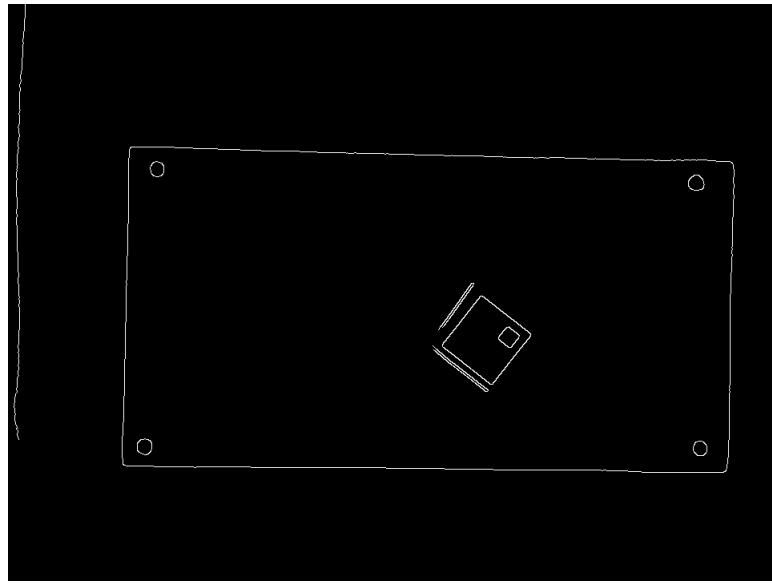


Figura 29: Contornos identificados para la calibración en Python.

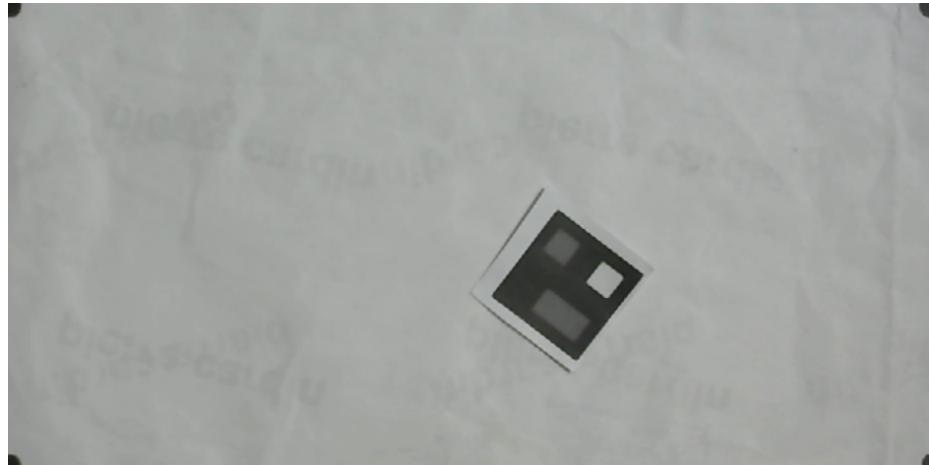


Figura 30: Ejemplo de imagen calibrada utilizando el algoritmo de Python.

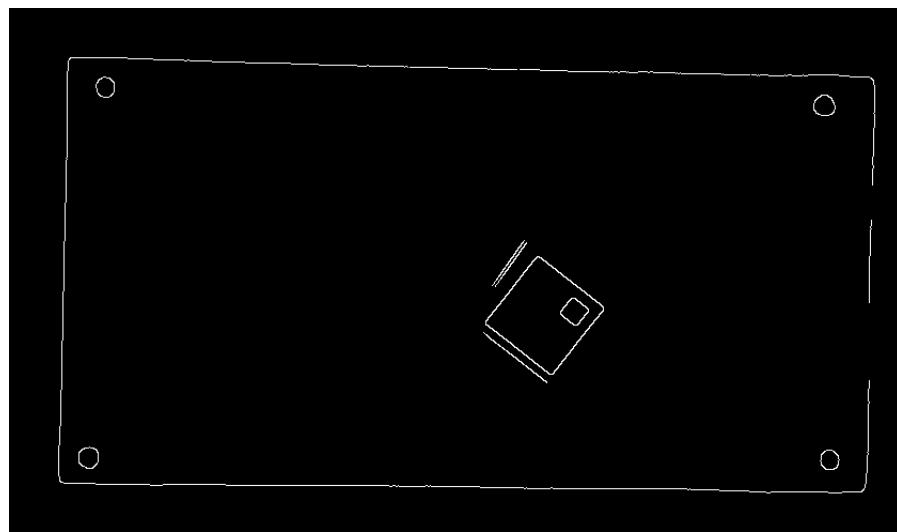


Figura 31: Ejemplo de contornos identificados por el algoritmo de C++

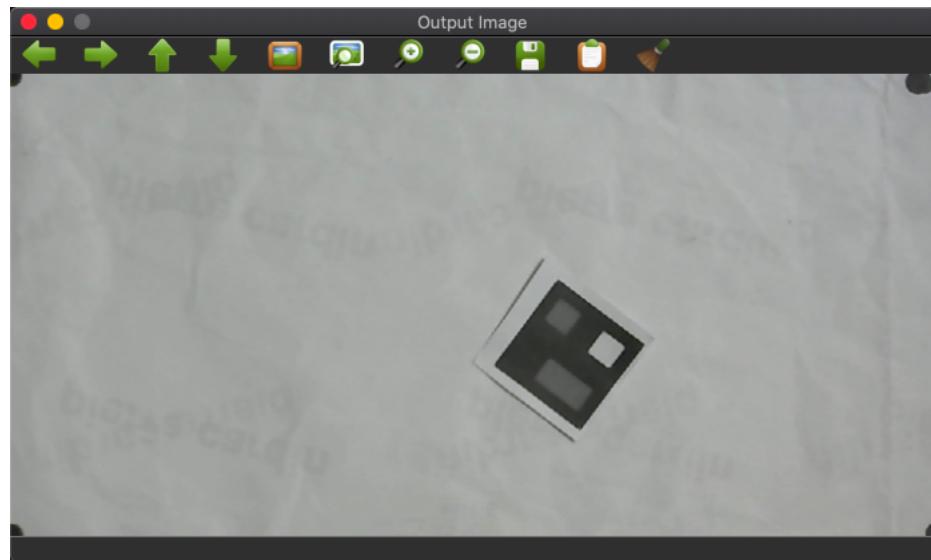


Figura 32: Ejemplo de imagen calibrada utilizando el algoritmo de C++.

Las figuras 33 y 34 muestran otro ejemplo de la aplicación del algoritmo de calibración en Python. Como se muestra en la Figura 33, el algoritmo es capaz de detectar los bordes (marcados con figuras circulares) y colocarlos como las esquinas que se desean a pesar de tener un contorno circular en el centro de la imagen. Esto debido a que, como ya se ha mencionado, el algoritmo busca los contornos que estén más cercanos a los bordes de la imagen original.

Por otra parte, la Figura 34 simula tener objetos en la mesa de pruebas, en este caso, un objeto rectangular y uno que simula ser un robot. Nuevamente, es capaz de detectar los contornos circulares (por eso también marca al robot) pero toma únicamente aquellos que estén más cercanos a las esquinas o bordes de la imagen original, lo que hace que el algoritmo sea robusto ante objetos que pueden estar colocados en la mesa. Sin embargo, es posible que, si hay algún contorno circular fuera del área de interés, el algoritmo lo puede tomar como una posible esquina y calibrar erróneamente.

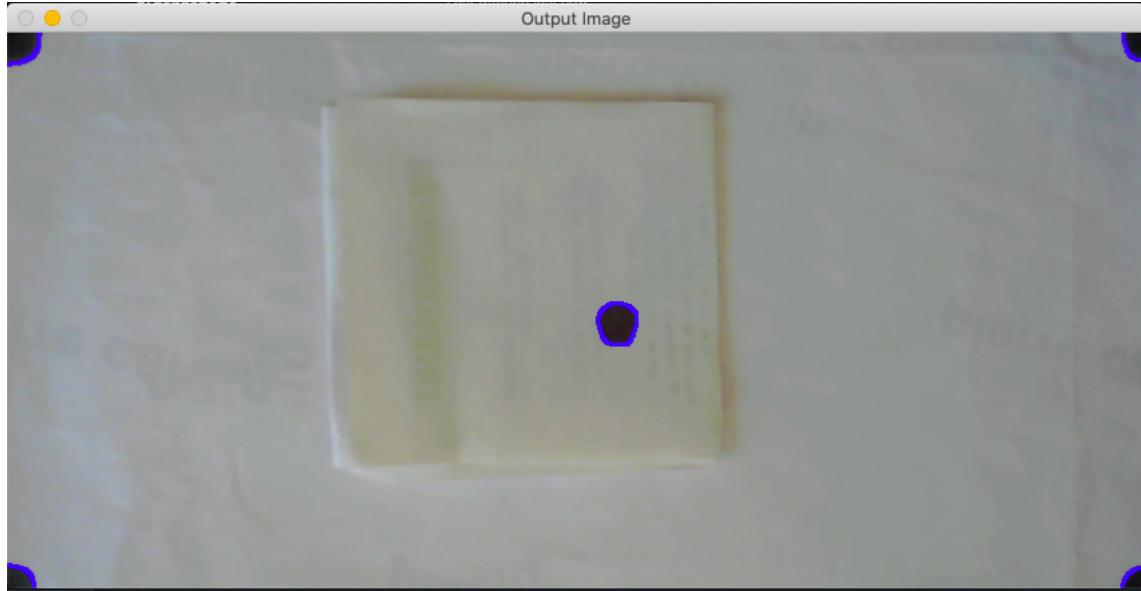


Figura 33: Primera prueba calibración de la cámara con contorno en la mesa utilizando Python.

