UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Herramienta de Software de Visión por Computadora para Aplicaciones de Robótica de Enjambre en una Mesa de Prueba - Fase III

Trabajo de graduación presentado por José Ignacio Ramírez Soto para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2022

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

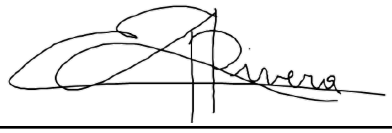
Herramienta de Software de Visión por Computadora para Aplicaciones de Robótica de Enjambre en una Mesa de Prueba - Fase III

Trabajo de graduación presentado por José Ignacio Ramírez Soto para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

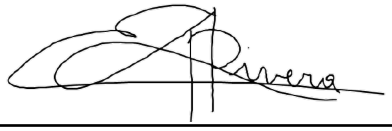
2022

Vo.Bo.:

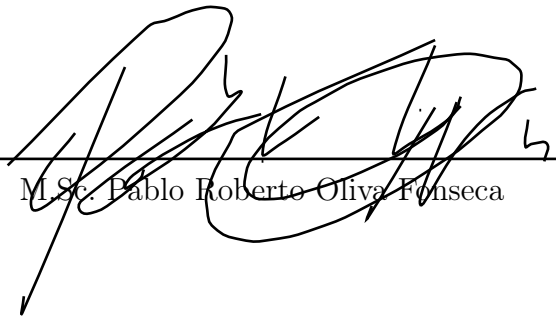
(f) 

Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) 

Dr. Luis Alberto Rivera Estrada

(f) 

M.Sc. Pablo Roberto Oliva Fonseca

(f) 

M.Sc. Pedro Iván Castillo Rivera

Fecha de aprobación: Guatemala, 05 de enero de 2022.

Prefacio

El interés en este trabajo surge en la visión por computadora y explorar las diferentes aplicaciones que se le pueden dar. Con este trabajo y sus resultados, espero pode brindar una herramienta flexible y de fácil uso a los investigadores en el área de robótica de enjambre para desarrollas sus trabajos.

Veo oportuno dar agradecimiento a las personas que me han apoyado a lo largo de este trabajo sin los cuales no podría haber llegado tan lejos. En primera instancia, a mi familia (principalmente a mi madre Patricia Ramírez) ya que siempre buscaron lo mejor para mí. A la Fundación de la Universidad del Valle de Guatemala por brindarme de esta gran oportunidad. A mi asesor, Luis Rivera, por su apoyo y consejos durante todo el desarrollo de este trabajo; así mismo a Miguel Zea por brindar apoyo en su tiempo disponible aún sin ser mi asesor. A mis amigos dentro y fuera de la carrera ya que fueron de gran apoyo en mi vida. Y finalmente a ciertas personas que me han acompañado desde mucho tiempo y que, aunque no lo sepan, me han ayudado a volver a sonreír cuando pensé que no podía, estas son Alejandro Fernández, Dorothy Clark y Sabrina Carpenter. Gracias a todos por tanto.

Índice

Lista de figuras

Lista de cuadros

Resumen

En este trabajo, se realizó una continuación a la herramienta de visión por computadora desarrollada en la UVG. Esto significó agregarle nuevas funcionalidades como detección de obstáculos, transmisión de datos entre dispositivos, entre otros. Esta herramienta se migró de lenguaje a Matlab. Debido a que la fase anterior no se logró validar en la mesa de pruebas ubicada en la UVG, se continuó con este paso y se verificaron ambas herramientas en dicha mesa. Se buscó las condiciones óptimas para estas pruebas y que fueran similares entre sí.

El objetivo principal de esta herramienta es permitir al usuario reconocer la pose de los agentes (o robots) en un área de trabajo y los obstáculos dentro de la misma. Todo esto es posible hacerlo con robots en movimiento con tasas de refresco moderadas. Lo cual se validó al ser utilizada en aplicaciones simples de robótica de enjambre bajo condiciones controladas. También se diseñó una interfaz gráfica que facilita al usuario el uso de la herramienta. Y se realizó una exploración de la factibilidad de realizar paralelización en los códigos para mejorar el rendimiento.

Abstract

In this work, a continuation of the computer vision tool developed at UVG was carried out. This meant adding new features such as obstacle detection, data transmission between devices, among others. This tool was migrated from language to Matlab. Given that the previous phase couldn't validate in the testing table located at UVG, this step was continued and both tools were verified on said table. The optimal conditions were pursued for these tests and that they were like each other.

The main objective of this tool is to allow the user to know the pose of the agents (or robots) in a given working area and the obstacles in it. All of this is possible to do with robots in motion with moderate refresh rates. Which was validated when being used in simple swarm robotics applications under controlled conditions. A graphic interface was also designed that helps the user the use of the tool. And there was an exploration of the feasibility of performing parallelization in the codes to improve performance.

CAPÍTULO 1

Introducción

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

Tomando en cuenta lo anterior y el hecho de que la Universidad del Valle de Guatemala ha desarrollado una herramienta que cumple exactamente esto, se consideró expandir el alcance de la misma a otro lenguaje más. Durante esta transición a otro lenguaje, se puede agregar funcionalidades que sean de beneficio al usuario.

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 y una breve porción de teoría que facilitará al lector a comprender ciertos conceptos claves de este trabajo.

La metodología para este trabajo constituyó en una investigación previa para entender el funcionamiento de la herramienta existen en Python y C++, tanto sintaxis del lenguaje 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 continuó buscando funciones que realizaran las mismos operaciones o similares en el lenguaje de Matlab. Estas se implementaron en el código de Matlab con una interfaz gráfica. Por último, se plantearon pruebas que se consideraron óptimas para verificar el correcto funcionamiento de la herramienta.

Luego de describir la metodología, se presentan los resultados obtenidos de las pruebas, validando así el alcance que se buscaba obtener en esta tesis; esto tanto de la nueva herramienta de Matlab como de la existente en Python. Finalmente, se presentan las conclusiones de este trabajo y se dan recomendaciones para futuras aplicaciones.

CAPÍTULO 2

Antecedentes

2.1. Visión por computadora aplicada en robótica de enjambre

El objetivo de la visión por computadora es interpretar imágenes; un gran campo es la detección de objetos y movimientos. La variedad de las aplicaciones en las que la visión por computadora abarca desde cosas simples como detección de grietas en puentes [1] hasta cosas más complejas como detección de actos de violencia [2]. Una aplicación no tan explorada es la detección de robots para uso en robótica de enjambre.

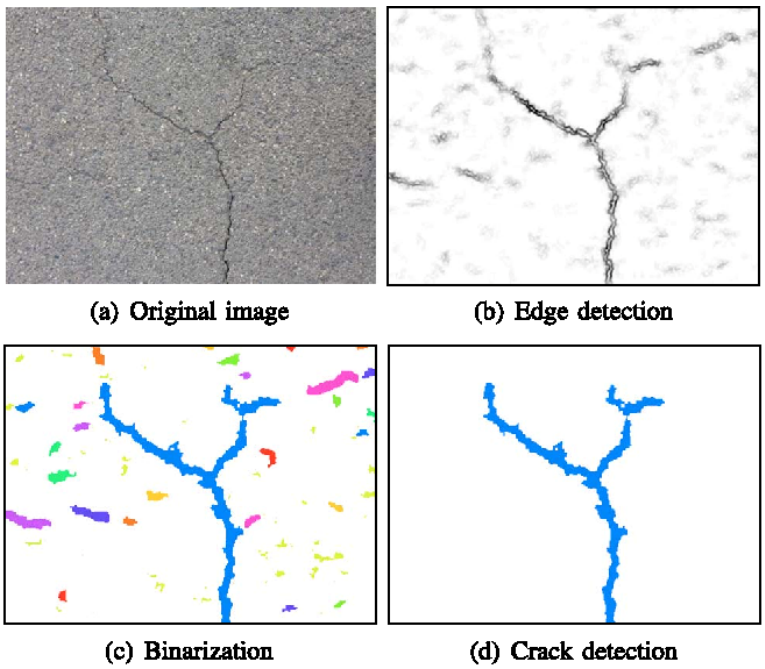


Figura 1: Ejemplo de proceso de detección de grietas [1].

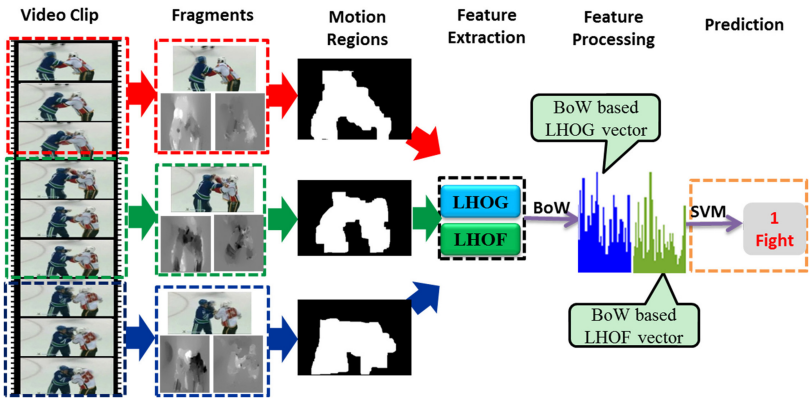


Figura 2: Ejemplo de proceso de detección de violencia [2].

En el artículo *Visión artificial y comunicación en robots cooperativos omnidireccionales* [3] se detalla el proceso de diseño tanto del robot en sí, del programa de comunicación y el software para la detección del mismo y la abstracción de su posición. En este se utiliza OpenCV junto con Python, aunque se menciona que existen versiones de OpenCV compatibles con otros lenguajes como C, C++, Java y Matlab. Esta es una librería gratuita para lograr de manera sencilla el procesamiento de imágenes en tiempo real.

2.2. Continuación Fases I y II

El presente trabajo de graduación es la continuación a los trabajos de André Rodas titulado "Desarrollo e implementación de algoritmo de visión por computadora en una mesa de pruebas para la experimentación con micro-robots móviles en robótica deenjambre" [4] y José Guerra titulado "Algoritmos de Visión por Computadora para el Reconocimiento de la Pose de Agentes Empleando Programación Orientada a Objetos y Multihilos" [5].

En la primera fase se diseñó y fabricó una mesa de pruebas de 130×90 *cm* que cuenta con tubos de acero para ajustar la altura de montura para una cámara de vídeo (mostrada en la Figura 3). Esta mesa es completamente blanca para crear mayor contraste y facilitar el procesamiento de imágenes. También se posicionaron círculos verdes de 1.6 *cm* de diámetro en las esquinas para poder ser reconocidos en la calibración. Se hicieron pruebas utilizando esta mesa para elegir el modelo de identificación más óptimo (tomando en consideración la facilidad de procesamiento, creación entre otros factores).

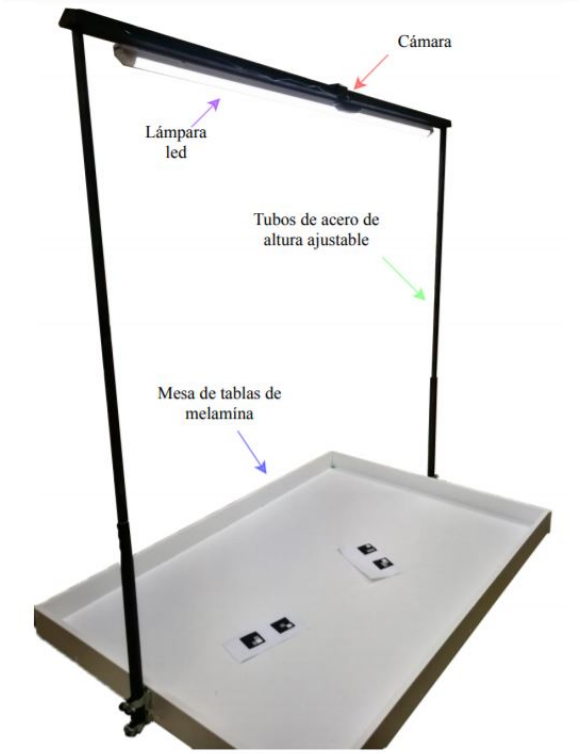


Figura 3: Mesa de prueba Robotat utilizada anteriormente [4].

En temas de software, se realizaron tres programas: generador de códigos para la identificación de cada robot en la imagen, calibrador de cámara que detecta las esquinas descritas anteriormente para alinear y cortar la imagen, y el programa para obtener la pose de los robots que obtiene las posiciones x, y y ángulo de rotación θ y almacena en una base de datos. Todos estos se desarrollaron en el lenguaje C++ y con su interfaz gráfica respectiva.

En la segunda fase, se migraron los programas al lenguaje de Python. Se mantuvieron los pseudocódigos y la idea general teniendo los mismos identificadores y programas, pero se modificaron ciertos algoritmos para hacer más precisos los resultados. Debido a la pandemia que inició el 2020, no se logró utilizar la misma mesa para verificar los programas que en la primera fase. Por lo mismo, se realizó una nueva mesa con dimensiones mucho menores (28×14 cm). Se llevaron a cabo pruebas de ambos algoritmos con esta nueva mesa para tener parámetros de rendimiento comparables entre sí. Se terminó concluyendo que ambos programas tienen rendimientos muy similares.

Se implementó programación en multihilos para intentar mejorar el rendimiento de los programas, pero este programa resultó tener menor rendimiento en cuanto a tiempo de ejecución. Se terminó concluyendo que fue ya que los hilos se ejecutaban en el mismo núcleo del procesador o simplemente que el costo de creación, ejecución y destrucción de los hilos era menor a los beneficios (también conocido como *overhead*). Y, por último, se realizaron pruebas preliminares en Matlab logrando detección básica de bordes.

2.2.1. Limitaciones

A pesar de todo lo logrado anteriormente, existen ciertas limitaciones tanto propias de la visión por computadora como limitaciones específicas de las fases que no se pudieron sobrevenir en su momento. De las primeras se pueden mencionar cosas como problemas de detección de objetos con bajo contraste o iluminación, limitaciones de resolución de imagen, resolución de medidas, entre otras. Por otro lado, específicamente de los trabajos anteriores, se tienen las siguientes:

1. Como se mencionó al final de la sección anterior, los resultados de la implementación con multihilos no mostró resultados satisfactorios en términos de tiempo de ejecución.
2. Al tener una calibración automática, si existen formas similares a los marcadores circulares fuera del área, esto puede causar un conflicto en la calibración.
3. Únicamente se realizaron pruebas preliminares en Matlab, pero no se llegó a un producto final.
4. Únicamente se detectan y procesan las poses de los robots, pero no de otros objetos que puedan obstruir su movimiento.
5. De nuevo, no se logró verificar los códigos de la segunda Fase en una mesa de prueba amplia como la que se encuentra en la universidad.