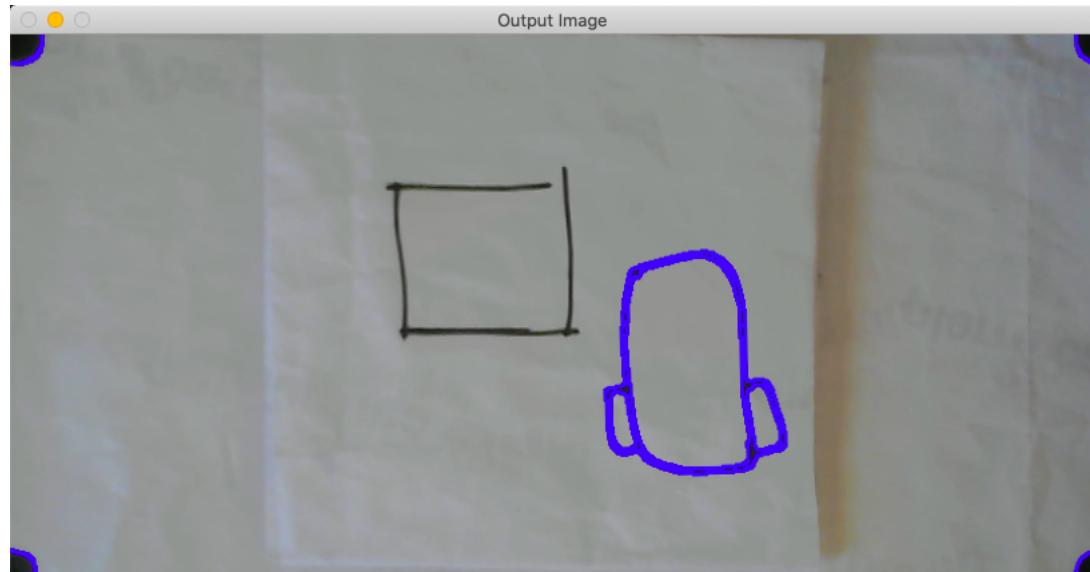


Figura 34: Segunda prueba calibración de la cámara con objetos en la mesa utilizando Python.

## 11.2. Migración del código para generación de marcadores e identificación

### 11.2.1. Generación del marcador

En la subsección [10.2] se muestra la Figura [20], ubicada en la página [31], que muestra el diagrama para la generación del marcador en C++. En general, el proceso es similar en Python y el código [11.5] muestra el programa que genera el marcador. La Figura [35] muestra un ejemplo de un marcador generado. Como se puede observar, en la interfaz se coloca un número entre 0 y 255. Por lo ya explicado, al ser un ID de 1 byte (8 bits), el máximo número posible puede ser 255 y el mínimo es 0. Posterior a esto, utilizando el código [11.5] se genera la imagen, tal cual lo muestra la Figura [35]. Otro ejemplo de otro marcador generado se muestra en la figura [36].

```
1 for u in range (0,3):
2     for v in range (0,3):
3
4         #para generar el pivote (ver la tesis de Andre)
5         #El pivote sirve para saber que cuadro debe estar alineado.
6         if k == -1:
7             for i in range(u*50+25, u*50+75):
8                 for i2 in range(v*50+25,v*50+75):
9                     Cod[i,i2] = 255 #llena el pivote, 255 = blanco
10        else:
11            #genera los otros cuadros en escala de grises, 125 = gris
12            t = num[7-k]
13            n = int(t)
14            for i3 in range(u*50+25, u*50+75):
15                for i4 in range(v*50+25,v*50+75):
16                    Cod[i3,i4] = n * 125
17
18        k = k + 1
19 cv.imshow('cod', Cod)
20 cv.waitKey(5000)
21 self.destroy_window()
22 #Formato del nombre de la imagen.
23 edge_img = "opencv_CodGenerator_{}.png".format(self.img_counter_code)
24 #Guarda el numero de frame (foto) que se tomo.
25 cv.imwrite(edge_img, Cod) #Guarda la foto
```

Código 11.5: Generación del Marcador en Python.

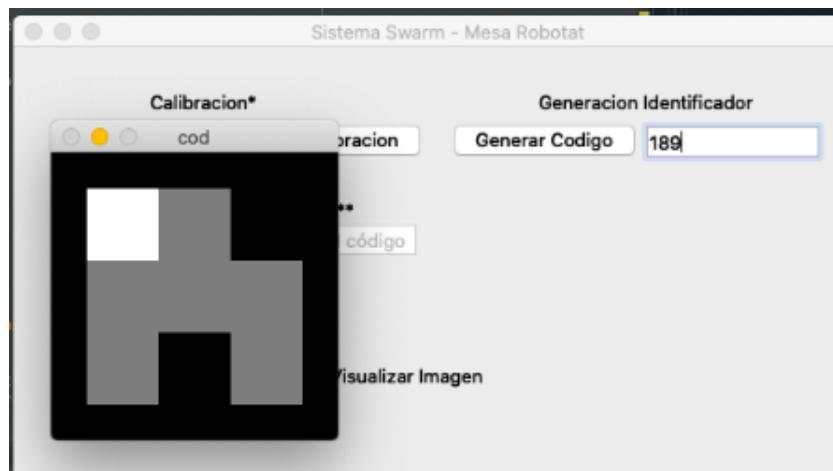


Figura 35: Ejemplo de marcador generado en Python con el ID 189.

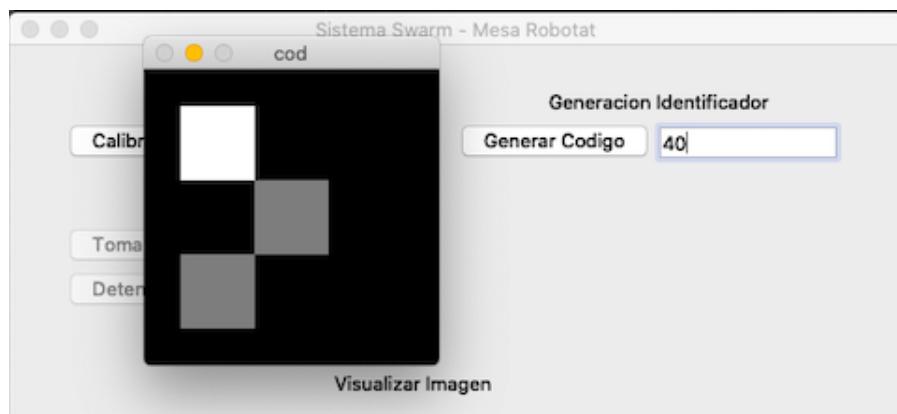


Figura 36: Ejemplo de marcador generado en Python con el ID 40.



Figura 37: Marcador generado con el ID 178

### 11.2.2. Identificación del ID

Una vez realizada la migración de la generación del marcador, se procedió a realizar la migración el código de la identificación. La detección del ID, se hace siguiendo el diagrama que se muestra en la Figura 38.

A manera comparativa se muestra una identificación de un marcador con ID 40 en Python y C++. El primer caso es utilizando C++. Una vez se ubica el marcador que se desea identificar, se realiza el código 11.6, que muestra como el algoritmo identifica el ID del marcador. Como se muestra en el código, se utiliza un ciclo para recorrer la imagen y así identificar el valor que se representa.

La Figura 39 muestra la imagen que se utilizará de ejemplo para la detección del marcador. Como se muestra en la Figura 40, una vez realizada la rotación de la imagen para colocar el pivote en la esquina superior izquierda, y utilizando el código 11.6, se identificó que el ID del marcador era de 40.

Ahora, el código 11.7 muestra como es el programa para la identificación en Python. Debido a que el marcador está conformado por 8 cuadros (aparte del blanco que representa el pivote), en Python se procede a cortar la imagen en cada uno de estos cuadros (líneas 1 a la 3 del código 11.7 que muestra como se obtienen 3 de los 8 cuadros). El valor de este cuadro recortado (que puede ir desde 0 cuando es color negro, hasta 255 siendo este un color blanco) se compara con una cota inferior y superior para determinar si esta dentro del rango que se requiere. El objetivo es ubicar cuadros cuyo rango este aproximadamente entre 125 y 175 (lo que representa un color gris y por ende, un número 1 en el código binario). 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 para luego obtener el número en decimal que representa (línea 31). Finalmente, la Figura 41 muestra como el programa es capaz de identificar correctamente el valor del marcador al igual que como lo hace en C++ en la misma imagen que muestra la Figura 39.

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 %. Aunque si las condiciones de luz no son adecuadas, puede que la identificación del ID sea errónea. En la siguiente secciones se muestran otras pruebas variando el tamaño de los marcadores, y con esto, mostrar otros casos del funcionamiento de este programa.

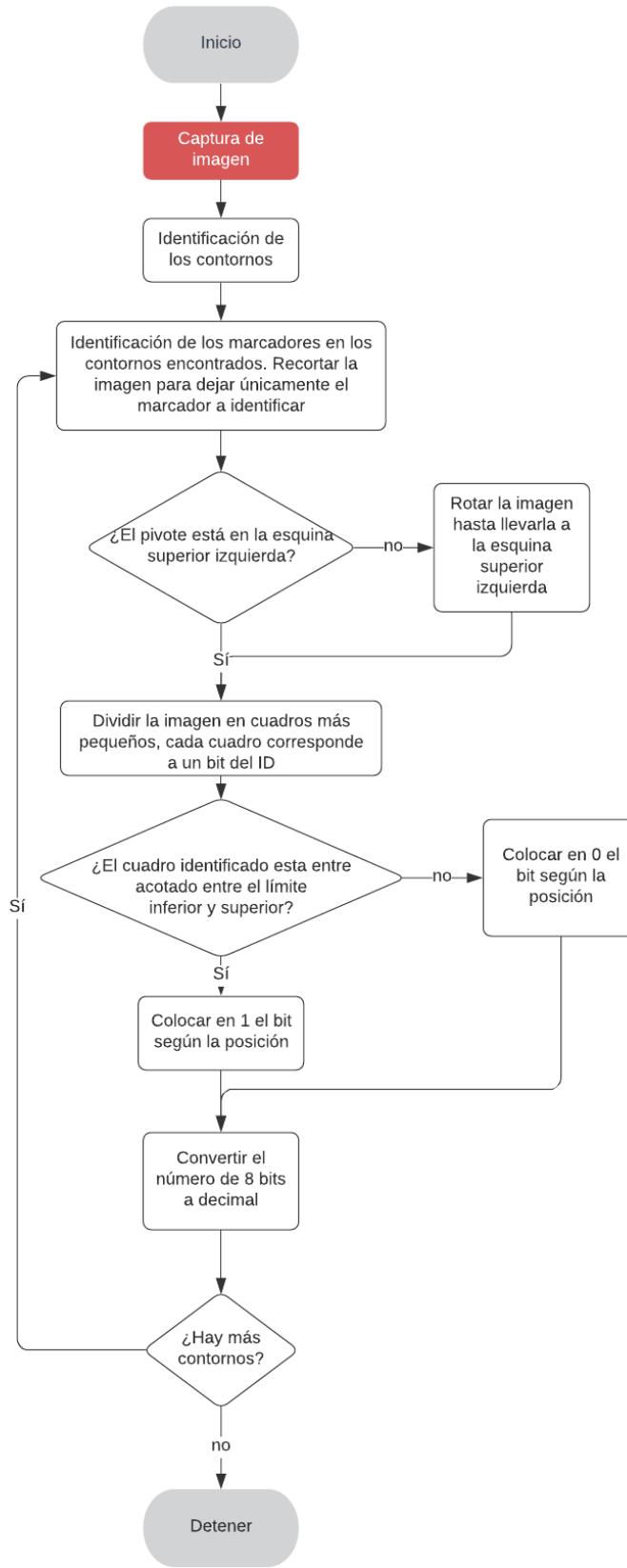


Figura 38: Diagrama de flujo para el reconocimiento del ID en el algoritmo de Python.

```

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()
5             .height * u / 4), (finalCropCodRotated.size().width * v / 4));
6         MatrizValColores[(u - 1)][(v - 1)] = ValColorTemp;
7         if ((ValColorTemp < EscalaColores[2]
8             GlobalColorDifThreshold) && (ValColorTemp > EscalaColores[0] +
9                 GlobalColorDifThreshold)) {
10            EscalaColores[1] = ValColorTemp;
11        } } }
12 //Extraemos el codigo binario
13 string CodigoBinString = "";
14 for (int u = 0; u < 3; ++u) {
15     for (int v = 0; v < 3; ++v) {
16         if ((u == 0) && (v == 0))
17             CodigoBinString = CodigoBinString;
18         else if ((MatrizValColores[u][v] > EscalaColores[1] -
19             GlobalColorDifThreshold) && (MatrizValColores[u][v] < EscalaColores[1] +
20                 GlobalColorDifThreshold))
21             CodigoBinString = CodigoBinString + "1";
22         else
23             CodigoBinString = CodigoBinString + "0";
24     }
25 }

```

Código 11.6: Reconocimiento del ID en C++

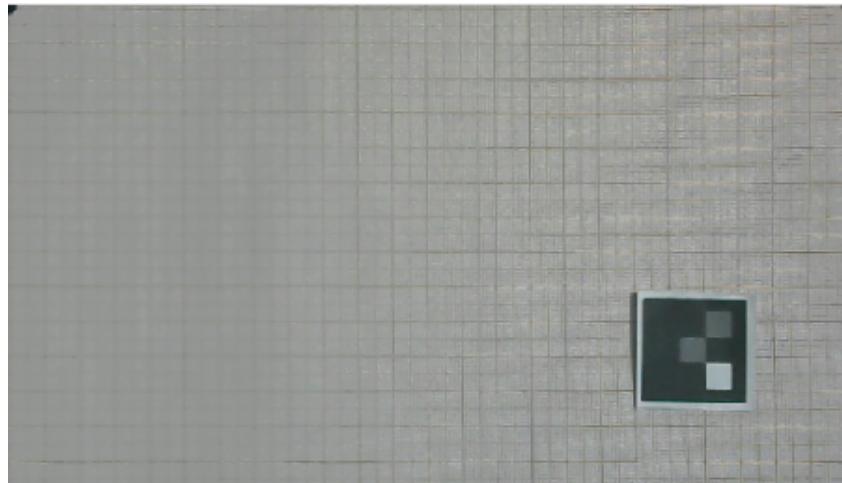


Figura 39: Ejemplo de imagen para identificar el marcador.

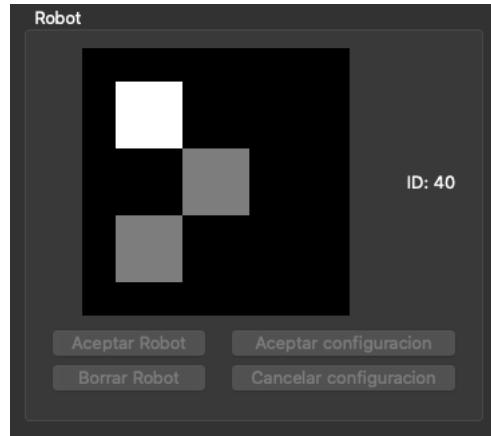


Figura 40: Marcador reconocido en C++.

```

1 temp_a1 = resized[int(height_Final_Rotated*1/8):40 , 65:100]
2 temp_a5 = resized[int(height_Final_Rotated*1/4 + 42):int(
    height_Final_Rotated*1/2 + 40) , int(height_Final_Rotated*1/8):int(
    height_Final_Rotated*1/8 + 23)]
3 temp_a7 = resized[int(height_Final_Rotated*1/2) + 15:int(
    height_Final_Rotated*1/2) + 45 , int(width_Final_Rotated*1/2)+20 :int(
    width_Final_Rotated*1/2) + 45]
4           ****OTRAS LNEAS AQUI*****
5 """
6 Se define un TRESHOLD_DETECT_MAX y TRESHOLD_DETECT_MIN para que el valor que
    tome como gris se adecuado.
7 Estos valores pueden variar segun la luz (normalmente varia
    TRESHOLD_DETECT_MIN)
8 """
9 for i in range (0, len(code)):
10 #con los thresholds establecidos, busca que valores sean grises y los
    cataloga como 1, sino, los cataloga como 0.
11     if code[i] > TRESHOLD_DETECT_MIN and code[i]< TRESHOLD_DETECT_MAX:
12
13         CódigoBinString = CódigoBinString + "1"
14     else:
15         CódigoBinString = CódigoBinString + "0"
16           ****OTRAS LNEAS AQUI*****
17 #esta funcion pasa el string de bits a formato de numero int.
18     tempID =int(CódigoBinString , 2)

```

Código 11.7: Reconocimiento del ID en Python

```

opencv_CalibSnapshot_0.png Canny Guardado!
Soy el hilo: 1
Entre a process_image
ID del robot 40

```

Figura 41: Resultado luego de la identificación del marcador en Python.

## CAPÍTULO 12

---

### Pruebas de detección de marcadores con diferentes tamaños en Python

---

Con el objetivo de agregar versatilidad a este algoritmo, se realizaron pruebas para identificar marcadores con diferentes tamaños. Las figuras 42, 43, 44 y 45 muestran cómo el algoritmo modificado en Python toma la imagen original y la reescalía a un tamaño estándar para su identificación. Dicho tamaño es de  $116 \times 116$  píxeles. 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 esta dimensiones en píxeles. Debido a la necesidad de partir el marcador en los pequeños cuadros que lo conforman (como se mencionó en el capítulo anterior), y ya que la división se hace manualmente ubicando los cuadros en la imagen basado en el ancho y alto de esta, resulta más simple llevarlo a este tamaño estándar para que se aplique a un único caso sin importar el tamaño del marcador, y no tener que ir variando donde se ubican los cuadros en la imagen si el tamaño es mayor o menor a  $3 \times 3$  cm.

Cabe mencionar que la identificación en marcadores 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.

Como se puede observar en la Figura 42, 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 42c y 42b 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 43 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 42c o 42b. Aún así, si las condiciones de luz no son adecuadas, puede que la identificación del marcador no sea correcta.