CAPÍTULO 3

Justificación

En los principios de la robótica, estos funcionaban sin ningún tipo de retroalimentación para realizar correcciones a sus acciones. Pronto se descubrió de la necesidad de los sensores para medir el estado actual de los actuadores del robot. Esta información luego es procesada para realizar correcciones y obtener un comportamiento más apegado a lo deseado. A esto se le conoce como sistemas de control. En los robots más sencillos que emplean estas técnicas, se utilizan sensores que únicamente dan información del estado actual del robot. Algunos sensores de esta índole según [6] son:

- Un Sistema de Posicionamiento Global (GPS) para la medición de posición global
- Una Unidad de Medición Inercial (IMU) para la medición tanto de velocidades y aceleraciones en los tres ejes como de la orientación del mismo respecto a un marco de referencia inicial.
- Un codificador rotatorio para la medición de posición angular de juntas revolutas en caso de poseer.
- Un sensor de distancia o proximidad para la medición de posición lineal de juntas prismáticas en caso de poseer.

En el capítulo 5 de [7] se mencionan más sensores y se detalla más a profundidad las aplicaciones y la manera de funcionar de los mismos. Todos los sensores mencionados ayudan a conocer información del robot en sí, pero esto nos deja sin información externa; es decir, no sabemos qué es lo que rodea al robot. Una manera sencilla tanto en implementación física como de programación es utilizar sensores de distancia para asegurarse que el robot no colisione con algún otro objeto. Esto le otorga un nivel básico de visión al robot. Pero cuando el área superficial del robot o el área de trabajo es mucho más compleja y amplia, esta solución por sí sola no es suficiente. En la robótica de enjambre, el problema es que se tendría que implementar esto en cada robot. Tomando en cuenta que los enjambres de robots pueden llegar a ser de cientos o miles de agentes hace que se eleve el costo de la operación.

Aquí es donde entra la importancia de la visión por computadora. Esto nos facilitaría la información del entorno y de los agentes como tal. Por lo que no se necesitaría de ningún otro sensor más que la cámara para obtener toda la información necesaria de todos y cada uno de los agentes, siempre que la cámara no se encuentre obstaculizada.

En las fases anteriores se han logrado avances sustanciales que hacen de la herramienta existente una posible solución a este problema. Sin embargo, como se describió en la sección de antecedentes, esta cuenta con una gran cantidad de limitaciones. Estas limitaciones hacen que, aunque sea una posible solución, no se pueda asegurar la funcionalidad deseada. Esto ya que aún no se han realizado pruebas en un área de trabajo suficientemente amplia para albergar múltiples robots. Estas pruebas han sido con los identificadores estáticos, no en movimiento. La implementación en uno de los lenguajes de programación debe de ser terminada. La implementación de programación multihilo no resultó en una reducción de tiempo, por lo que este aún debe ser optimizado para tener una tasa de refresco más alta y poder hacer las mediciones lo más cercano posible a tiempo real. Y que el problema de identificar el entorno aún no ha sido siquiera tratado.

Sin muchas de estas mejoras, la herramienta no puede ser utilizada con el propósito principal por el cual se inició este proyecto. Por lo que la importancia de este trabajo de graduación es mejorar estos algoritmos para ayudar a las aplicaciones de robótica de enjambre que se desarrollara en la Universidad del Valle de Guatemala.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Agregar funcionalidad a la herramienta de visión por computadora desarrollada en fases previas, adaptarla a la mesa de prueba Robotat de la UVG, y validar el sistema completo en aplicaciones simples de robótica de enjambre.

4.2. Objetivos específicos

- Migrar algoritmos de visión por computadora que se han desarrollado en fases previas en los lenguajes de programación Python y C++, a Matlab.
- Desarrollar algoritmos para el mapeo de la superficie efectiva de la mesa de prueba, y la detección de obstáculos.
- Mejorar los algoritmos de detección de pose, para poder aplicarlos a robots en movimiento.
- Explorar la factibilidad de paralelizar los algoritmos desarrollados para mejorar el rendimiento de los mismos.
- Validar la herramienta de software mejorada en la mesa de prueba Robotat.

CAPÍTULO 5

Alcance

En esta tesis se utilizaron los programas desarrollados con anterioridad por André Rodas y José Pablo Guerra en C++ y Python respectivamente. Un resultado de este trabajo es continuar con este último y realizar la verificación del mismo en la mesa de pruebas ubicada en la UVG ya que esto no se logró realizar en su fase. El siguiente resultado fue la adaptación de estos dos algoritmos a un tercer lenguaje, siendo Matlab el elegido para dar mayor flexibilidad al usuario de elegir el lenguaje de desarrollo.

Dicho programa permite al usuario conocer las posiciones y orientaciones de los agentes ubicados en un área de trabajo junto con cualquier obstáculo en el mismo. Se incluye la posibilidad de calibrar la cámara para únicamente tener en imagen el área de interés marcada por círculos en las esquinas. Además, se permite al usuario la toma de imágenes para procesar e identificar los robots de manera continua, efectivamente haciéndose en tiempo real. Y también se provee una manera de generar las imágenes de los códigos con el número de identificación que se necesite.

Se realizaron pruebas para validar estas herramientas variando diferentes parámetros como las posiciones, orientaciones y tamaños de los robots, tamaño del área de trabajo y condiciones de iluminación. De esto se pudo determinar las condiciones ideales de iluminación y verificar que la herramienta realizara la detección de la manera correcta.

El alcance de este trabajo se vio limitado por la pandemia aún en curso del Covid-19, ya que se tuvo un tiempo limitado con la mesa para perfeccionar la herramienta y realizar todas las pruebas deseadas. Sin embargo, se realizaron las pruebas necesarias para hacer una validación correcta de la herramienta.

CAPÍTULO 6

Marco teórico

6.1. Robótica de enjambre

La robótica de enjambre es la implementación en robots de algoritmos de inteligencia de enjambre. La característica principal de estos robots es su bajo costo, y por ende, la gran escalabilidad de la población. Esto junto con la premisa de la inteligencia de enjambre para realizar tareas complejas utilizando un conjunto de robots simples da lugar a una gran cantidad de aplicaciones relevantes. En el artículo *Reasearch Advance in Swarm Robotics* [8] se detallan las inspiraciones de distintos algoritmos del campo como pájaros, hormigas, peces, etcétera. También se mencionan varias aplicaciones como ayuda post-desastre u otras aplicaciones peligrosas para humanos, aplicaciones militares, aplicaciones de mapeo de terrenos, limpieza de fugas de aceite, entre otros.

Junto con todos estos algoritmos de inteligencia de enjambre, se han diseñado diversos prototipos de robots para implementar estos en el mundo físico. Existe una gran cantidad de equipos que han dado sus propuestas para este tipo de robots. El prototipo más conocido son los *kilobots*, estos son robots auto-organizantes desarrollados en los laboratorios de *Harvard School of Engineering and Applied Sciences (SEAS)*. Estos basan su movimiento en dos motores vibratorios que le permiten deslizarse a través de superficies. Cuentan con transmisores y receptores infrarrojos que le permiten comunicarse con los robots vecinos y medir proximidad, pero no de todo el enjambre. El algoritmo de estos robots les permite recrear figuras con ellos mismos a gran escala. Algunas de las desventajas de estos es que los tiempos para que las figuras se completen son muy extensos y las baterías de los mismos duran poco [9].

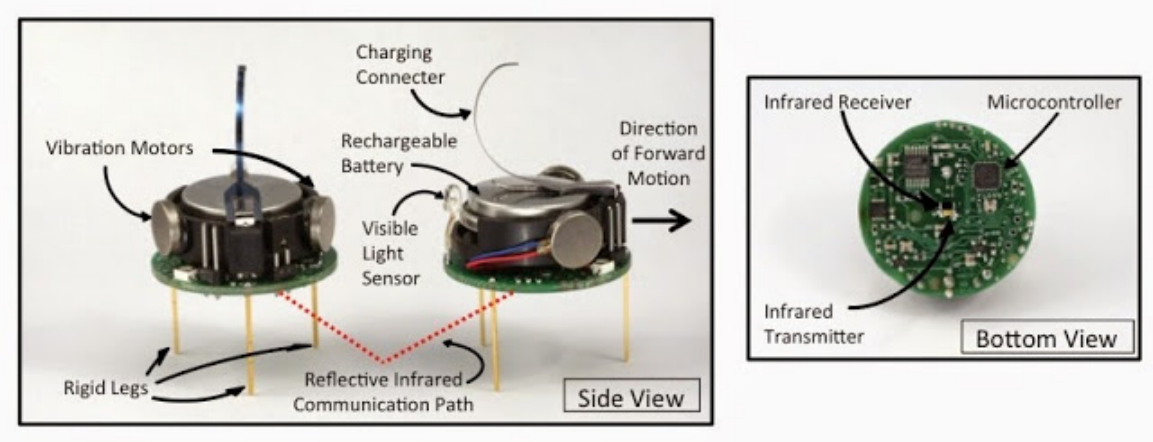


Figura 4: Detalle de Kilobots [10].

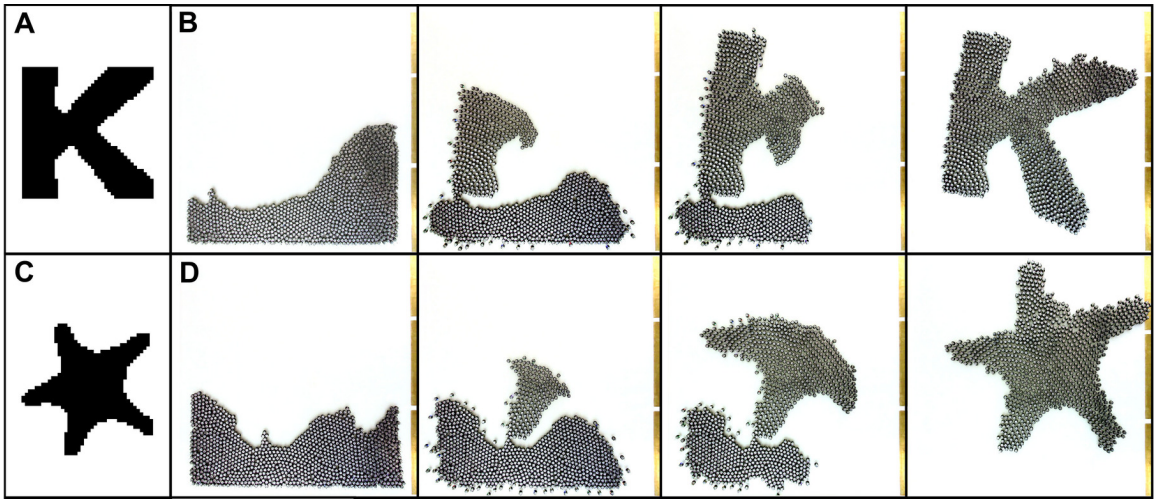


Figura 5: Conjunto de kilobots formando figuras [11].

Otros prototipos mucho más simplistas y recientes son *Smarticles*. La única habilidad que poseen es mover sus aletas. Esto no le permite hacer tareas tan complejas como al resto de robots, pero al juntar varios de estos, se puede hacer que muevan el recipiente en el que se encuentran [12].

Figura 6: Conjunto de Smarticles [13].

6.2. Visión por computadora

Casi todas las especies de animales utilizan ojos los cuales han sido inventados múltiples veces, de distintas formas y funcionando bajo distintos principios. La mayoría de vertebrados tienen dos ojos, pero existen animales como las arañas o pectínidos cuentan con varios. Aún animales simples como las abejas, cuyos cerebros se componen de solo 10^6 neuronas (comparado a nuestras 10^11) son capaces de realizar tareas complejas y críticas para la supervivencia como buscar comida y regresar a la colmena usando su visión.

Nuestra propia experiencia es que los ojos son sensores muy efectivos para reconocimiento, navegación, evasión de obstáculos y manipulación. Las cámaras imitan la función de un ojo y se desea hacer uso de ellas para crear competencias basadas en visión para robots.

El desarrollo de la tecnología ha hecho posible el uso de cámaras como ojos para los robots. Por gran parte de la historia de la visión por computadora, datando de los 1960s, las cámaras electrónicas eran incómodas, costosas, y el podes computacional no era suficiente. Al día de hoy, cámaras para celulares cuestan unos pocos dólares, y los dispositivos electrónicos están estandarizados con poder computacional paralelo masivo. Nuevos algoritmos, sensores baratos y poder computacional basto hacen la visión un sensor práctico en la actualidad. [14]

La visión por computadora, también llamada visión artificial, consta en métodos de adquirir, procesar, analizar y, en general, comprender una imagen; esta se traduce a información numérica que luego puede ser utilizada según sea la aplicación. En el libro *"Robotics Vision and Control Fundamental Algorithms in MATLAB"* [14] se habla con más profundidad de los fundamentos, desde la luz en escena siendo reflejada y capturada por el lente, convertida a una imagen digital y procesada por varios algorítmos para extraer la información requerida. Un diagrama de los pasos generales se muestra en la Figura 7.

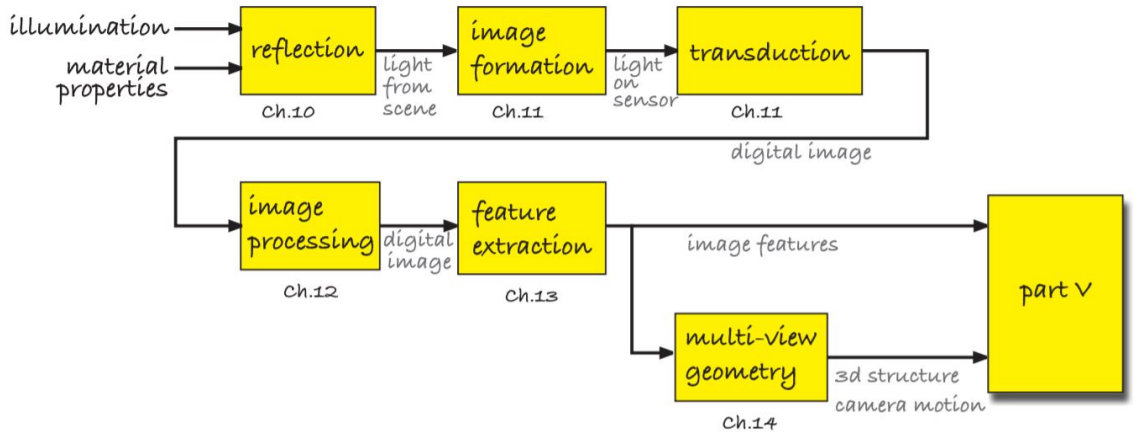


Figura 7: Diagrama de pasos implicados en el procesamiento de imagen [14]

En el libro *Computer Vision: Algorithms and Applications* [15] también se habla a profundidad de la historia, técnicas, algoritmos y aplicaciones de esta disciplina de la ciencia. Cabe mencionar que durante los últimos años, se han hecho grandes avances en esta área gracias a las mejoras en sistemas de aprendizaje artificial (*Deep Learning*, *Machine Learning*, *Big Data*). Una de las principales aplicaciones que se ha visto beneficiada es el reconocimiento facial, pero otras aplicaciones menos comerciales también están siendo mejoradas con algoritmos de aprendizaje novedosos. Entre estas se puede mencionar mapeo de ambientes en 3D y edición de imágenes.