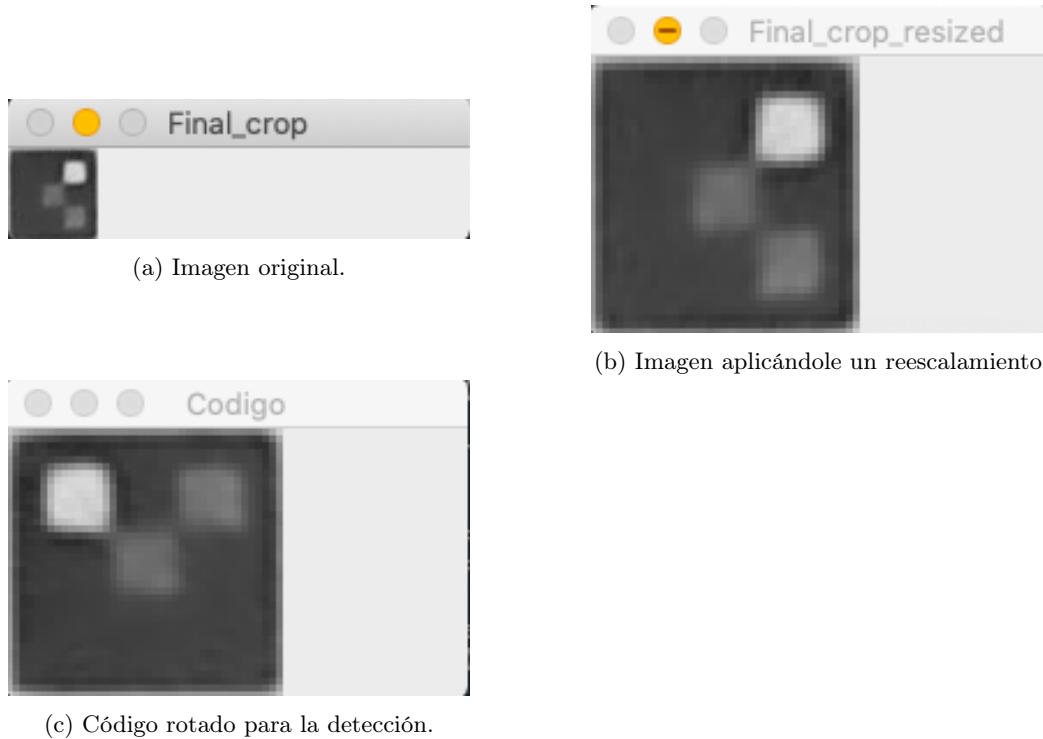
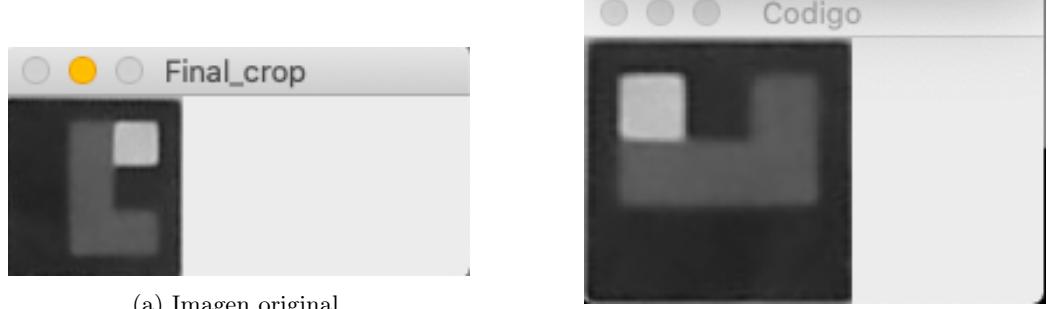


Figura 42: Detección y reescalamiento de código generado para tamaño  $1 \times 1$  cm.



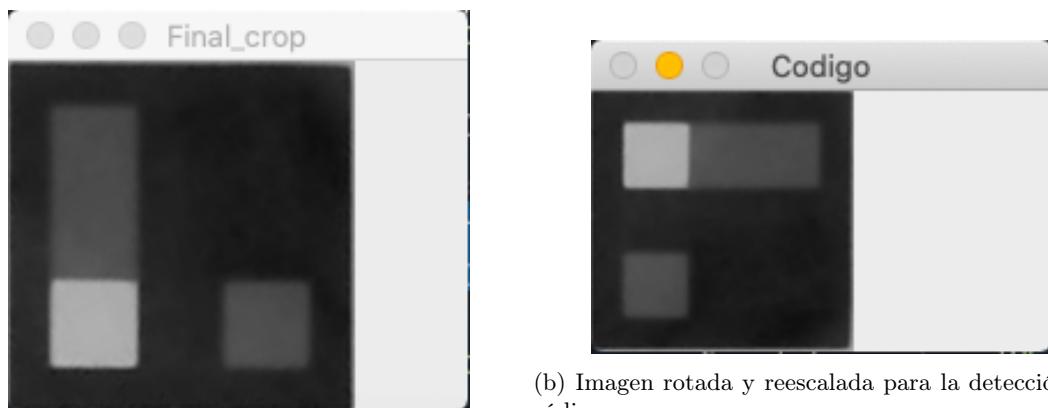
(a) Imagen original.

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

Figura 43: 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. Esto se observa en las figuras 44 y 45, 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 la identificación puede funcionar mejor al pasar de dimensiones más grandes hacia dimensiones pequeñas (en caso de ser necesario) y no tanto a la inversa.

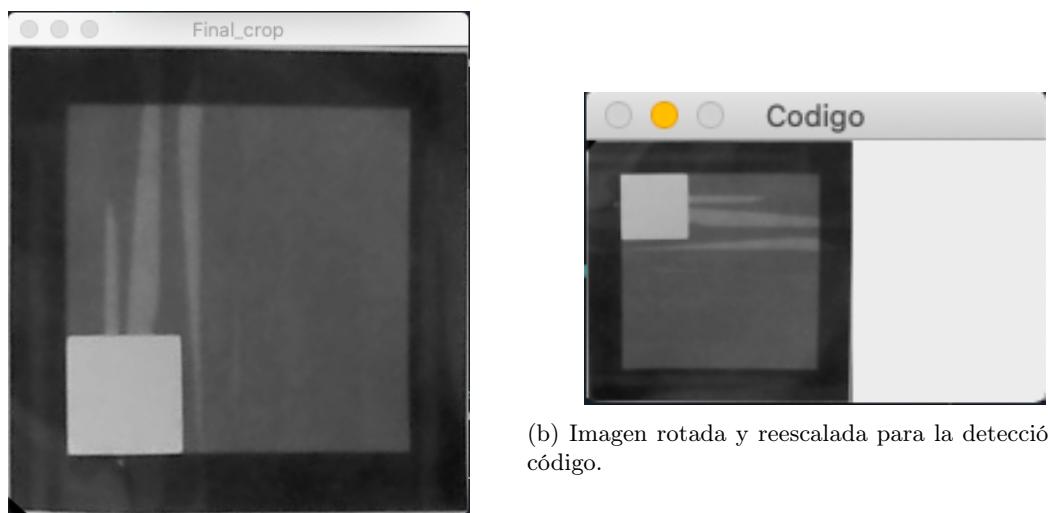
La ventaja de pasar a tamaños más pequeños (un tamaño definido) es , como se ha mencionado, que el algoritmo ubica los diferentes cuadros que conforman los marcadores para obtener el ID que representa. Al ser esto una ubicación ya definida, es decir, dentro del código se define las posiciones que debe buscar en la imagen, resulta más fácil pasar de cualquier tamaño (más grande o más pequeño) hacia el tamaño de  $3 \times 3$  cm, siendo este el tamaño estándar, 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 44: 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 45: Detección y reescalamiento de código generado para tamaño de  $8 \times 8$  cm.

# CAPÍTULO 13

---

## Comparación entre los algoritmos de C++ y Python

---

### 13.1. 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  $24 \times 16$  cm. La Figura 46 muestra una imagen ya calibrada por el algoritmo utilizando esta mesa. La Figura 47 muestra los ejes de referencia para la toma de estos datos. Como se observa, el origen se encuentra ubicado en la esquina superior izquierda de la imagen. Los valores positivos del eje X se encuentran hacia la derecha de la imagen y los valores positivos del eje Y se encuentra hacia abajo de la imagen.

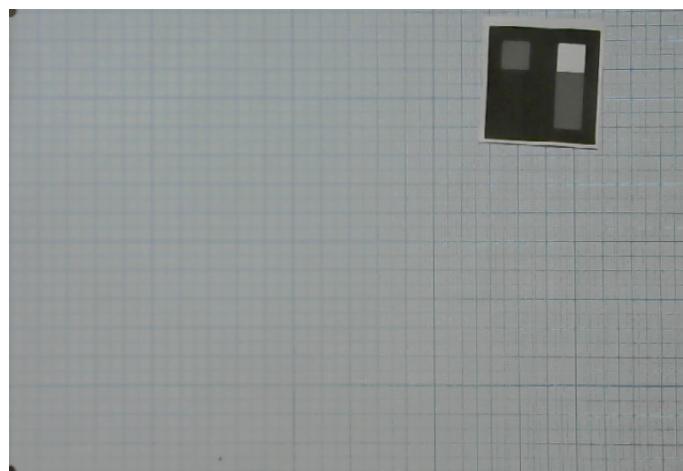


Figura 46: Mesa de pruebas para la obtención de la posición real de los objetos

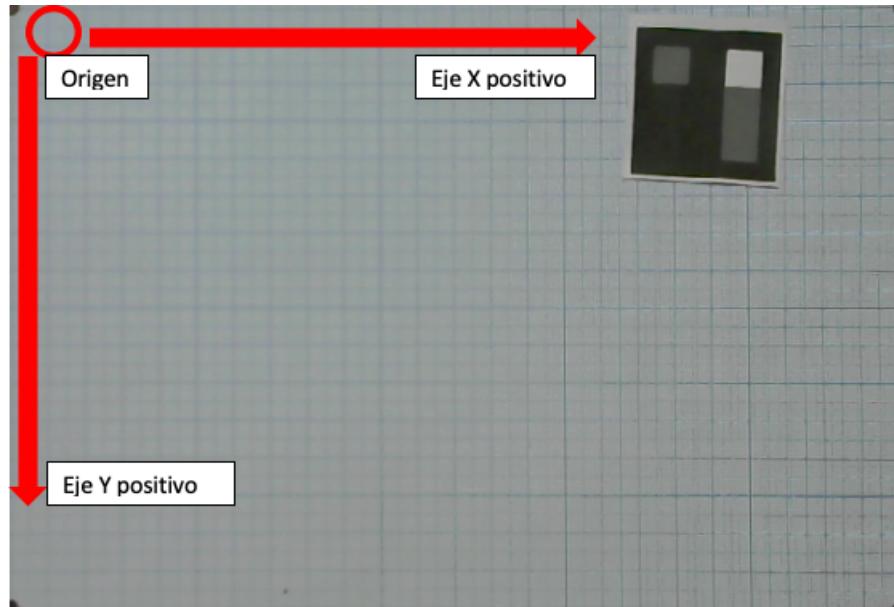


Figura 47: Marco de referencia para la toma de pose.