En el campo de la robótica particularmente, usualmente se utiliza para obtener información geométrica del entorno. Por ejemplo, en el caso de un carro autónomo, poder identificar en dónde se encuentran los otros vehículos relativos a él, la ubicación de las líneas de las carreteras, señalizaciones de tránsito, entre otros. En el caso de realizar control sobre un robot, se puede utilizar la visión por computadora como sensores para conseguir las coordenadas de un objeto a ser manipulado.

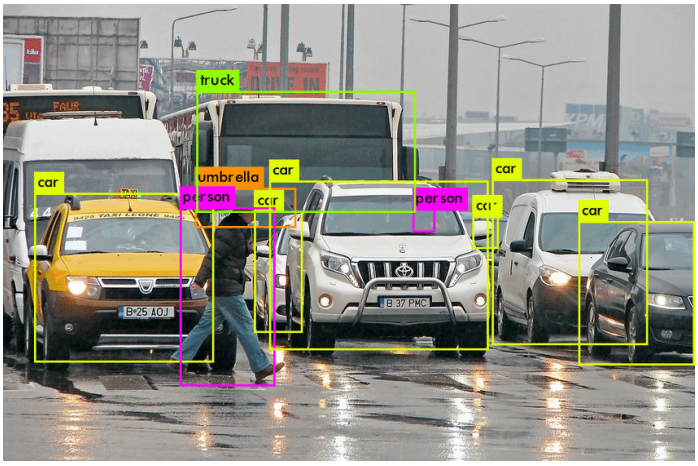


Figura 8: Visión por computadora en vehículos autónomos [16].

6.2.1. Herramientas de visión por computadora

Para un campo tan amplio y llamativo como es la visión por computadora tiene múltiples herramientas que facilitan al programador la implementación del mismo. En este trabajo de tesis se utilizarán OpenCV y Matlab. OpenCV contiene múltiples algoritmos para realizar tareas de procesamiento de imágenes y vídeo. Esta librería es soportada en lenguajes como Python, C, Java, Javascript, Matlab y Octave. También cabe mencionar que es compatible con otros sistemas operativos como Linux y Android. Una de las mayores limitaciones de esta es la escalabilidad, ya que al tener conjuntos de imágenes amplios, este puede ser más lento de lo normal. Además, para solventar este problema, no se cuenta con soporte para GPUs (Tarjetas gráficas dedicadas) sino depende de CUDA para esto. Por lo que se necesita de una tarjeta con esta arquitectura para mejorar el rendimiento.

Matlab por el otro lado, es un programa utilizado para mayor variedad de aplicaciones. Sin embargo, una gran aplicación es el procesamiento de imágenes. Este cuenta con funciones especializadas y optimizadas con este objetivo en mente. Este cuenta con una interfaz y sintaxis fácil e intuitiva por lo que es muy utilizado. Sin embargo, este programa es de pago y puede no ser el más rápido en tiempo de ejecución. Por estas razones, se considera mejor para investigación o prototipos sobre un producto comercial.

Es de notar que estas no son las únicas herramientas existentes. Por ejemplo, existe *AForge.NET/Accord.NET* que es utilizado para el lenguaje *Microsoft .NET*, este es simple e intuitivo y con tiempos de procesamiento bajos, pero no tanto como otras herramientas. *SimpleCV* es un marco de trabajo que cuenta con librerías de visión de computadora integradas (incluida OpenCV) y simplifica la sintaxis y el nivel de conocimiento requerido para la programación. Sin embargo, este únicamente está disponible en Python. *BoofCV* es una reescritura de OpenCV únicamente disponible en Java.

Con las nuevas tecnologías de inteligencia artificial, también existen herramientas como Tensorflow para entrenar modelos de visión por computadora. Aunque estas puede a llegar a ser muy precisas de se entrenadas correctamente, el tiempo de entrenamiento y el equipo requerido no es muy accesible para la mayoría de los desarrolladores.

6.3. Programación Multihilo

Cuando un programa es ejecutado, el sistema operativo crea un proceso que contiene el código y la información del programa manejando el proceso hasta que este termine. En cada proceso, la ejecución del programa implica inicializar y mantener una gran cantidad de información como: el estado del proceso, el contador de instrucción, los valores de los registros, descripciones de archivos, solicitudes de entradas/salidas, entre otros. Existen sistemas operativos que permiten mulitiprocesamiento, pero debido al gran volumen de información por proceso, se hace caro crear y manejar múltiples procesos.

Un hilo es una unidad de control dentro de un proceso. Cada hilo ejecuta una función en el programa. Cada proceso inicia con un hilo principal que ejecuta la función principal del programa. En programas multihilo, el hilo principal crea hilos secundarios que ejecutan otras funciones. Cada hilo tiene sus propios registros de memoria, contador de instrucción y puntero de pila. No obstante, cada hilo de un proceso comparte la información, código, recursos y espacio de dirección del proceso. Toda la información mencionada en el párrafo anterior que describe un proceso también es compartida entre hilos; esto reduce de manera significativa los gastos generales involucrados en crear y manejar hilos.

La programación multihilo puede acelerar el desempeño a través de paralelización. Un programa que hace uso de dos procesadores puede ejecutarse en aproximadamente la mitad del tiempo. Sin embargo, este nivel de agilización usualmente no puede ser obtenido debido a los gastos requeridos para coordinar los hilos. De igual manera, existen otras ventajas como que se tienen menos gastos al hacer comunicación de hilos intraprocesos comparado a comunicación entre procesos. [17]

6.4. Paralelización

Es de notar del párrafo anterior, el hecho que un programa sea multihilo no implica que esté paralelizado en un sentido estricto. Por ejemplo, si se crea una función, al momento de llamarla, se crea un hilo nuevo que se ejecuta mientras el hilo principal espera a que este se termine de ejecutar. Esto implica que lo que en realidad se está explorando durante el trabajo deberá de ser paralelización, ya que lo que se busca es optimizar el algoritmo para que se ejecute en el menor tiempo posible.

La clave de la paralelización es la concurrencia explotable. La concurrencia existe en un problema computacional cuando este puede ser descompuesto en subproblemas que pueden se ejecutados de manera segura al mismo tiempo. La mayoría de problemas computacionales contienen esta concurrencia explotable. El programador trabaja con esta al crear un algoritmo paralelo e implementando el algoritmo usando un entorno de programación paralelo.

Un ejemplo simple que demuestra la esencia de la computación paralela es el siguiente: se necesita realizar la suma de un conjunto de datos de n cantidad de datos. En lugar de realizar la suma de los valores secuencialmente, el conjunto de datos puede ser dividido y realizar las sumas de los subconjuntos de manera paralela. Las sumas parciales luego son combinadas para obtener el resultado final. Esta ejecución en paralelo nos permite obtener el resultado antes.

Este tipo de programación puede presentar desafíos únicos. Con frecuencia, las tareas concurrentes que componen el problema incluyen dependencias que deben ser identificadas y manejadas de manera adecuada. El orden en el que se ejecutan las tareas puede cambiar las respuestas de manera no determinista. Por ejemplo, en el problema anterior, las sumas parciales no pueden ser combinadas entre sí hasta que su propio cálculo termine. El identificar estos problemas y solucionarlos de manera que no afecten el resultado final puede tomar un esfuerzo considerable. Incluso, la efectividad de un algoritmo paralelizado depende de lo bien que este se mapea en la computadora; esto implica que un algoritmo puede ser muy efectivo en una arquitectura de CPU, pero un desastre en otra. [18]

CAPÍTULO 7

Mesas de prueba

En este capítulo se muestran las especificaciones de las diferentes mesas de prueba que se utilizaron para verificar los algoritmos en distintos momentos del desarrollo de este trabajo, tanto dimensiones físicas como materiales y estructura. También se darán los motivos de por qué cada una de las mismas se vio necesaria.

7.1. Mesa de pruebas iniciales

Debido a la pandemia del Covid-19, durante la fase anterior, José Guerra se vio en la necesidad de improvisar una mesa en la cual realizar sus pruebas desde su hogar. Según las especificaciones provistas en su tesis, esta tenía dimensiones de 28×14 cm. Ya que estas eran las condiciones bajo las cuales se desarrollaron y validaron los programas, se intentó hacer una mesa con características similares. Esta consistió simplemente de una hoja tamaño carta (8.5×11 $pulgadas$ o 21.59×27.94 cm) sobre una mesa (ver Figura 3). Esta hoja no contenía ninguna cuadrícula ya que se no consideró necesario para las pruebas. Sobre esta mesa, se posicionaría la cámara con ayuda de una base simple construida con madera como se ve en la Figura 9. Ya que la mesa en sí es móvil, no se consideró en darle medidas exactas a las piezas de la base.

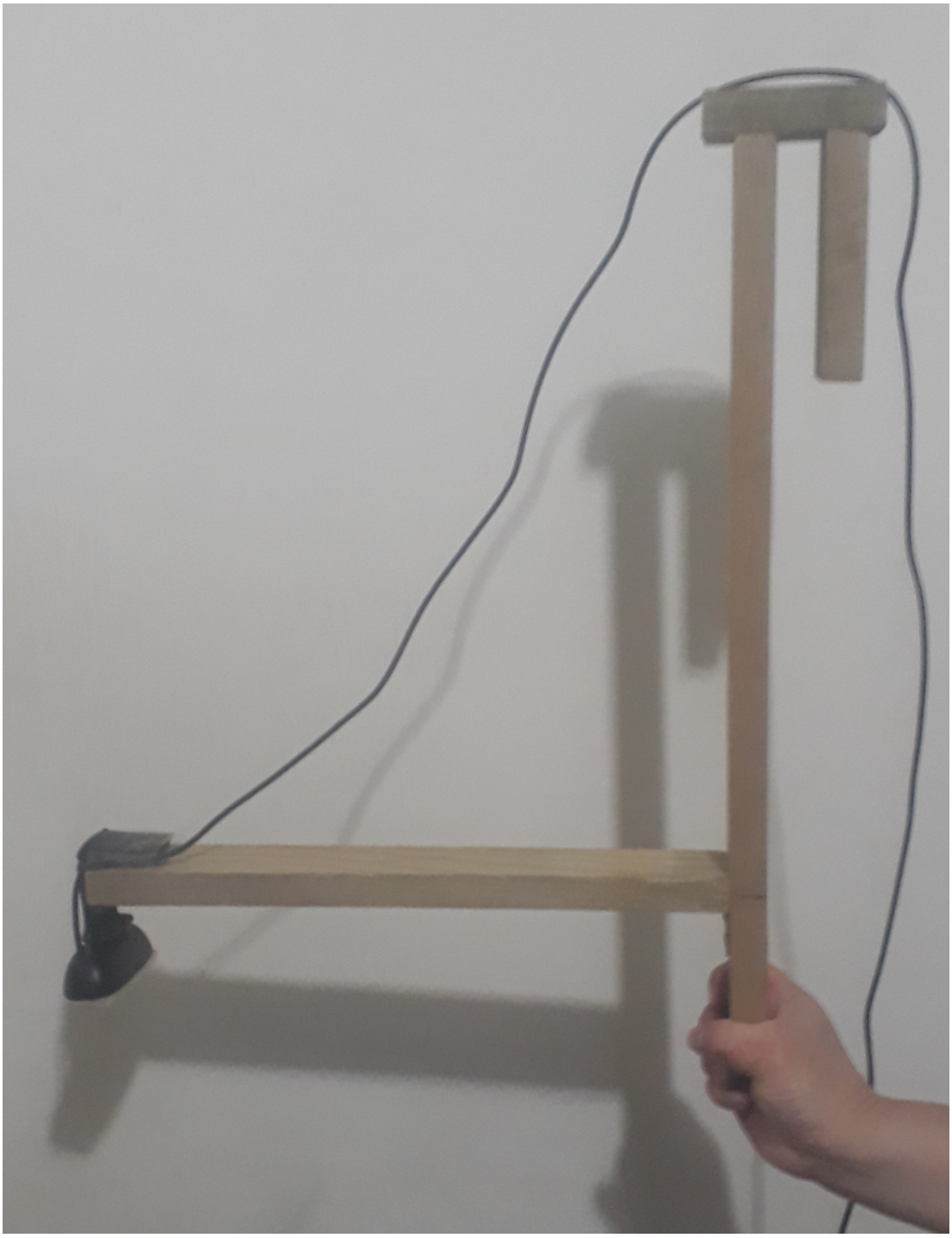


Figura 9: Base para sujeción de la cámara

Para los marcadores de las esquinas del área efectiva, se cortaron círculos de aproximadamente 16 mm de diámetro de color negro para tener contraste claro con el color de la mesa. Dentro de esta área, se colocaron los códigos y obstáculos para realizar las pruebas iniciales de la herramienta existente y comenzar a desarrollar los algoritmos en Matlab. Un ejemplo de la configuración se puede ver en la Figura 10.

Figura 10: Mesa utilizada para las pruebas iniciales

7.2. Mesa de pruebas de escalamiento

Ya que aún se está desarrollando la pandemia del Covid-19 durante el año 2021, para minimizar la exposición y así cuidar de la comunidad, se planteó primero realizar pruebas en un área más amplia (aunque no tanto como la mesa de la universidad). Al ser una mesa intermedia temporal, no se quería invertir mucho tiempo construyéndola. Por esta razón, se utilizó una tabla de melamina de 4×2 *pies* (122×61 cm aprox.) de color blanco colocada sobre el suelo debajo de la cámara como se ve en la Figura 11. Estas dimensiones son los suficientemente grandes para hacer pruebas a una escala de $2:3$ (0.8×0.6 m) de la mesa final. A su vez, se siguió utilizando la base para la cámara de la Figura 9 en el mismo lugar. Un ejemplo de la configuración final se puede ver en la Figura 11.