El algoritmo se corrió 30 veces (15 aproximando a enteros la posición y 15 sin aproximación a enteros) en ambos lenguajes y el Cuadro 1 muestra la posición real de los robots en la mesa, obtenidas en las primeras 15 corridas. Los resultados de las ejecuciones se muestran en el Cuadro 2. Como es posible observar, la posición es igual en ambos casos (tanto Python como C++) y resultan muy similares también con respecto a la posición real en la mesa. Sin embargo, como el algoritmo aproxima las medidas a números enteros, es obvio que la variación con respecto a la posición real sea de decimas. Se realiza esta primera prueba comparativa con la posición aproximada a enteros ya que el algoritmo original lo presentaba. Aunque luego se realizaron otras pruebas para mostrar otros puntos de comparación en estas medidas.

Posición real		
X (cm)	Y (cm)	Theta (grados)
9.5	7.5	85
5	3	90
20	13	90
12	4.7	90
14.5	8	149
19.5	13.5	25
3.5	13.5	120
10.5	8	68
20	7.5	114
9	7.5	177

Cuadro 1: Posición real para el caso con aproximación a enteros.

Python			C++		
X (cm)	Y (cm)	Theta (grados)	X (cm)	Y (cm)	Theta (grados)
9	7	85	9	7	85
5	3	90	5	3	91
19	13	90	19	13	90
11	4	91	11	4	90
14	8	147	14	8	147
19	13	25	19	13	25
3	13	121	3	13	121
10	8	68	10	8	68
19	7	114	19	7	114
9	7	177	9	7	176

Cuadro 2: Posición detectada por los algoritmos de Python y C++ con aproximación a enteros.

Como se mencionó, el Cuadro 2 muestra los resultados del algoritmo luego de aproximar las medidas a números enteros, sin embargo, para tener más información de la posición, se realizaron las mismas pruebas pero dejando valores decimales. El Cuadro 3 muestra la posición real en la mesa y el Cuadro 4 muestra la posición detectada por los algoritmos sin aproximar a enteros. 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.

Posición real		
X (cm)	Y (cm)	Theta (grados)
10.5	10.5	15
19.7	3	90
5.5	3.5	32
10.3	12.5	80
3.5	8	15
10.1	7.8	84
10	3	55
16	9.5	150
20.2	13	144
19.5	7.6	127

Cuadro 3: Posición real para el caso sin aproximación a enteros.

Python			C++		
X (cm)	Y (cm)	Theta (grados)	X (cm)	Y (cm)	Theta (grados)
9.93	11.7	14	10.03	11.6	14
19.44	3.0	90	19.62	2.94	90
5.28	3.48	32	5.2	3.51	32
9.94	12.79	79	10.05	12.63	79
3.43	7.76	15	3.59	7.9	15
10.05	7.91	84	10.12	7.84	84
9.53	3.33	57	9.47	3.3	59
15.41	9.33	149	15.59	9.21	149
19.62	13.2	144	19.85	12.97	144
19.47	7.8	127	19.71	7.73	127

Cuadro 4: Posición detectada por los algoritmos de Python y C++ sin aproximación a enteros.

Finalmente se hizo un análisis estadístico de los datos obtenidos en los cuadros 2 y 4. 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 mejor referencia de la diferencia en la posición.

En el Cuadros 5 y Figura 48 se observan las frecuencias de la diferencia (en centímetros) entre el algoritmo de Python y C++. Como se muestra, la frecuencia cuando se realiza una aproximación a números enteros es prácticamente 0 centímetros (Cuadro 5), 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 48. 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 5: Frecuencia para la toma de pose con aproximación a entero (eje x y eje y)

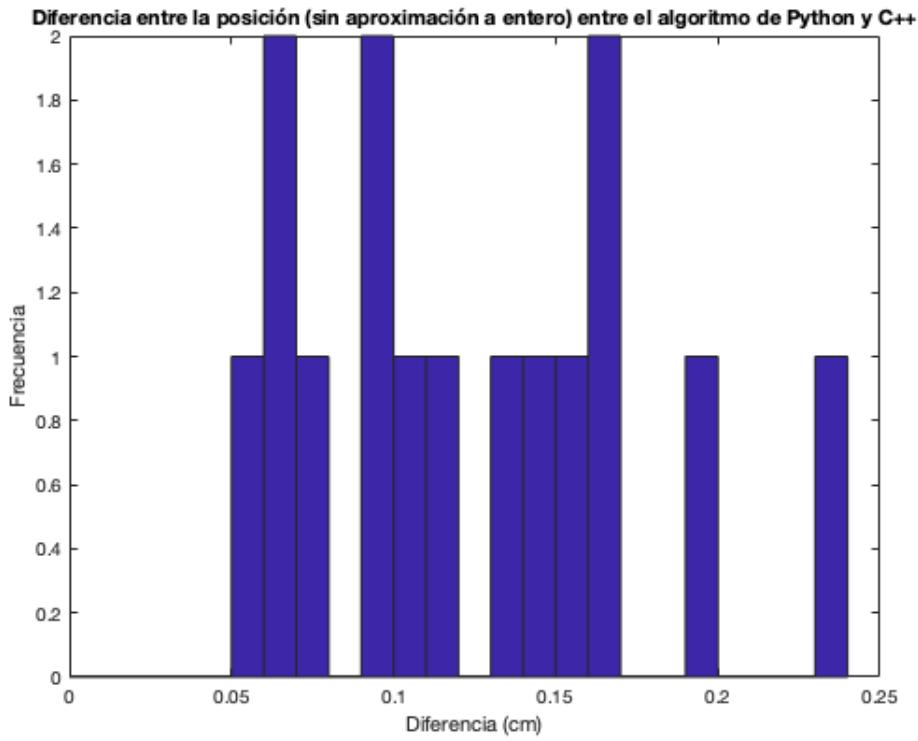


Figura 48: Frecuencia para la diferencia de la posición entre Python y C++ sin aproximación a entero

Finalmente, con respecto a la posición real, la Figura 49 muestra que la diferencia que más se repite es aproximadamente 0.5 centímetros, que fue calculada con la posición obtenida en Python. La Figura 50 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.

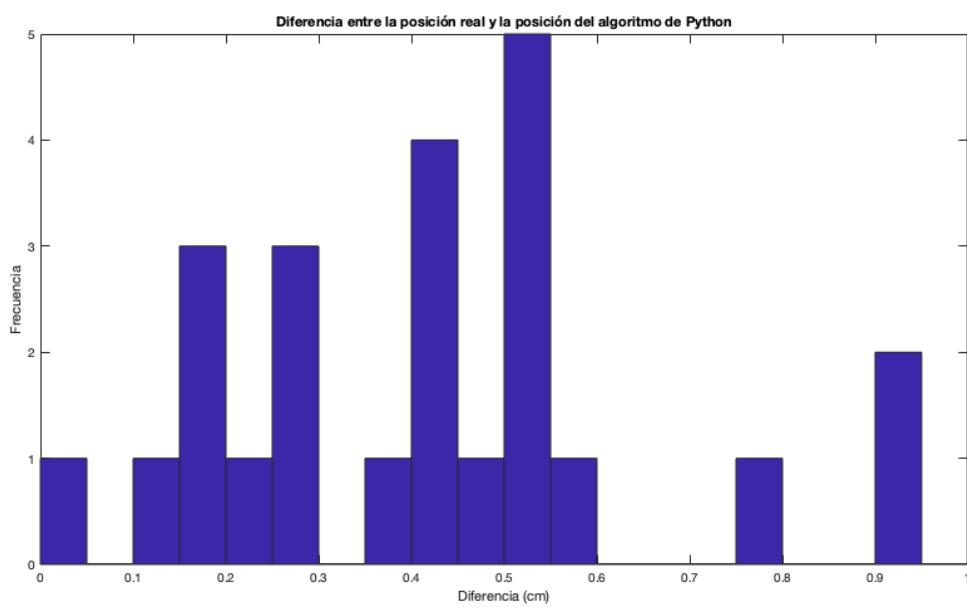


Figura 49: Frecuencia para la toma de pose en Python con respecto a la posición real

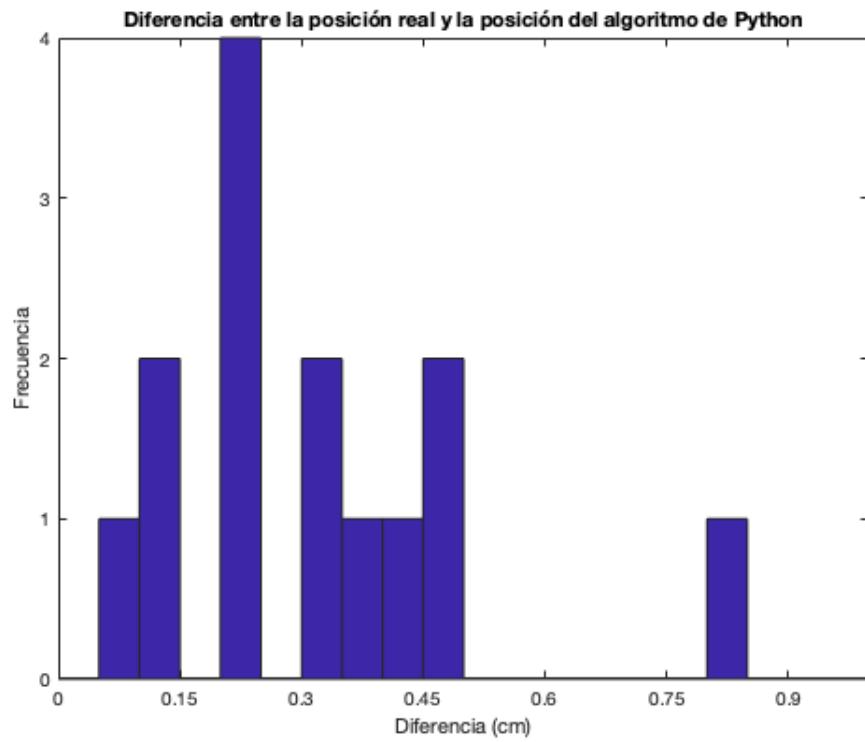


Figura 50: Frecuencia para la toma de pose en C++ con respecto a la posición real

Además de calcular la frecuencia, se calculó la media y la moda para cada caso mostrado en las figuras 48, 49 y 50. Los resultados se muestran en los cuadros 6, 7, 8 y 9. 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 diferencia media de 0.467 grados. En el Cuadro 7 se puede observar que la media ya es distinta, ya que aquí se dejaron los datos con cifras decimales. Para este caso, la diferencia promedio es de 0.1157 centímetros con una moda de 0.07 cm respectivamente. Para el ángulo se obtiene que la diferencia promedio es de 0.2 grados, siendo la moda de 0 grados, para este caso.

Parámetro	Media	Moda	Desviación estándar
Posición (cm)	0	0	0
Ángulo (grados)	0.467	0	0.6399

Cuadro 6: Media y moda para la posición con aproximación a enteros entre los algoritmos de Python y C++

Parámetro	Media	Moda	Desviación estándar
Posición (cm)	0.1157	0.085	0.0526
Ángulo (grados)	0.2	0	0.5606

Cuadro 7: Media y moda para la posición sin aproximación a enteros entre los algoritmos de Python y C++

Ahora, con respecto a la posición real, el Cuadro 8 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 9 indican que la media de la diferencia es de 0.27 centímetros, lo que representa que la diferencia del algoritmo de Python con respecto a las medidas reales es un 27% mayor en comparación a la diferencia para el algoritmo de C++. 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).

Parámetro	Media	Moda	Desviación estándar
Posición (cm)	0.3758	0.5	0.2314

Cuadro 8: Media y moda de la diferencia con respecto a la posición real del algoritmo en Python (ambos ejes).

Parámetro	Media	Moda	Desviación estándar
Posición (cm)	0.2736	NA	0.1968

Cuadro 9: Media y moda de la diferencia con respecto a la posición real del algoritmo en C++ (ambos ejes).

En los cuadros [10] y [11] se muestran las diferencias de la posición detectada por los algoritmos tanto de Python como de C++ pero realizando una comparación por eje y sin aproximar a enteros. Como es posible observar, tanto para Python como para C++ la diferencia entre las medias de los ejes es pequeña, siendo 0.13 cm para Python y 0.06 cm para C++. Además, la moda en Python también tiene concordancia con los datos presentados en el Cuadro [8]. Finalmente, es posible ver que independientemente del lenguaje utilizado, el eje que en promedio representa una mayor diferencia es el eje X. Aún así, la diferencia en ambos ejes es menor a 0.5 cm lo cual continúa siendo un buen resultado tomando el tamaño de la mesa donde se realizaron estas pruebas.

Parámetro	Media	Moda	Desviación estándar
Eje X (cm)	0.4408	0.5	0.3063
Eje Y (cm)	0.3108	0.5	0.2855

Cuadro 10: Diferencia por eje (X,Y) entre la posición detectada por el algoritmo de Python y la posición real.

Parámetro	Media	Moda	Desviación estándar
Eje X (cm)	0.3042	NA	0.2201
Eje Y (cm)	0.2429	0.04	0.2947

Cuadro 11: Diferencia por eje (X,Y) entre la posición detectada por el algoritmo de C++ y la posición real.

### 13.2. Comparación de rendimiento y tiempos de ejecución

El Cuadro [12] muestra los tiempos de ejecución entre algoritmos con su implementación multihilos. 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 dos hilos de procesamiento, uno de actualizar los datos de los robots y el hilo principal. El algoritmo de C++ tiene únicamente dos hilos, el principal y uno de procesamiento. Esta diferencia recae en poder aportar modularidad al código de Python, ya que separando el procesamiento en dos, 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 dos hilos adicionales y al *overhead* al pasar de un hilo a otro, así como también a que los hilos posiblemente se procesan en un solo *core*. Aunque como se mencionó, un hilo es para actualización de datos por lo que podría ser eliminado.

Tomando en cuenta lo anterior, al eliminar el hilo de actualización de datos, y unificar los dos hilos de procesamiento, se obtiene los resultados que muestra el Cuadro 13. Esta es una implementación muy similar a la que se realizó C++, pues tiene únicamente dos hilos. Como se observa, existe una mejora en comparación con su versión teniendo cuatro 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. est\'andar	T. M\'aximo(s)	Repeticiones
Python	0.2519 s	0.29 s	0.031	0.31	45
C++	0.1083 s	0.1073 s	0.00597	0.1402	45

Cuadro 12: Medidas de tiempo para los algoritmos usando 4 hilos para Python y 2 hilos para C++

Lenguaje	Media	Moda	Desv. Est\'andar	T. M\'aximo(s)	Repeticiones
Python	0.06358 s	0.047 s	0.02659	0.119	45

Cuadro 13: Medidas de tiempo para el algoritmo en Python usando 2 hilos

También se realizó un cálculo del tiempo de ejecución sin hilos. El Cuadro 14 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 dos hilos y 4.5 veces más rápido que su versión con 4 hilos. Sin embargo, el tiempo de ejecución en C++ sin hilos es muy similar al tiempo de ejecución en la implementación con dos hilos.

Lenguaje	Media	Moda	Desv. est\'andar	T. M\'aximo(s)	Repeticiones
Python	0.05513 s	0.085 s	0.02808	0.100	45
C++	0.01076 s	0.1070 s	0.0063	0.1388	45

Cuadro 14: Medidas de tiempo para el algoritmo en Python y C++ 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. Además, ayuda a que un programa no sea secuencial, es decir, que no este esperando a que un recurso se libere o se obtenga para seguir su ejecución.

Un ejemplo de lo anterior es tener una función de captura continua de imagen, es decir, que el programa este capturando fotos de la mesa cada cierto tiempo de manera autónoma. Sin hilos, por ejemplo, el programa deberá tomar una foto, luego procesarla y esperar a que este proceso termine para capturar la siguiente imagen. Sin embargo, con multihilos, puede estar continuamente capturando imágenes, procesándolas al mismo tiempo y continuar con la captura sin que nada bloquee al programa para seguir su ejecución. Este caso ejemplifica la ventaja del uso de multihilos. Además, aplicando paralelismo (que los hilos se ejecuten en diferentes *cores*) puede ayudar a reducir estos tiempos de ejecución.

### 13.3. 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 51.

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 sobre ella).

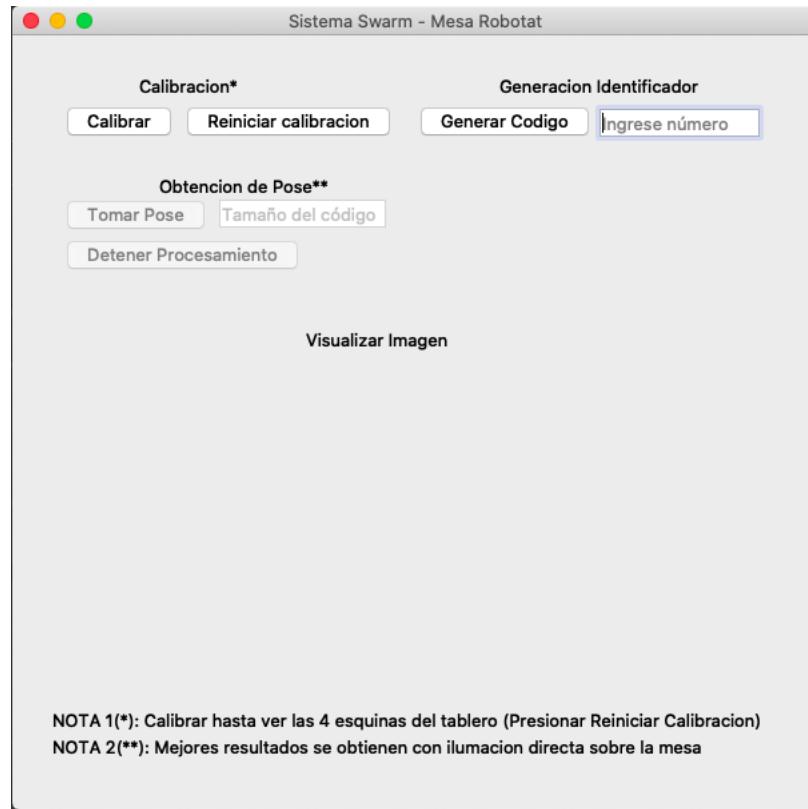


Figura 51: Primer prototipo de interfaz de usuario.

# CAPÍTULO 14

---

## Pruebas preliminares con Matlab

---

Además de las pruebas mencionadas en los lenguajes de Python y C++, también se iniciaron algunas pruebas con Matlab. El objetivo principal de este capítulo es presentar esas primeras pruebas realizadas en Matlab como primeros pasos para la migración de la herramienta hacia este lenguaje. Esto, además de los lenguajes utilizados en este trabajo, brindará mucha más versatilidad al usuario ya que ofrece una tercera opción para trabajar y utilizar esta herramienta.

Dentro de las pruebas realizadas se incluye captura de imágenes mediante el uso de la cámara web, detección de contornos utilizando el detector de bordes de Canny y la ubicación de la posición de dichos contornos identificados.

### 14.1. Captura de imágenes usando Matlab

Como se observa en la Figura 52, uno de los primeros pasos es realizar capturas utilizando la cámara web. Para esto, se utiliza el paquete de Matlab para cámaras web. Este permite detectar las cámaras que tiene conectada la computadora, así como tomar fotografías mediante el uso de la cámara que se deseé.