Esta mesa intermedia se consideró oportuna ya que al momento de escalar las condiciones, los programas pueden presentar varios problemas. Por ejemplo, la resolución de la cámara puede no ser suficiente para lograr distinguir detalles en las imágenes, la cantidad de identificadores en el área puede confundir al programa o aumentar el tiempo de ejecución, la distorsión de la imagen debido al lente de la cámara puede dejar de ser despreciable y necesitar una compensación, entre otros posibles problemas. Ya que se tiene un tiempo limitado durante el cuál se puede estar en el laboratorio, esta mesa sirvió para contemplar

dichos errores y corregirlos sin la limitante del tiempo. Aún si estos problemas se volvieran a presentar durante las pruebas con la mesa final, se tendría idea de qué hacer en dichos casos.

Figura 11: Mesa utilizada para las pruebas de escalamiento

7.3. Mesa de pruebas UVG

Por último, se realizaron las pruebas y ajustes finales con la mesa de pruebas que se encuentra en la universidad. Como se puede ver en la Figura 12, la base se construyó con tablas de melamina. A los costados, se sujetaron tubos de acero para ajustar la altura de la cámara. En la parte superior se encuentra una base en la que sostener la cámara de manera centrada. Ya que la iluminación es uno de los aspectos que mayor efecto tiene sobre el resultado del procesamiento de la imagen, esta cuenta con una lámpara LED para tener mejor control sobre la iluminación del área de trabajo. Y, aunque no se aprecie en la imagen, esta contaba con pequeños círculos hechos con marcador en forma de marcar una cuadrícula en la mesa. En la sección 8.3 se detallará más sobre estos.

Las dimensiones máximas del área de trabajo son de 120×90 *cm* de largo y ancho y una altura máxima de 120 *cm*. Los tubos de acero tiene un diámetro de 2 *pulgadas* (50.8 *mm*) y en la parte superior cuentan con un cambio de diámetro a 7/8 *pulgadas* (22.23 *mm*) roscado. La viga que soporta la cámara y lámpara es una tabla de madera con agujeros de 1 *pulgadas* (25.4 *mm*) en sus extremos para encajar con la parte roscada de los tubos.

Figura 12: Mesa de prueba Robotat utilizada ubicada en la UVG.

CAPÍTULO 8

Desarrollo de aplicación en Matlab

En este capítulo se describe del proceso de desarrollo de los algoritmos en Matlab, los problemas que se tuvieron y la manera en que se solucionaron. Se basó en gran parte sobre los programas existentes en Python y C. Sin embargo, dado que algunas de las funciones de OpenCV no son iguales o no se encuentran en la librería (o *ToolBox*) de visión por computadora, se tuvieron que hacer ciertas modificaciones. Aparte, se hicieron algunos cambios que se consideraron mejoras para el funcionamiento y versatilidad del programa. Se creó una interfaz gráfica de usuario (GUI) para facilitar el uso de la herramienta.

Tanto la implementación en Python como en C utilizan la librería de OpenCV para realizar el procesamiento de las imágenes. Aunque es posible realizar el vínculo entre OpenCV y Matlab, se decidió utilizar funciones propias de Matlab y ciertas funciones de librerías externas, ya que las funciones nativas de Matlab pueden realizar la gran mayoría del procesamiento, y se complementa con ciertas librerías en los pocos casos que se necesita. Por el mismo hecho que Matlab tiene funciones que pueden hacer lo mismo que OpenCV, son muy poco comunes las aplicaciones que hacen este enlace. Lo que conlleva a que no exista mucha documentación y ejemplos de esto. En total, se deben de instalar tres librerías:

1. *MATLAB Support Package for USB Webcams* [19]
2. *Robotics Toolbox* [20]
3. *Machine Vision Toolbox* [21]

Estas últimas dos librerías fueron desarrolladas por Peter Corke y junto con estas, publicó un libro [14] que detalla de gran manera todas las funciones de estas librerías y da ejemplos completos con el código de Matlab para mejor entendimiento.

8.1. Identificadores

8.1.1. Definición de identificadores

Antes de hablar de la implementación del programa que genera los identificadores, se darán ciertas definiciones para entender de mejor manera los capítulos siguientes. Los identificadores son las imágenes que se colocarán sobre cada robot para poder identificarlo. El diseño se definió en el trabajo [4]. Luego de considerar varios diseños se decidió por el que se puede ver en la Figura 13.

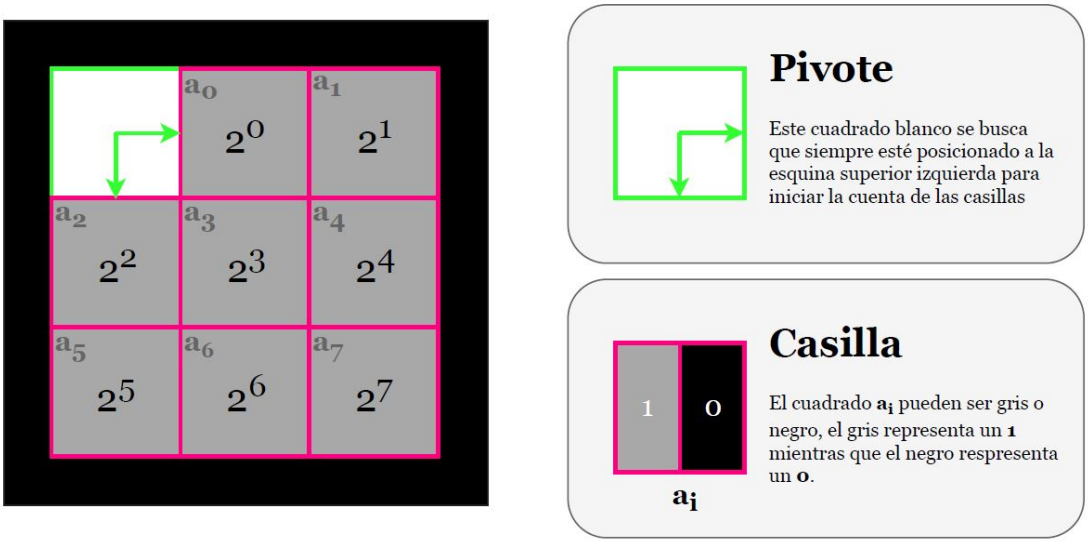


Figura 13: Definición de diseño para identificadores de robots [4]

Este consiste en un cuadrado segmentado en nueve sub-cuadrados. Uno de ellos es color blanco (con un valor numérico de 0) que será denominado "pivote". El pivote es una referencia para determinar la rotación del identificador, ya que cuando este se encuentra en la esquina superior izquierda se considera a 0 de inclinación. Los ocho cuadrados restantes podrán ser de color negro (con un valor numérico de 255) o gris (con un valor numérico de 128). Cada casilla representa un valor en una secuencia de a_0 hasta a_7 como se observa en la Figura 13 y toma el valor de 1 si es gris o 0 si es gris. El valor de identificación del robot correspondiente a un identificador se puede calcular con la ecuación 1

$$ID = \sum_{i=0}^{7} (a_i) 2^i \tag{1}$$

8.1.2. Código para generación de identificadores

El algoritmo de generación de códigos siguió el diagrama de flujo implementado en [4] con una simplificación. En el caso de Matlab, se restringió el rango de valores máximos y

mínimos a 255 y 0 respectivamente en la entrada de la interfaz gráfica directamente en lugar del programa. Esta implementación se mantuvo en una función en un archivo separado que es ejecutado dentro de la interfaz. Se optó por esta opción en lugar de tener las funciones en el código interno de la interfaz ya que en caso que alguien únicamente quiera utilizar esto, se puede descargar el archivo y no es necesario instalar la interfaz completa junto con todas sus dependencias de *toolboxes*. Para darle mayor flexibilidad al usuario, también se agregó una entrada del nombre del archivo en lugar de darle un nombre fijo y poder tener diferentes códigos con nombres diferentes.

Se consideró cambiar el formato de la imagen a PNG o JPEG. Según [22], las mayores diferencias entre PNG y JPEG son la calidad y el tamaño del archivo, siendo el primero diez veces más pesado que el segundo, pero con mejor calidad. Ya que la imagen del identificador no tiene mayores detalles que necesiten de un formato PNG, se decidió no utilizar este. Comparando el JPEG con el original JPG utilizado en la herramienta ya existente, el artículo en [23] hace notar que estos similares, con la diferencia que JPEG es un formato más reciente, pero utilizado principalmente en cámaras digitales y en páginas web. Por lo que se decidió mantener el formato de JPG.

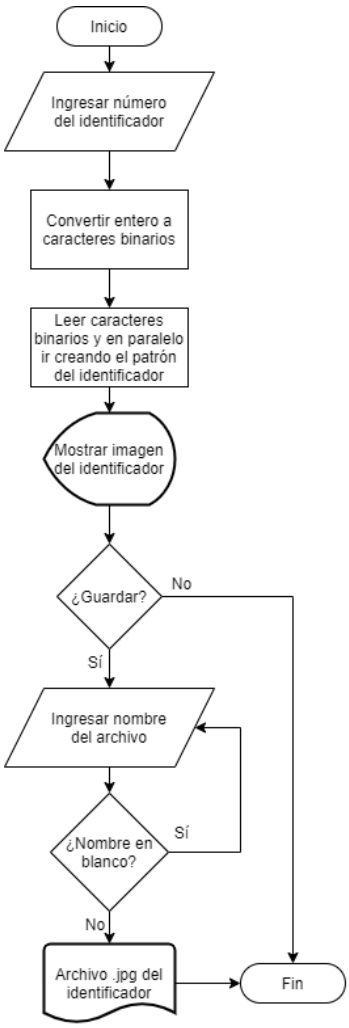


Figura 14: Diagrama de flujo del algoritmo para generación de identificadores.

8.2. Código para calibración de la cámara

El algoritmo utilizado para calibrar la cámara sigue los mismos principios de los programas en Python y C. La Figura 15 muestra el diagrama de flujo a seguir en el programa de calibración. Primero, cabe mencionar que se agregó un modo manual de calibración en caso de no llegar a obtener los resultados deseados por problemas de iluminación, por ejemplo. Esto ahorra la mayoría del procesamiento lo cuál disminuye el tiempo. También es importante hacer notar que se incluyó el algoritmo de detección de obstáculos ya que este es parte de la calibración, pero se profundizará en mayor medida en la siguiente sección (8.3).

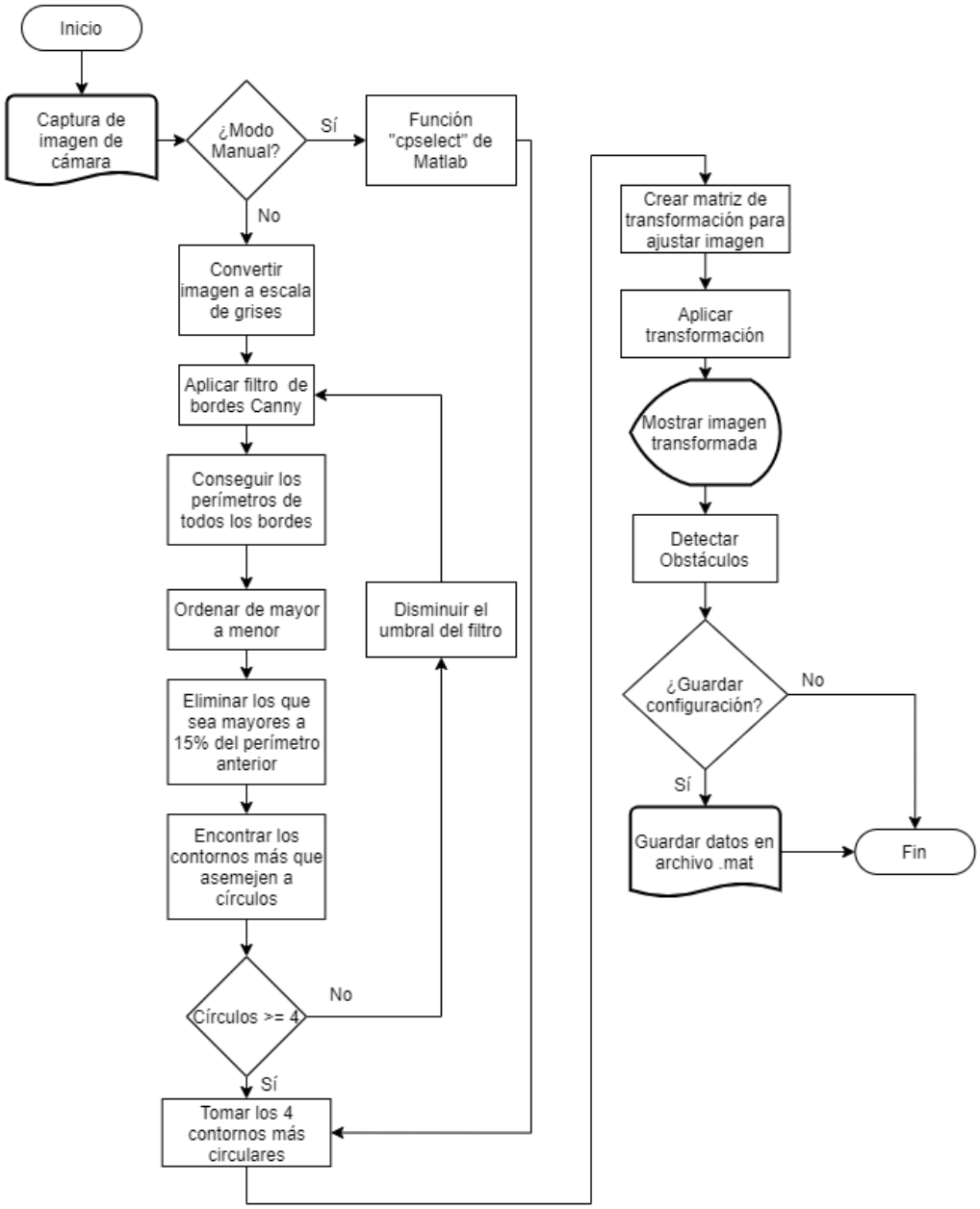


Figura 15: Diagrama de flujo del algoritmo para calibrar la cámara.

Para la detección de bordes, Matlab cuenta con diferentes métodos que, a pesar de que tienen el mismo propósito, lo ejecutan de distintas maneras y pueden llegar a resultados muy distintos bajo ciertas condiciones. Los métodos son: *Sobel, Canny, Prewitt, Roberts,* y *fuzzy logic*. Se realizaron pruebas con todos estos, pero algunos eran muy sensibles y detectaban pequeñas sombras como bordes aún con valores de umbral altos, por lo que se utilizó el filtro Canny (que es uno de los más comunes por su precisión en la salida). Al realizar pruebas con niveles bajos de luz, se descubrió que, aunque el umbral es adaptativo, un valor fijo de umbral no lograba detectar los bordes a partir de cierto punto. Por lo que se decidió ir disminuyendo el valor hasta encontrar cuatro o más marcadores circulares como potenciales esquinas. Durante las pruebas, se encontró que a veces la detección de bordes tomaba el contorno de la mesa como un borde. Por la manera en que se detectan los demás contornos, se debe de eliminar este contorno, de haberlo. Un ejemplo del resultado de la aplicación de detección de bordes se puede ver en la imagen superior derecha de la Figura 16.