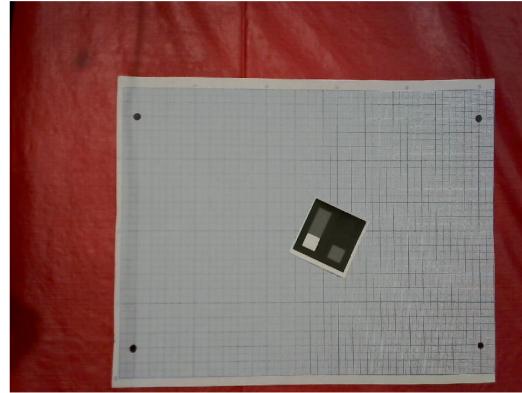


Figura 52: Imagen capturada usando Matlab y la cámara web.

## 14.2. Corrección de perspectiva previo a la calibración

Al momento de calibrar la cámara se requiere tener dos pasos importantes: el primero consiste en corregir la perspectiva de la imagen, es decir, alinear la imagen con respecto a ciertos puntos de interés. Esto significa que si la imagen esta rotada o tiene cierto ángulo de perspectiva, este se pueda corregir para que este alineada a ciertos puntos, en este caso, los cuatro puntos marcados en la esquina de la mesa de pruebas. Finalmente corregida esa perspectiva, se procede a recortar la imagen en el área de interés de trabajo. Sin embargo, este último paso aún falta en esta parte utilizando Matlab. Aún así, se logró corregir la perspectiva exitosamente como lo muestra la Figura 53. En dicha figura se puede observar como en la imagen de la derecha, el cuadro verde no logra alinearse con los puntos de interés y existe cierta rotación. Ya en la imagen de la izquierda, el cuadro verde y los cuatro puntos están mejor alineados y la mesa de pruebas esta paralela al cuadro verde, lo cual muestra que se corrigió la perspectiva de la foto.

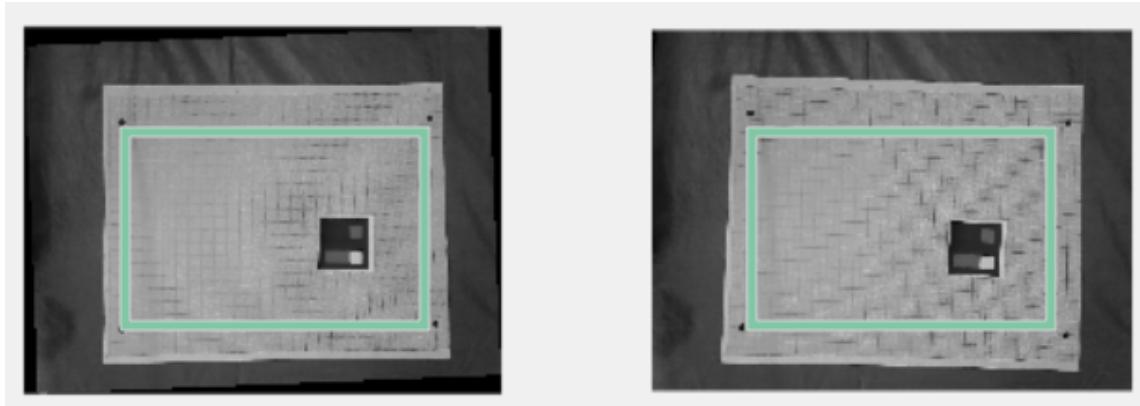


Figura 53: Corrección de perspectiva de la imagen usando Matlab.

### 14.3. Generación de los marcadores

Otro importante paso es la realización de los marcadores. Como en la versión de Python, se realizó una migración desde C++ hacia Matlab. El resultado se muestra en la Figura 54. Como se puede observar, la generación de marcadores es correcta e igual tanto para C++, Python y Matlab.

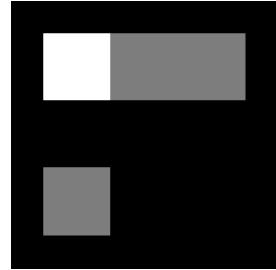


Figura 54: Ejemplo de marcador generado usando Matlab.

### 14.4. Detección de contornos en Matlab

Una de las primeras pruebas fue la detección de los contornos para la identificación de los marcadores. Como se ha mencionado, el proceso general es utilizar el filtro de Canny para detectar los bordes de interés. Luego, usando esos bordes, encontrar aquellos que representen un marcador en la mesa, sacar su posición y ángulo de rotación.

La Figura 55 muestra la imagen que se utilizó para estas pruebas. De momento, para estas pruebas, la imagen ya está calibrada ya que aún no hay una implementación de calibración en Matlab. La Figura 56 muestra la imagen luego de pasar el detector de bordes de Canny sobre la imagen. Finalmente, las figuras 57 y 58 muestran un caso de identificación de uno de los contornos que el programa encontró y su respectivo centro.

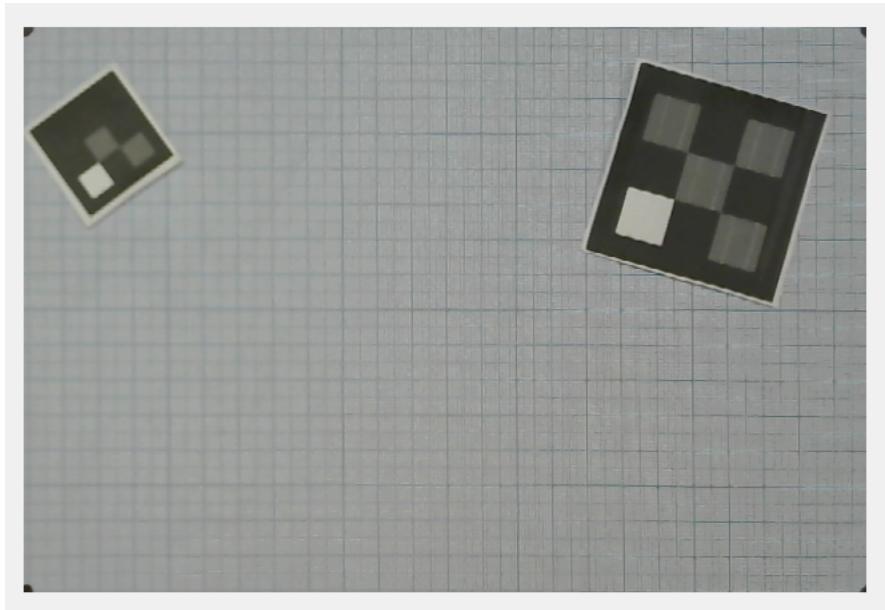


Figura 55: Imagen para pruebas en Matlab.

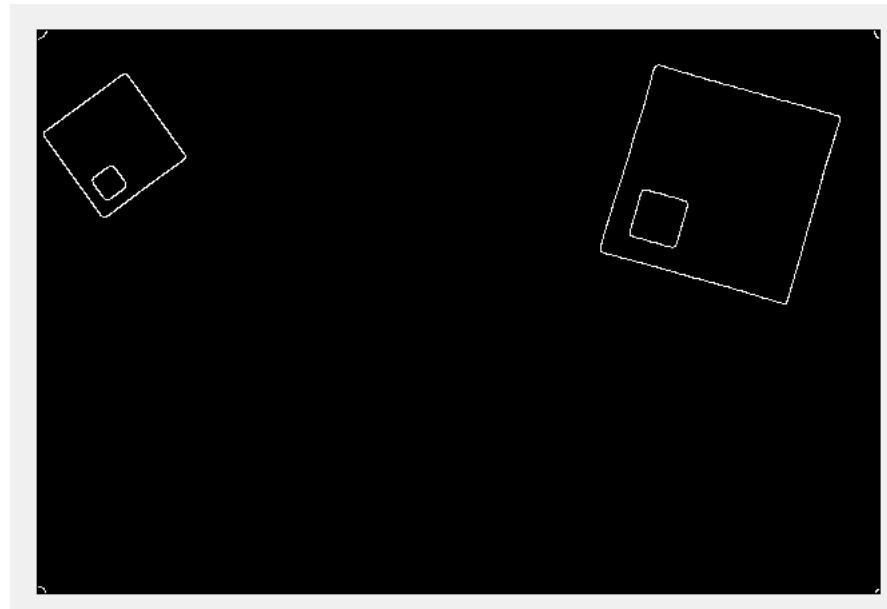


Figura 56: Primeras pruebas de detección de contornos usando Canny en Matlab.

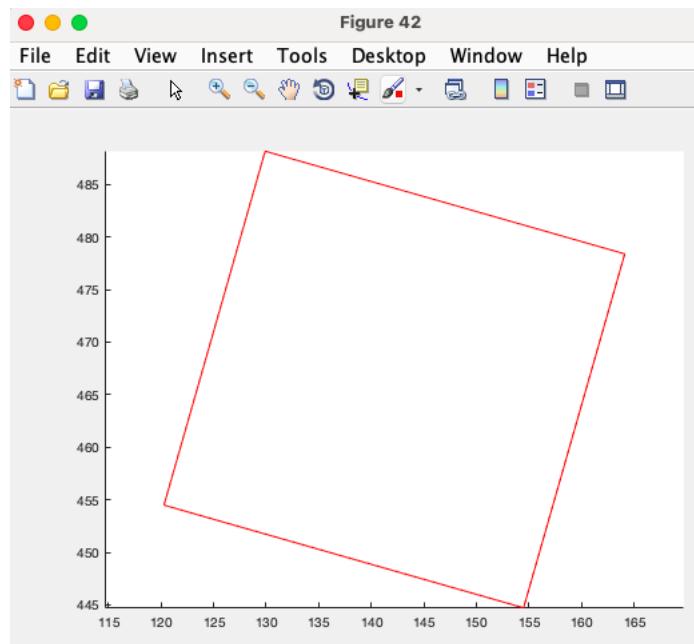


Figura 57: Prueba de detección de contornos en Matlab.

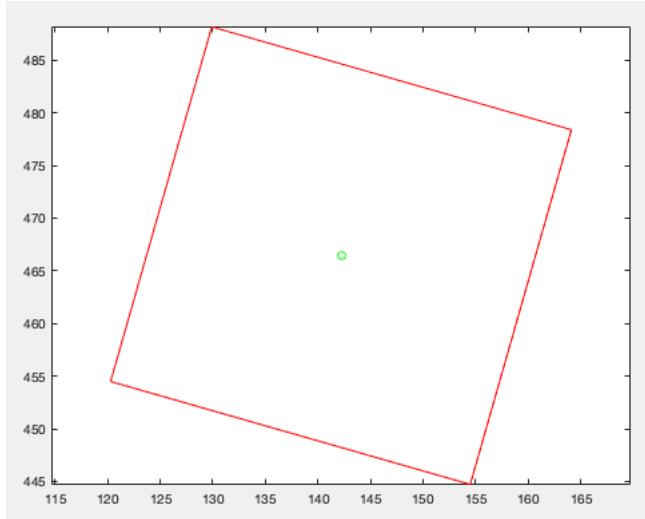


Figura 58: Prueba de detección de centro del contorno en Matlab.

Ahora, las figuras 59 y 60 muestra como el programa es capaz de colocar un rectángulo alrededor de los contornos identificados en la Figura 56. Como se puede observar, también es capaz de detectar los centros de dichos contornos (los círculos verdes marcados). El objetivo de esta función utilizada es buscar los rectángulos más pequeños que encierran los contornos que se identifiquen mediante el uso de Canny. Una vez detectados, se procede a encontrar la posición en X e Y de estos rectángulos (mediante el centro de estos cuadrados o rectángulos). Esto sirve para ubicar en la mesa la posición de los robots. Esta función sería una análoga a `minAreaRect()` de OpenCV en C++ y Python

El motivo por el cual las figuras 59 y 60 parecen estar rotadas con respecto a la Figura 56 es debido a los ejes. Como se mencionó, el eje Y positivo se encuentra en la esquina superior izquierda hacia abajo, mientras que la herramienta de graficación de Matlab lo toma hacia arriba. Esto da la impresión de que los objetos estuvieran espejeados con respecto al original pero solamente es por los ejes de referencia. Aún así, la detección de la posición es correcta.

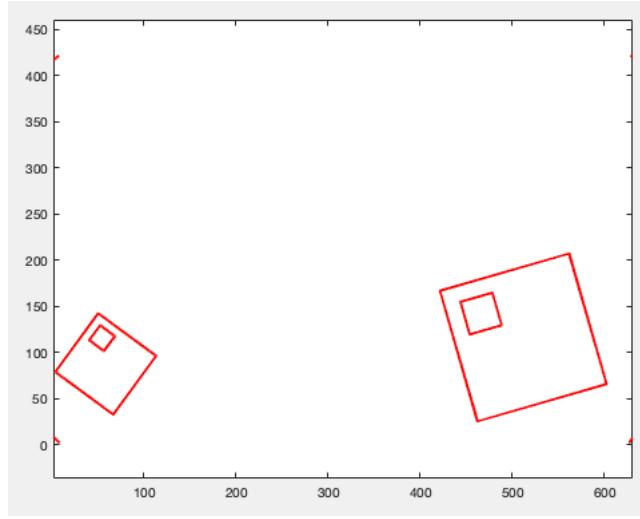


Figura 59: Ubicación de los contornos utilizando .

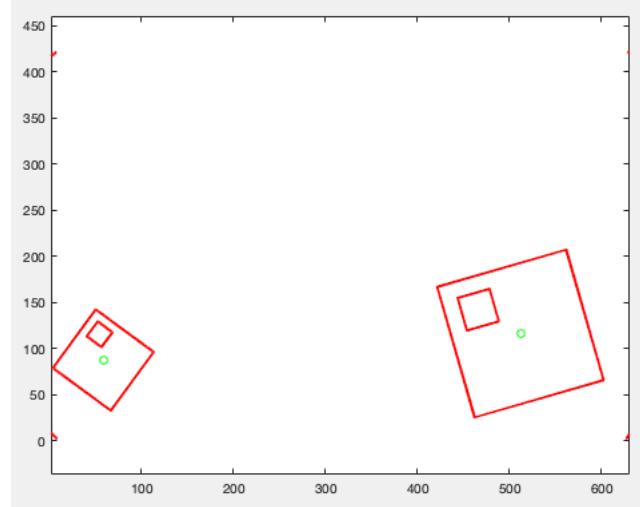


Figura 60: Prueba de detección de centro del contorno en Matlab.

# CAPÍTULO 15

---

## Conclusiones

---

1. Trabajar con una resolución de  $920 \times 760$  permite identificar códigos de un tamaño mínimo de  $3 \times 3$  cm hasta 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 físicas reales y de hasta 0.19 centímetros con la implementación en C++ y las medidas reales. 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, en el lenguaje Python, es mayor que su versión sin hilos. Esto puede deberse a la cantidad de hilos implementados y a otros factores como que estos se estén procesando en un solo *core* o el *overhead* que puede existir entre hilos.

# CAPÍTULO 16

---

## Recomendaciones

---

1. Es posible identificar correctamente los marcadores en condiciones 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.
2. 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.
3. Es posible utilizar verdadera paralelización 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 los programas.
4. 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.
5. El uso de **Matlab** puede ser una opción para explorar como otra alternativa para la implementación de esta herramienta de *software*. Esto brindará mayor versatilidad al usuario según sea la aplicación en la que desee utilizar esta herramienta.

---

## Bibliografía

---

1. Canalís, L. A. *Swarm Intelligence in Computer Vision: an Application to Object Tracking* (Universidad de las Palmas de Gran Canaria, mar. de 2010).
2. Lizarazo Zambrano, J. A. y Ramos Velandia, M. A. *Visión artificial y comunicación en robots cooperativos omnidireccionales* (Universidad Central, Facultad de Ingeniería y Ciencias Básicas, ene. de 2016).
3. Inoue, H. y Nakatani, T. Performance of multi-process and multi-thread processing on multi-core SMT processors, 1-10 (2010).
4. Felten, E. W. y McNamee, D. J. *Improving the performance of message-passing applications by multithreading* (Citeseer, 1992).
5. Rodas Hernández, A. J. *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. Briega, R. L. *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. Hernández, J. C., Rodríguez, J. y Santiago, M. F. *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. Pizarro, D. A., Campos, P. y Tozzi, C. L. *Comparación de técnicas de calibración de cámaras digitales*. *Universidad Tarapacá* **13**, 58-67 (2005).
11. Ricolfe Viala, C. y Sánchez Salmerón, A. J. *Procedimiento completo para el calibrado de cámaras utilizando una plantilla plana*. *Revista Iberoamericana de Automática e Información Industrial* **5**, 93-101 (2008).

12. Lanzaro, F. y Fuentes, M. *Calibración y Posicionamiento 3D* <https://iie.fing.edu.uy/investigacion/grupos/gti/timag/trabajos/2014/candombe/calibracion.html>, visitado el 14/09/2020.
13. Cabrera Quirós, L., Campos Gómez, R. y Castro Godínez, J. *Pasos críticos en la estimación de pose en cámara: una evaluación usando la biblioteca LTI-LIB2*, 60-69 (sep. de 2013).
14. Coronado, E. *Patrones de calibración para OpenCV* <https://mecatronicauaslp.wordpress.com/2014/03/08/patrones-de-calibracion-para-opencv/>, visitado el 14/09/2020.
15. Carver, R. H. y Tai, K.-C. *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs* (John Wiley & Sons, 2005).
16. Yong, P. y Ho, E. T. W. *Streaming brain and physiological signal acquisition system for IoT neuroscience application*, 752-757 (dic. de 2016).
17. Chawla, P. *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. Doherty, E. *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=Cj0KCQjwoPL2BRDxARIaEMm9y\\_jZ7Mu3oH1wRJc5uHdWIeMdhGHbLeR22kkNxokV1-YZS-isYFjZKIArzfEALw\\_wcB](https://www.educative.io/blog/object-oriented-programming?aid=5082902844932096&utm_source=google&utm_medium=cpc&utm_campaign=blog-dynamic&gclid=Cj0KCQjwoPL2BRDxARIaEMm9y_jZ7Mu3oH1wRJc5uHdWIeMdhGHbLeR22kkNxokV1-YZS-isYFjZKIArzfEALw_wcB), visitado el 05/06/2020.
20. Alarcón, J. M. *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

---

## Glosario

---

**C++** : Es un lenguaje de programación diseñado en 1979 por Bjarne Stroustrup. La intención de su creación fue extender mecanismos que permiten la manipulación de objetos. [27, 42]

**Canny** : Algoritmo de Canny es un operador desarrollado por John F. Canny en 1986 que utiliza un algoritmo de múltiples etapas para detectar una amplia gama de bordes en imágenes. [40]

**marcador** : Identificador visual que permite colocarle un ID a cada robot en la mesa. [45]

**marcadores** : Identificador visual que permite colocarle un ID a cada robot en la mesa. [27, 30]

**multihilos** : En sistemas operativos, un hilo (del inglés **thread**), es un proceso subproceso una secuencia de tareas encadenadas muy pequeña que puede ser ejecutada por un sistema operativo. Los multihilos, por tanto, son una combinación de varios hilos. [62]

**Python** : Es un lenguaje de scripting independiente de plataforma y orientado a objetos, preparado para realizar cualquier tipo de programa. [35, 37, 40, 42]