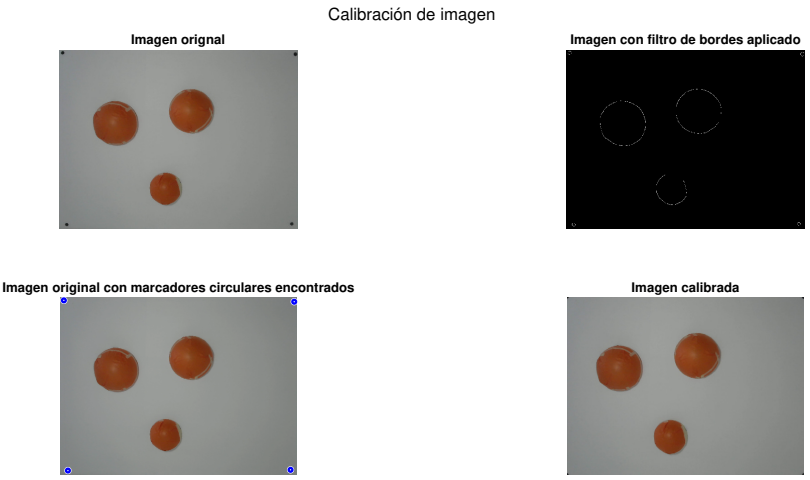


Figura 16: Demostración de los pasos que se siguen durante la calibración.

Para la detección de los círculos en sí, se utilizó la función *imfindcircles* de Matlab. Esta encuentra los contornos más similares a círculos dentro de una imagen dada. Los parámetros de salida son los centroides, los radios y un valor llamado *metric* de cada círculo encontrado. Este último es una medida entre 0 y 1 que denota qué tan circular es el contorno encontrado, siendo 1 un círculo perfecto. Estas tres salidas se encuentran ordenadas según este parámetro y se puede limitar la sensibilidad para descartar los contornos con bajo *metric*. En la imagen inferior izquierda de la Figura 16 se puede muestran los círculos encontrados con esta función.

Para enderezar y recortar la imagen para únicamente tener el área de interés, se realizar una transformación. Esta se puede definir a través de una matriz y la misma se encuentra con la función *fitgeotrans* que toma como entrada las cuatro esquinas de la imagen original y las cuatro esquinas encontradas anteriormente. Ya que las esquinas seleccionadas pueden no estar formando un rectángulo perfecto o por las deformaciones del lente de la cámara, se necesita realizar correcciones. Por esta razón, se utilizó una transformación de tipo proyectiva (este siendo un parámetro de la función). Existen otros tipos de transformaciones pero algunas de estas no pueden compensar por estas imperfecciones mencionadas y otras hacen

distintas compensaciones para distintos casos de imágenes, como por ejemplo, una imagen que tiene distintas distorsiones en distintas partes de la imagen. La imagen inferior derecha de la Figura 16 es la resultante luego de aplicar la matriz de transformación resultante de este ejemplo de calibración.

8.3. Código para cartografiar el terreno

Esta parte del código no es parte de la herramienta original, por lo que no se tenía en que basarse para la implementación en Matlab. El objetivo de este código es detectar la presencia de obstáculos ya que es información potencialmente útil para que los programas de los vehículos logren esquivarlos o sepan con anterioridad la existencia de estos para generar una trayectoria que no intreseque los mismos. En la Figura 17 se muestra el diagrama de flujo de el algoritmo completo.

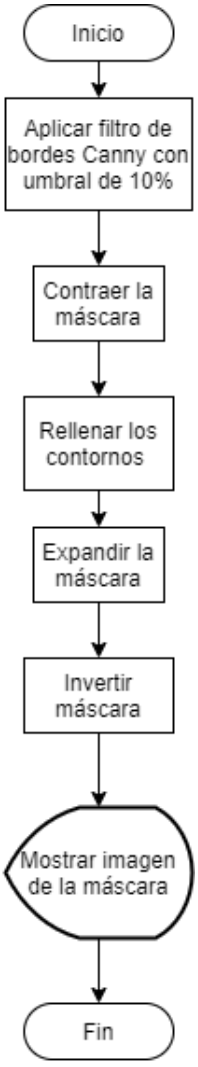


Figura 17: Diagrama de flujo del algoritmo para detectar obstáculos.

En esta parte del código, se decidió hacer una gran simplificación para evitar sobrecomplicar el código; Los obstáculos serán estáticos, esto nos permite hacer este proceso de calibración una única vez (primera "fase") y utilizar esta información luego durante la toma de pose (segunda "fase"). Al tener primero una "fase" de calibración separada de la detección de identificadores y luego una segunda "fase" de detección que se mantiene en bucle hasta que se desee, se decidió hacer la convención que durante la "fase" de calibración, no se tendrá ningún identificador en el área de la mesa. Esto implica que cualquier objeto dentro de la mesa durante esta "fase" se tomará como un obstáculo, se incluirá en una máscara. Esta máscara será una imagen del mismo tamaño que la imagen original, pero binaria, es decir, solo contendrá valores de cero o uno. En este caso, los píxeles con valor cero denotaran puntos en donde se encuentra un obstáculo y los valores uno denotan espacios por donde los robots pueden maniobrar libremente.

Esta simplificación causó problemas con los círculos hechos con marcador mencionados en la sección 7.3, ya que todos estos se detectaban como obstáculos en lugar de tomarse como parte de la mesa. Una opción era restringir el tamaño mínimo de los obstáculos, pero esta sería una gran limitación. En esta instancia, se optó por cubrir la mesa con pliegos de papel blanco para homogeneizar el color de la mesa.

Con esto establecido, se definió el umbral del filtro de bordes suficientemente bajo para que se pueda detectar cualquier obstáculo, pero no lo suficientemente bajo para que se confundan algunas sombras que puedan generarse. Luego de hacer varias pruebas con diversos obstáculos y condiciones de iluminación, el valor final para esto fue 0.1 o 10 %.

Antes de rellenar los contornos, se contrae la máscara ligeramente ya que se pueden generar "bordes" de unos pocos píxeles que en realidad no son parte de ningún obstáculo; esto debido al valor de umbral bajo, pero al contraer la máscara, se pueden eliminar sin problemas. Y luego de rellenarla, se expande el resultado ya que a veces la detección de bordes muestra bordes ligeramente más gruesos y puede interferir con la toma de pose.

Por último, se invierte la máscara ya que se desea mantener lo que no sean obstáculos, pero el procesamiento seguido regresa lo opuesto. Debido a la simplificación inicial, este programa es muy simple. A la izquierda de la Figura 18 se muestra la imagen luego del proceso de calibración descrito en la sección 8.2 y del lado derecho la máscara de obstáculos resultante del proceso descrito en esta sección.

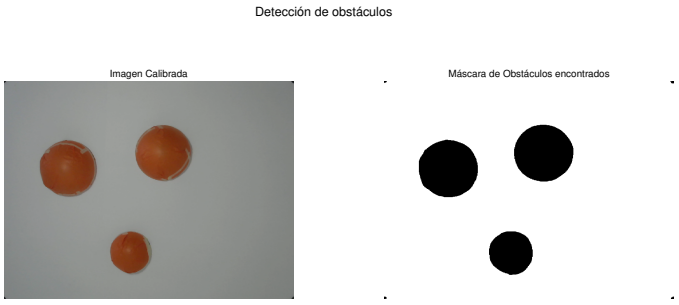


Figura 18: Demostración de los pasos que se siguen durante la detección de obstáculos.

8.4. Código para toma de pose de identificadores

Este programa se implementó en un ciclo que se ejecuta hasta que se desee detener el programa para que esté funcionando en tiempo real. Una opción para esto fue utilizar un temporizador para tener una tasa de refresco constante, pero el tiempo promedio puede variar ampliamente dependiendo de la computadora (como se hará notar en el capítulo de resultados 9). Debido a esto, si se hace un valor muy bajo, este puede crear problemas cuando el temporizador venza y no se haya terminado el procesamiento actual. Y si se mantiene un temporizador alto, no se estaría aprovechando el poder de procesamiento de las computadoras capaces de hacer el procesamiento más rápidamente. Por lo que se consideró más óptimo utilizar únicamente un ciclo continuo.

El diagrama de flujo del algoritmo se puede ver en detalle en la Figura 19. Como se estableció que las condiciones de la calibración no varían, la transformación encontrada en esta se utiliza siempre sin ningún cambio. Al saber que los obstáculos son inmóviles para este trabajo, se aplica la máscara para no intentar buscar identificadores en estas áreas y encontrar accidentalmente algunos.

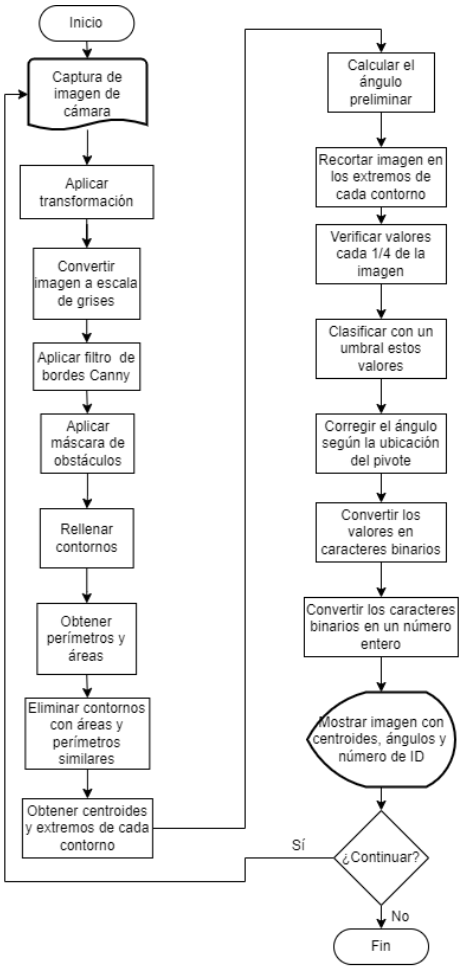


Figura 19: Diagrama de flujo del algoritmo para obtener la pose de los identificadores.

Para el filtro Canny, se consideró hacer de manera similar a la calibración e iterar decrementando el valor del umbral, pero luego de realizar diversas pruebas con múltiples tamaños de código y distintas condiciones de luz, se concluyó que un umbral de 50 % resulta en un filtrado exitoso en todos los casos realizados. Luego de aplicar la máscara, ya que las sombras o tonos pueden cambiar entre iteraciones debido a la luz ambiente, se pueden tener bordes extra como se pueden ver en la Figura 20. Debido a esto, se rellenan los contornos cerrados. En el caso de los identificadores, estos quedarían como un cuadrado blanco relleno; mientras que los bordes sobrantes no tendrían ningún tipo de relleno.

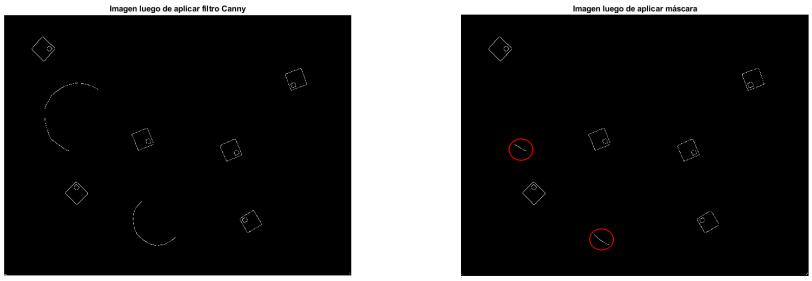


Figura 20: Imagen con filtro de bordes y máscara.

Al utilizar la función *regionprops* solicitando perímetros y áreas, estas se pueden comparar entre sí. En el caso de los identificadores, estos dos serían muy distintos, mientras que los bordes extras serán muy similares. De esta manera, se eliminarían los que fueran similares, haciendo un pequeño filtrado de los contornos según esta proporción entre área y perímetro. En la Figura 21 se muestra un ejemplo del filtrado y se marca con círculos azules los centros de los contornos encontrados.

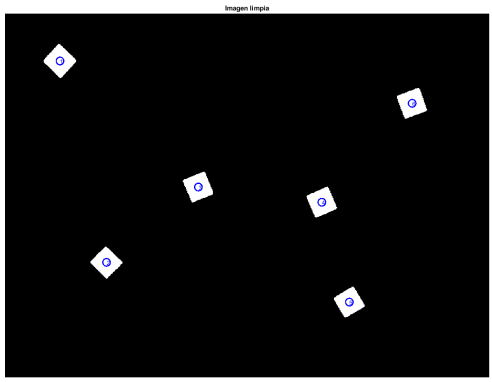


Figura 21: Imagen con filtro de bordes y máscara (con filtro de proporción).

La función *regionprops* da como resultados propiedades de las regiones de una imagen binaria (o en blanco y negro). Dos de las posibles propiedades son *BoundingBox* y *Orientation*. La primera devuelve los puntos del rectángulo horizontal más pequeño que puede encerrar completamente un área. Y la segunda da el ángulo de orientación del área. Se decidió no utilizar la primera ya que la orientación de este rectángulo siempre es horizontal, aunque el identificador se encuentre rotado, por lo que esto no es muy útil para el procesamiento. En su lugar se utilizó la función encontrada en [24]. Esta función da un rectángulo con la misma orientación que el identificador. Por lo que esto se puede utilizar para recortar la sección de la imagen original y facilitar el resto del código. En la Figura 22 se muestra en colores las esquinas de uno de los cuadros retornados por esta función. Y un ejemplo de los recortes de identificadores resultantes de este proceso se muestra en la Figura 23.

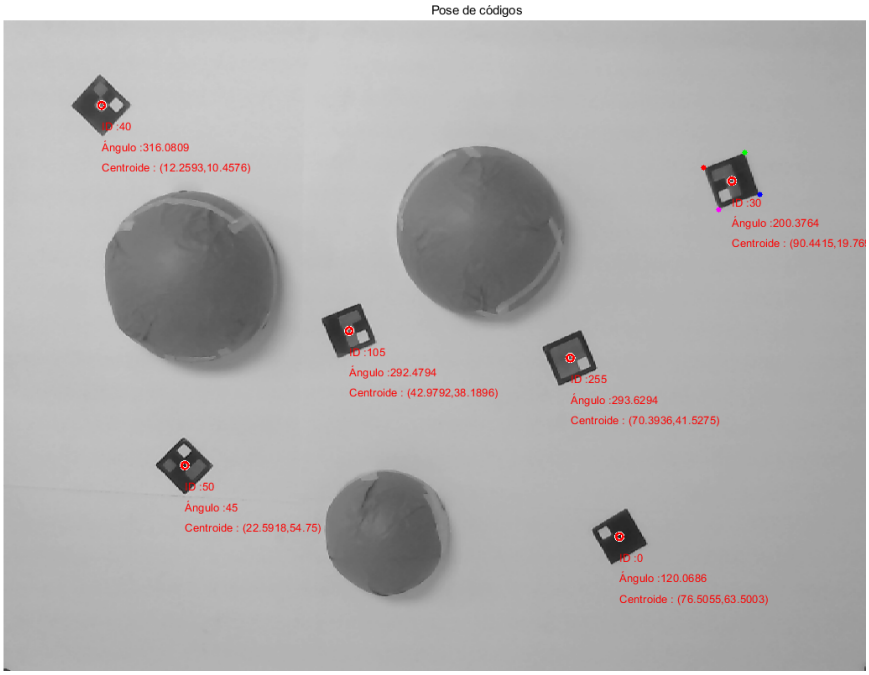


Figura 22: Imagen con números de identificación y poses encontradas.

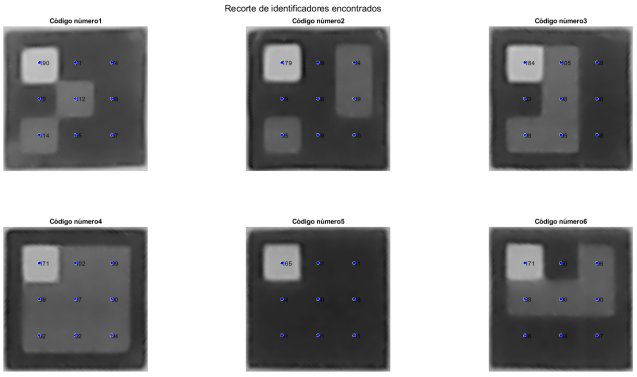


Figura 23: Recortes de los identificadores encontrados.

Para encontrar el ángulo, se podría haber utilizado la propiedad *Orientation*, pero, al menos en Matlab, se reconoce el eje principal (respecto al cual se referencia el ángulo) como el que tenga mayor cantidad de píxeles de longitud, por lo que si un borde resulta más grueso que los demás, puede afectar el resultado y no ser tan exacto. Ya que se contaba con las esquinas mostradas en la Figura 22 de cada identificador, únicamente se calculó el ángulo entre dos de ellas. En específico, las primeras dos (colores rojo y verde) para mantener consistencia entre todos y que sea fácilmente escalable. Estas siempre dan un valor entre 0° y 90°. Este ángulo preliminar se ajustaría dependiendo del lugar del pivote en el recorte de la Figura 23. Cabe notar que esta fue la manera en la que se implementó en las fases anteriores.

Por último, para el umbral de los valores puntuales del número del identificador, se terminó usando menor a 40 para un 0 (o cuadro negro), entre 40 y 145 para 1 (o cuadro gris), y mayor a 145 para detectar el pivote. Este último se hace para rotar los códigos y que siempre se encuentre el pivote en la esquina superior izquierda para que cada 1/4 de la imagen (como los círculos azules en la Figura 23) equivalga al mismo valor en la cadena binaria sin importar la orientación original.

En la Figura 22 se muestra la imagen calibrada. Sobre esta, se marcan con círculos rojos los centros de los marcadores y se agrega el texto con la información de identificación y pose a un lado.

8.5. Interfaz gráfica de usuario

Teniendo los códigos en programas de texto funcionales, se procedió a diseñar una interfaz gráfica de usuario (GUI por sus siglas en inglés). Para separar los tres códigos principales (generación de códigos, calibración y toma de pose) se hicieron pestañas separadas a través de las cuales el usuario puede navegar dependiendo de lo que desee hacer. En cada una de estas se colocó una imagen con el logotipo de la universidad y un área de texto originalmente en blanco en la cual se indicará al usuario de cualquier mensaje relevante como errores o resultados de procesos.

Una captura de la pestaña de generación de códigos se puede ver en la Figura 24. Se tiene una entrada numérica que directamente limita el valor permitido entre 0 y 255 para ahorrar esta verificación en el código. Se tiene también una entrada de texto en la que se ingresa el nombre con el que se desea guardar la imagen. Además, hay un botón para generar el código y otro para guardarlo. Este último únicamente se activa luego de haber generado un código. Y cabe mencionar que se despliega la vista previa de la imagen en una figura externa en lugar de estar embebida en la interfaz ya que si el usuario desea guardar la imagen con otro formato distinto a ".jpg" puede hacerlo desde la figura. Esto da más opciones al usuario de desearlas.

Figura 24: Captura de pestaña de generación de códigos de interfaz gráfica.

En la Figura 25 se ve una captura de la pestaña de calibración. Lo primero que se ve es un menú desplegable para seleccionar la cámara con la que se desea trabajar. En caso que se trabaje con múltiples cámaras para diferentes propósitos o que se tenga una cámara integrada, como en los portátiles. Y junto con este un botón para conectar a la cámara. Luego, un botón para realizar la calibración (incluido la detección de obstáculos) y, en caso de no obtener los resultados deseados, uno para realizar una selección manual. Cabe mencionar que luego de hacer la selección, se debe presionar de nuevo el botón de calibrar para que se ejecute el código.

Ya que el algoritmo procesa las posiciones en píxeles como dimensionales, se debe hacer una conversión entre píxeles y las dimensionales que se deseen. Para eso se agregaron dos entradas numéricas en las que se ingresan los valores de longitud y ancho del área efectiva, es decir, la distancia entre los marcadores circulares que denotan las esquinas. Estas dimensiones no tienen que ser estrictamente alguna unidad fija ya que no se hace ningún cálculo que las requiera. Por lo que el usuario debe de ingresar las medidas en las unidades que desea los resultados.

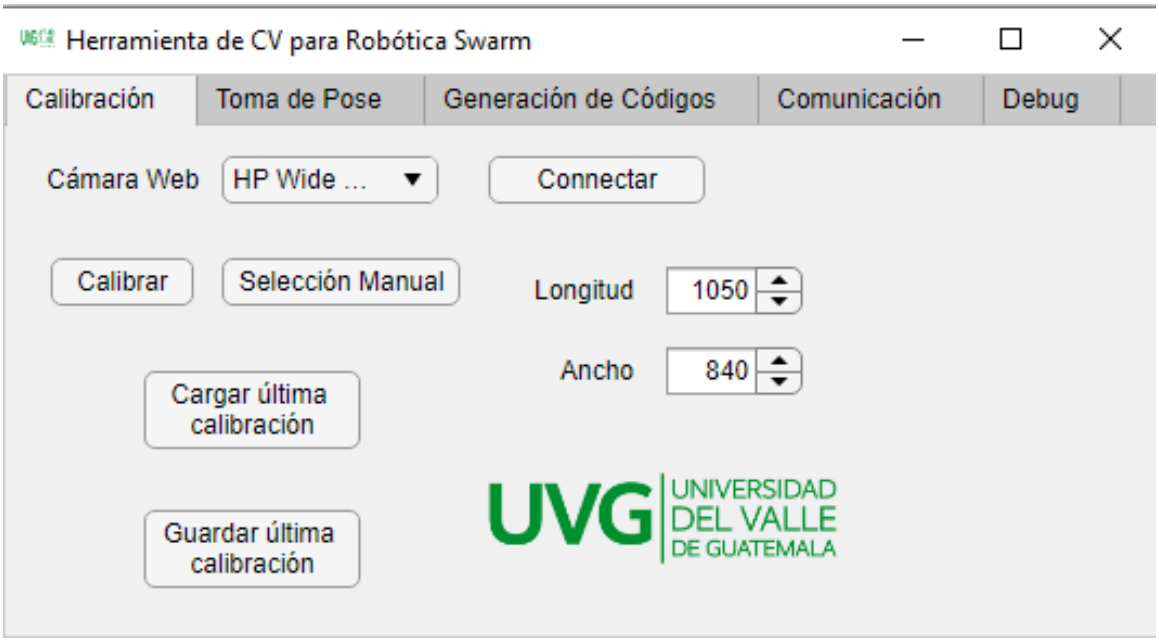


Figura 25: Captura de pestaña de calibración de interfaz gráfica.

Ya que es muy probable que las esquinas de la mesa y los obstáculos no varíen entre sesiones de uso del programa, se agregó un botón para guardar la calibración realizada y otro para cargar la última calibración guardada. Esto guarda cuarto variables: la variable de la conexión con la cámara, la matriz de transformación de las esquinas, la máscara de obstáculos y una referencia de las dimensiones de la imagen. Esto ahorra todo el proceso de calibración entre sesiones de trabajo. Luego de cargar las variables, se despliegan una imagen de los resultados de la calibración para verificar que sigan siendo correctos y no se necesite de una nueva calibración.

Se muestra una captura de la pestaña de toma de poses en la Figura 26. Aquí se tiene un botón para comenzar a tomar pose y otro para detenerlo. El primero se activa únicamente luego de tener los datos de calibración, ya que sin ellos, no se puede ejecutar el código. En la parte derecha, se tienen tres casillas de verificación. Ya que para los resultados era necesario guardar los tiempos entre ejecuciones, se agregó una casilla para activar o desactivar esta parte del código, ya que no siempre se desea ejecutar. Otra casilla para verificar si se desea realizar únicamente una toma de pose o si se desea mantener en tiempo real. De ser así, se activa el botón para detener la toma de pose cuando se requiera. Y la última casilla para desplegar los resultados. Esto puede acelerar el proceso ya que Matlab requiere de cierto tiempo en desplegar las imágenes y sus resultados.

Figura 26: Captura de pestaña de toma de pose de interfaz gráfica.

Ya que durante el desarrollo de los códigos se tuvieron errores, para facilitar la depuración del programa y la identificación de errores, se agregó una pestaña de depuración que se puede ver en la Figura 27. En esta se tienen opciones como desplegar imágenes intermedias de los procesos de calibración y toma de pose, y trabajar con imágenes en lugar de cámara. Todas estas se fueron agregando conforme se consideraron necesarias.

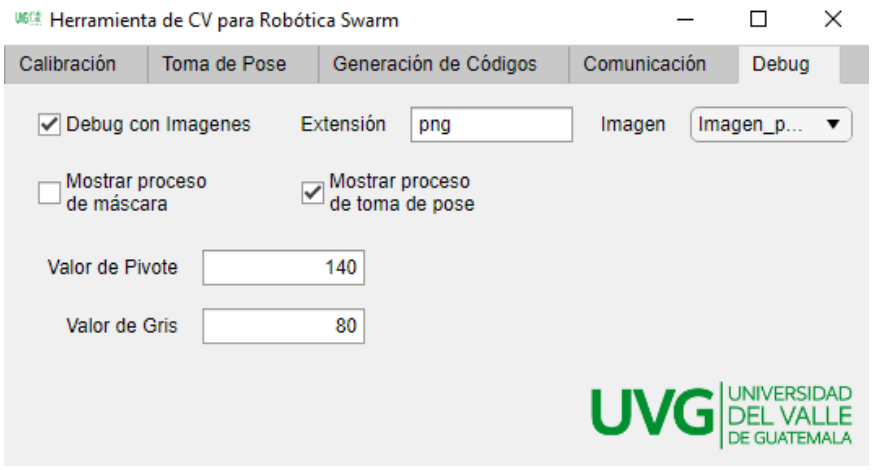


Figura 27: Captura de pestaña de depuración de interfaz gráfica.

Y por último, debido a una integración que se decidió hacer para validar el concepto de implementación en robótica de enjambre, se agregó una pestaña para la comunicación con los agentes. En esta se puede habilitar o deshabilitar distintos protocolos de comunicación y agregar o eliminar las direcciones IP de los dispositivos necesarios en la red. Se hablará más a detalle de esto en la Sección 8.7. Una captura de esta pestaña se muestra en la Figura 28.

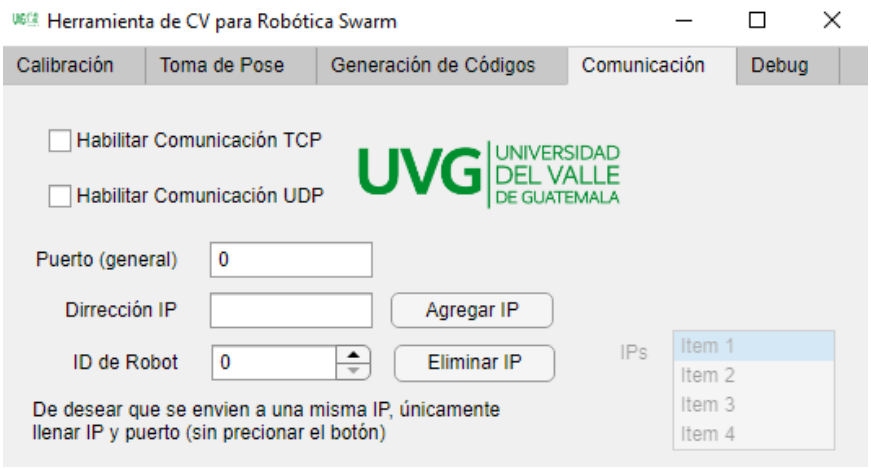


Figura 28: Captura de pestaña de comunicación de interfaz gráfica.

Aquí se tiene la opción de utilizar protocolo UDP o TCP. Aunque para la integración únicamente se utilizó el protocolo UDP, se mantuvo la opción de TCP ya que esta es una opción muy utilizada también y se desea dar más opciones al usuario. Para ambas se necesitan los mismos parámetros. Para esto se tiene una entrada numérica para el puerto de comunicación, una entrada numérica para el número de identificador del agente y una entrada de texto para la dirección IP del gente. Y botones para agregar y eliminar IPs.

Por lo que por cada agente que se desee conectar, se debe de cambiar la dirección y su respectivo número de identificador. Conforme se agreguen más agentes, aparecerán en el listado de la parte inferior derecha. Esta no es editable directamente, sino se deben de utilizar los botones de agregar y eliminar para modificarla. Cabe mencionar que ya que se utiliza la misma entrada en la interfaz para el puerto de comunicación, no se pueden tener habilitados ambos protocolos.

8.6. Pruebas de paralelización

En Matlab existe la librería llamada *Parallel Computing Toolbox* que se utiliza para paralelización, pero su uso más común es ejecutar múltiples simulaciones de un programa con diferentes condiciones iniciales o diferentes parámetros. Sin embargo, es posible utilizar las funciones de esta toolbox para crear funciones diferentes que se ejecuten como hilos separados. También da opción de hacer uso de una tarjeta gráfica dedicada para procesamiento intensivo de imágenes. Como por ejemplo, en el artículo por Diener y Elsherbeni [25] se reportan aceleraciones de tiempo significativas utilizando métodos desarrollados con esta librería.

En esta librería se pueden ejecutar códigos paralelizados con trabajadores locales o, de ser necesario, en múltiples computadoras conectadas a la misma red. Esto último usualmente se utiliza cuando se trabaja con cantidades de datos tan grandes que no se pueden procesar únicamente en una computadora. Esto no será necesario para los programas desarrollados en este trabajo, por lo que se utilizarán trabajadores locales.

Analizando el diagrama de flujo, se decidió dividir el código de la abstracción de pose de los robots en dos hilos secundarios y un hilo principal. En todos los códigos multihilo existe un hilo principal que maneja los hilos secundarios. En este caso, el hilo principal (Hilo 0) se encargará únicamente de manejar los hilos secundarios, intercambiar información entre hilos y de guardar los datos en un archivo. Esto puede ser reemplazado por transmisión de los resultados a través de los diversos protocolos de comunicación disponibles. Los diagramas de flujo de los hilos secundarios se pueden ver en la Figura 29.

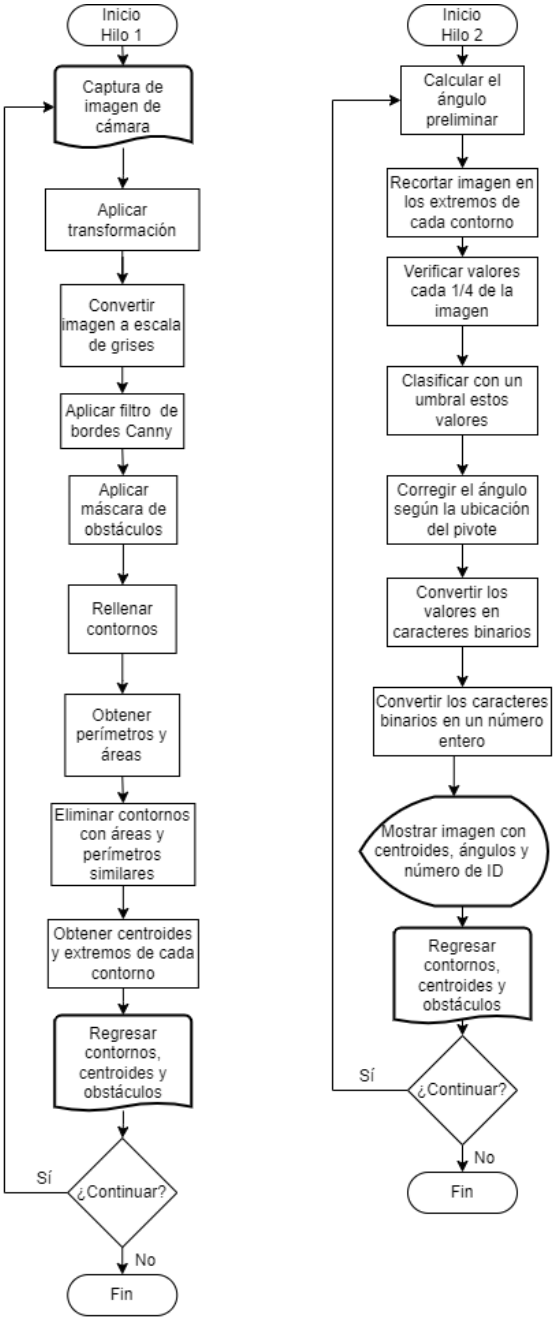


Figura 29: Diagrama de flujo de los hilos utilizados para paralelización.

El primer hilo secundario (Hilo 1) se encargaría de obtener la captura de la imagen de la cámara, realizar el pre-procesamiento y obtener los datos de los contornos en la imagen. Este hilo devuelve al hilo principal la imagen de la cámara en escala de grises y los cuadros delimitadores de los contornos de los identificadores.

El segundo hilo secundario ("Hilo 2") ejecuta a su vez múltiples sub-hilos, el número depende de la cantidad de identificadores. La librería cuenta con una función (*parfor*) que ejecuta un hilo distinto por cada índice dado. Por lo que en este Hilo 2, se ejecutan n sub-hilos siendo n la cantidad de identificadores en el área de trabajo. Cada uno de estos hilos recibirá los datos devueltos por el Hilo 1. Con esta información, calculará la información restante del identificador asignado al hilo. Es decir, se obtendrán las coordenadas del centro, el ángulo de orientación y el número de identificador. Esta información será regresada al Hilo 2 y este se encargará de almacenarlo y ordenarlo. Por último, estos datos se darán de vuelta al Hilo 0 y este se encargará de almacenarlos o transmitirlos.

Se consideró realizar más divisiones en el Hilo 1, pero en ese caso, se necesitaría intercambiar mayor cantidad de variables y cada variable de tamaños grandes. Por lo que se mantuvo en un hilo de manera que las variables finales del hilo fueran solo dos y del menor tamaño posible.

Cabe mencionar que la única diferencia entre el programa paralelizado y no paralelizado es la creación y ejecución de las funciones en hilos. Todos los demás aspectos del código se mantuvieron iguales para que la única variable en esta prueba fuera la paralelización.

8.7. Validación de concepto

Con el fin de validar que el concepto de esta herramienta es realmente aplicable a robótica de enjambre, se trabajó en conjunto con otros dos proyectos de graduación en desarrollo paralelo al de este documento. Uno por Alex Maas [26] y otro por Andrés Sierra [27]. El primero implementa una versión modificada del algoritmo PSO en múltiples Raspberry Pi (agentes). Estas se conectan a una red por WiFi y se comunican entre sí a través del protocolo de comunicación UDP. En cada iteración, cada agente obtiene sus coordenadas, calcula su propia iteración local del algoritmo con una función de costo dada y envían sus resultados entre sí. Esta función de costo es una función numérica de la forma $z = f(x, y)$; por ejemplo, la función de una esfera. Con esta información, determinan cuál es el valor mínimo para seguir en esta dirección y encontrar la meta. Esta meta es un valor mínimo (local o global). En este caso, se busca proveer a cada agente con su pose respecto un sistema de coordenadas global para que estas logren realizar sus cálculos.

Idealmente, se debería de tener un identificador sobre los agentes (en este caso las Raspberry Pi). Sin embargo, aún no se cuenta con una plataforma móvil que realice el movimiento automático del agente. Y aunque este proyecto ya se encuentra en desarrollo, no se encuentra en una etapa en la que se pueda incluir en esta integración, por lo que se tuvo que prescindir de la plataforma móvil.

Se consideró tener los identificadores sobre las Raspberry Pi y realizar los movimientos de forma manual según fuesen necesarios. Pero esto traía una gran inconveniencia: que cada una

necesitaría estar conectada a una fuente de alimentación lo que significa múltiples cables en el área de trabajo. Esto resulta en mayor problema para la detección de identificadores pero sí estorbarían al momento de ajustar las posiciones entre cada iteración. Por lo que se decidió mantener las Raspberry Pi fuera de la mesa de pruebas y únicamente los identificadores se colocarían directamente sobre la mesa de la misma manera que se realizó el desarrollo de esta herramienta.

De igual manera, se realizó una verificación que al tener el relieve de un objeto debajo del identificador, este no causaría ningún problema. Aunque se genera una sombra con el suficiente contraste como para que el filtro Canny lo detecte, esta es eliminada y se logra exitosamente una toma de pose como se muestra en la Figura 30.

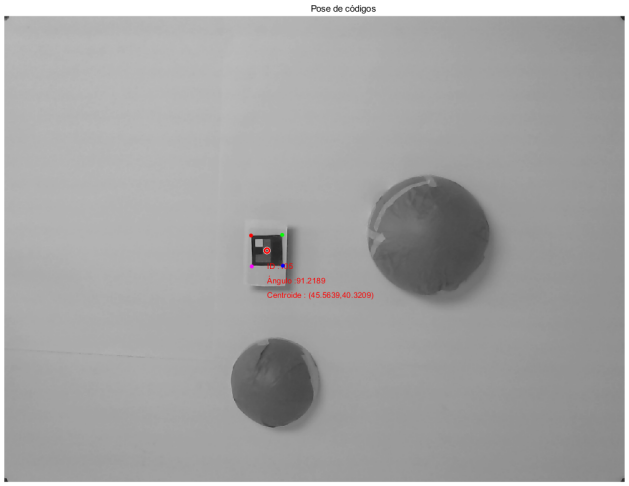


Figura 30: Toma de pose con identificador sobre una Raspberry Pi.

En segundo trabajo con el que se realizaron pruebas iniciales de integración implementa una versión del algoritmo Ant Colony Optimization (ACO). Este dado un mapa de nodos, un punto de inicio y uno de meta, calcula la ruta óptima atravesando los nodos provistos. Y procede a ejecutar un controlador que modulará el movimiento del agente para seguir la trayectoria establecida por el algoritmo. En este caso, se buscaría proveer al agente con el mapa de nodos; tanto las coordenadas de los nodos como las conexiones entre ellos. Y luego de tener establecida su trayectoria, ir proveyendo al agente de su posición actual para actualizar el controlador.

El mapa de nodos con el que se ha validado el algoritmo ACO ha sido en forma de cuadrícula. En el caso más sencillo, todos los nodos se conectarían entre sí, pero también existe la posibilidad que algunas conexiones no sean posibles de navegar. Principalmente, esto ocurriría cuando se detecte algún objeto entre ellos. Por lo que se imprimieron y recortaron marcadores circulares iguales a los utilizados para las esquinas. Estos denotarán las posiciones de los nodos del mapa. Y también rectángulos de x cm estos se utilizaran como obstáculos entre los nodos para interrumpir las conexiones.

Ya que la manera en la que se desarrolló la detección de obstáculos requiere que estos sean estáticos, no se podrá realizar una prueba dinámica en la que se interrumpan conexiones entre iteraciones.

Código adicional

En el caso de las pruebas con el algoritmo PSO, no se realizó ninguna modificación ni adición al código original. Sin embargo, para el algoritmo ACO, se necesita detectar los nodos sin tratarlos como obstáculos. Por lo que se decidió hacer un paso extra durante la calibración. Luego de encontrar las esquinas del área de trabajo y detectar los obstáculos, se colocarán los marcadores circulares que denotarán a los nodos. Con esta imagen se realiza el diagrama de flujo que se ve en la Figura 31. Este comienza de la misma manera a la toma de pose, realizando una captura de la cámara y aplicando varios filtros para obtener una imagen con únicamente los objetos que se desean. En este caso, se tendría una imagen con los nodos. En esta se identifican los círculos de la misma manera que se realiza en la calibración.

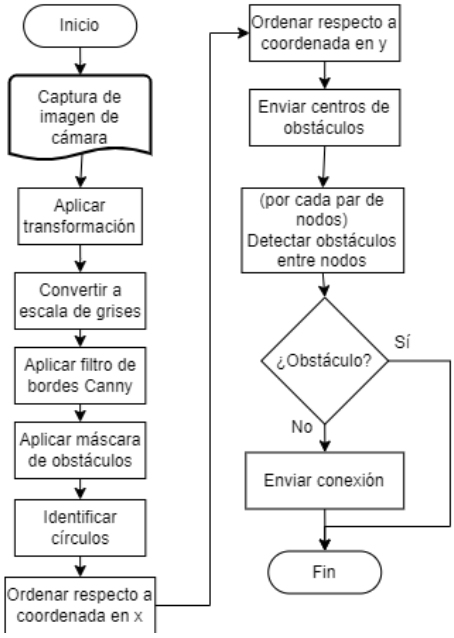


Figura 31: Diagrama de flujo de proceso de detección de nodos.

Ya que los nodos deben tener un orden y se encuentran distribuidos en forma de cuadrícula, se ordenan. De primero se ordenan respecto a la coordenada x, agrupando los que tengan un valor similar en esta coordenada. Y luego se ordenan conforme la coordenada en y. Teniendo los datos de todos los nodos de manera ordenada, se procede a enviar las coordenadas de cada nodo. Para esto, se estableció una cadena de caracteres que siga la siguiente forma: C, ID, X, Y, en donde ID es el número de nodo, X es la coordenada en la dirección x, Y es la coordenada en la dirección y con el identificador "C" para denotar que se envía un conjunto de coordenadas. Estos valores se redondearon a una cifra, ya que en este caso no se requerían cambios tan pequeños en posición.

Y por último, se verifican las conexiones entre nodos. Para esto, se toma un par de nodos colindantes y se realiza una interpolación entre estos, generando una serie de puntos de muestreo. En cada uno de estos, se toma el valor de la máscara de obstáculos obtenida en la calibración para verificar si existe un obstáculo entre estos nodos. En todas las pruebas realizadas se mantuvo una cantidad de 10 puntos de muestreo entre nodos, pero esto limita el tamaño y posición de los obstáculos. Si se tienen obstáculos más angostos, se tiene que agregar más puntos de muestreo o verificar que el obstáculo se encuentre sobre alguno de estos. De no existir un obstáculo entre el par de nodos, se envía la conexión de la siguiente forma: G, ID_1, ID_2, donde ID_1 e ID_2 son el par de nodos a conectar y "G" es el identificador para denotar que el comando es de conexión.

Adicionalmente, se tuvo que modificar el código de toma de pose del robot. Ya que el código original no toma en consideración la existencia de los nodos, se tuvo que agregar un filtro para eliminarlos. En este caso, se utilizó la propiedad "Circularidad" de cada contorno. En este caso, Matlab calcula una proporción entre área y perímetro de cada contorno de la siguiente manera:

$$\frac{4 * Área * \pi}{perímetro^2} \tag{2}$$

Para un círculo perfecto, el valor de la circularidad es 1.00. Sin embargo, en las imágenes no se puede tener una circularidad perfecta debido a los píxeles que lo conforman. Esto no permite tener un círculo, sino una aproximación de un círculo. Por lo que los valores reales de circularidad de un círculo según pruebas iniciales se encuentra entre 1.05 y 1.3. Entonces, durante el proceso de toma de pose, antes de obtener los centroides y demás datos, se eliminan los contornos que tengan circularidad en este rango. Esto deja en la imagen únicamente los identificadores y se puede continuar con el algoritmo como se explicó en la sección 8.4.

8.8. Pruebas con robots en movimiento

Durante el desarrollo de los programas y la interfaz, se trabajó con identificadores estáticos. En la validación de la sección 8.7 se trabajó con identificadores que, aunque no realizaban un movimiento automático con motores, se realizaba un movimiento manual entre iteraciones. Esto para los algoritmos PSO y ACO significaba que se movían entre iteraciones.

Sin embargo, para esto, se tenía que ejecutar el código mediante un botón en la interfaz en cada iteración. La razón principal de esto es que no se tiene una plataforma móvil. Sin embargo, al momento de contar con una plataforma que realice los movimientos del robot constantemente y de manera autónoma, este método de ejecución del código no será cómodo. Por lo que se agregó una opción para mantener la ejecución en bucle. A su vez, también se agregó un botón para detener el bucle de ejecución. Esto permite poder trabajar con robots en movimiento continuo y autónomo sin la necesidad de ejecutar el código manualmente cada vez que se requiera.

CAPÍTULO 9

Resultados

En este capítulo se presentan los resultados de todas las pruebas que se realizaron, tanto de Matlab como de Python.

9.1. Matlab

9.1.1. Generación de identificadores

En la Figura 32 se puede ver la imagen resultante de ejecutar el código con el parámetro de identificador número 155.

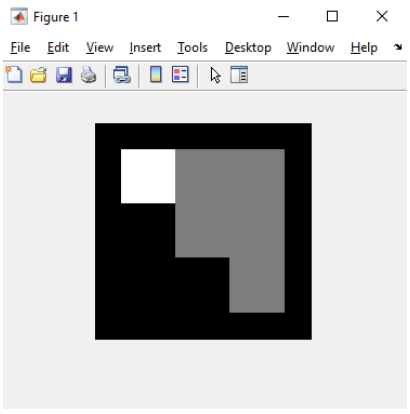


Figura 32: Imagen resultante de generación del código con número 155.

9.1.2. Calibración de cámara

En la Figura 33 se puede ver el ejemplo de una calibración exitosa. Al lado izquierdo se ve la imagen cortada y rotada de manera que únicamente el área de trabajo se observe luego de la transformación, y al lado derecho, se ve la máscara de obstáculos generada; en el caso de la máscara, se tiene en color negro los píxeles en donde se encontraron obstáculos. Esta puede luego utilizarse en otros algoritmos de planificación de trayectorias para evitar dichos obstáculos.

Figura 33: Imagen resultante de la calibración en Matlab.

El anterior fue un resultado exitoso tanto con la luz de la lámpara de la mesa como con las luces del laboratorio apagadas. Pero cuando cualquiera de estas se enciende, se tiene una imagen como la que se observa en la Figura 34. Se puede apreciar que se forman franjas negras en la imagen. Ambas fuentes de luz mencionadas contienen múltiples puntos por los cuales se emite la luz. Luego de realizar una prueba con una única bombilla LED de 12W se consiguió eliminar este patrón. Cabe mencionar que esto, en su mayoría, no resulta en resultados incorrectos de parte de la herramienta, pero al presentar los resultados, esto puede no ser deseado.

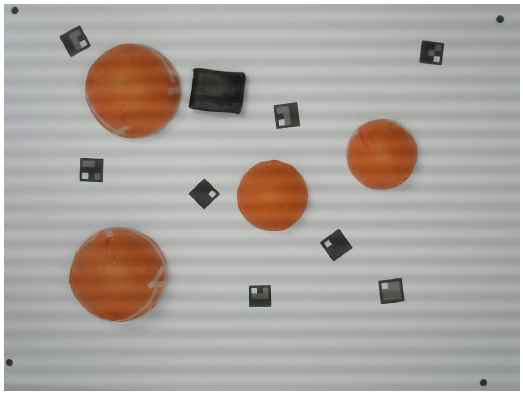


Figura 34: Imagen con franjas formadas por interferencia.

9.1.3. Toma de pose

En la Figura 35 se puede ver la imagen resultante de la toma de pose exitosa. Esta misma figura es modificada en tiempo real si es que se activa la opción en la interfaz. Sobre cada indicador encontrado, se coloca un círculo para denotar el centroide encontrado, y tres textos distintos mostrando el número del identificador encontrado, el ángulo y la posición (x, y) al que se encuentra. La dimensión del ángulo se da en grados y respecto al eje x horizontal (derecho). Las dimensiones de las coordenadas se encuentran en centímetros y respecto a la esquina superior izquierda. Esto ya que Matlab reconoce este como el origen o $(0, 0)$ cm.

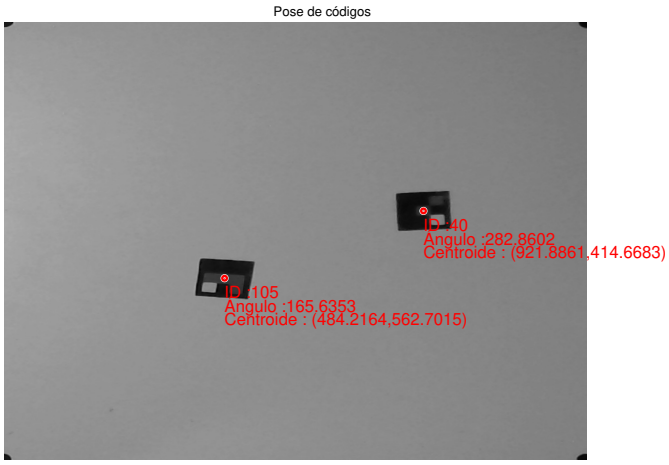


Figura 35: Imagen resultante de la toma de pose en Matlab.

9.1.4. Tiempo de ejecución

En este caso, se utilizaron dos dispositivos distintos para realizar las pruebas. Una computadora de escritorio del laboratorio C118 del edicifio CIT en la UVG y un portátil personal; las especificaciones de ambos dispositivos se encuentran en el Cuadro 1.

	Computadora UVG	**Portátil personal**
Procesador	Intel i7-10700	AMD A10-9600P R5
Memoria RAM (GB)	16.0	16.0
Tarjeta gráfica	Quadro P400	-

Cuadro 1: Cuadro de especificaciones de dispositivos utilizados

Cabe mencionar que para las pruebas con la computadora del CIT no se utilizó la mesa de pruebas como en el portátil, sino que se redujo el área de trabajo a 1/3 aproximadamente. Esto puede que influyera en los resultados.

En ambos dispositivos se tomaron 500 datos de tiempos de ejecución y se realizó un

pequeño análisis estadístico que se puede ver en el Cuadro 2. Es importante mencionar que en todas las iteraciones de estas pruebas se ejecutó el código sin opción de desplegar resultados en una imagen. En el caso del portátil con un único identificador, este tardó en promedio 6.985 s, pero cabe mencionar que esta mantiene ciertos procesos de fondo que consumen muchos recursos, por lo que los tiempos son altos. En cuanto a la computadora de la UVG, los tiempos de ejecución se pueden ver en la gráfica de la Figura 36.

	Media (s)	**Máximo (s)**	**Mínimo (s)**	**Desviación Estándar (s)**
1 Identificador (CIT)	0.1061	0.2358	0.0918	0.0134
1 Identificador (Portátil)	6.9848	8.1487	6.4697	0.2894

Cuadro 2: Cuadro de estadísticas de tiempos de ejecución

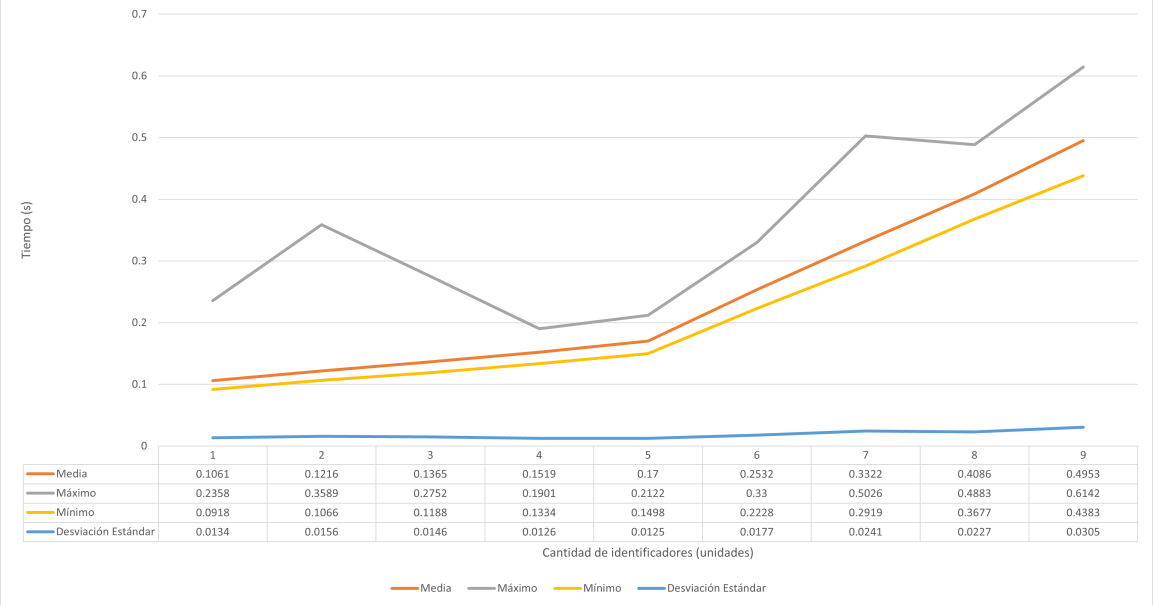


Figura 36: Gráfica de tiempos de ejecución contra cantidad de identificadores.

Se puede ver que el valor promedio aumenta conforme la cantidad de identificadores aumenta. En estas pruebas se llegó a colocar hasta 25 identificadores resultando en un promedio de 0.4953 *seg*. Basado en los resultados, se puede ver un comportamiento lineal tanto en el tiempo promedio como el mínimo. Es evidente que esto puede llegar a ser un problema, especialmente en aplicaciones de robótica de enjambre en donde se puede llegar a tener cientos de agentes.

Sin embargo, algo notable es que aunque el tiempo promedio aumente, la desviación estándar se mantiene casi constante. Por lo que para una aplicación en la que se prefiera una tasa de refresco más constante sin necesidad de tener tiempos muy bajos, esta herramienta sería ideal.

Lo siguiente fue realizar pruebas con los códigos paralelizados. En este caso, antes del bucle principal, se debe de inicializar las instancias de los múltiples trabajadores que realizarán

las tareas de los nodos. Esto se ve limitado por el equipo en el que se realice y la capacidad del procesador. En las pruebas realizadas a continuación se utilizaron 8 trabajadores, aunque se realizaron pruebas con menor cantidad de trabajadores pero, estadísticamente, no existe diferencia en los tiempos de ejecución.

Las primeras veces que se utilizó la herramienta se verificaron los tiempos individuales de cada iteración y se notó que durante las primeras iteraciones luego de inicializar a los trabajadores, los tiempos eran mayores. Luego de alrededor de 10 iteraciones, el tiempo se estabilizaba en un rango. Por esta razón, se realizaron 550 iteraciones, pero para las estadísticas se tomaron en consideración las últimas 500 iteraciones. En la Figura 37 se muestra las gráficas de tiempos conforme se aumentan la cantidad de identificadores en el área de trabajo. En color verde se muestran los tiempos sin paralelización y en color turquesa se muestran los tiempos con paralelización.

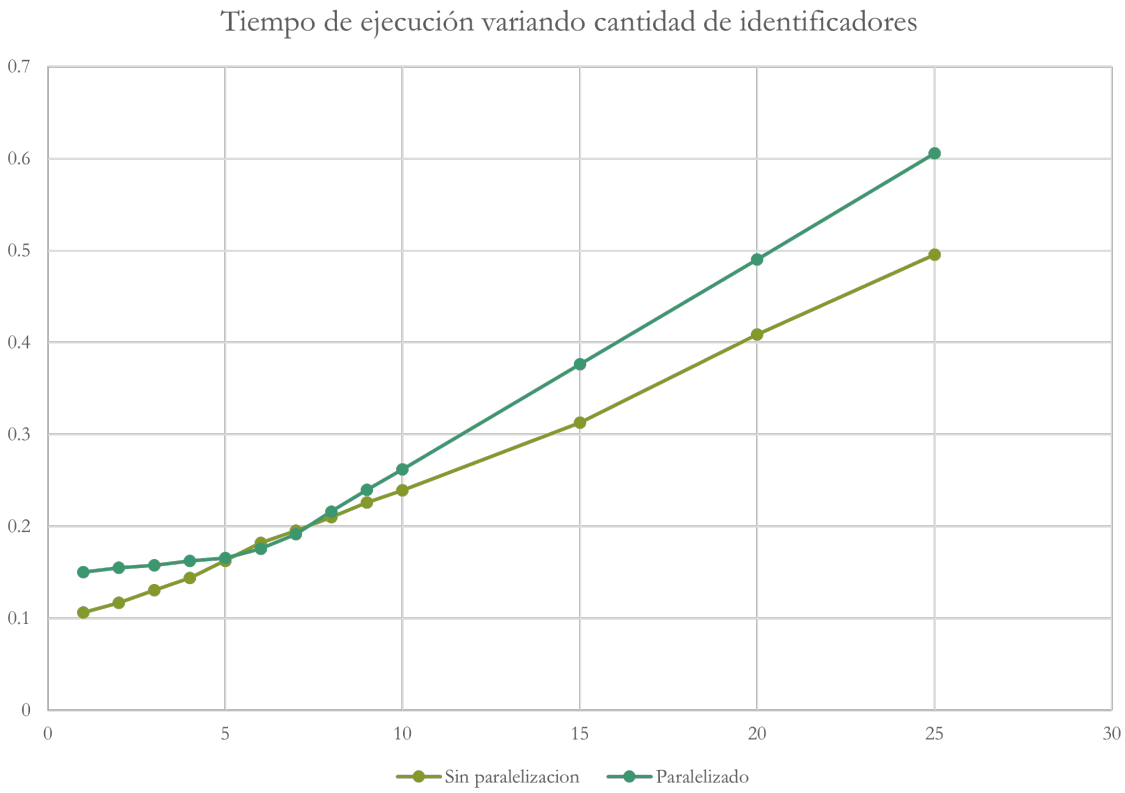


Figura 37: Gráfica de comparación de tiempos entre programación secuencia y multihilos aumentando la cantidad de identificadores.

Es posible ver que entre 5 y 7 identificadores, el tiempo con paralelización implementada es menor a la programación secuencial. Sin embargo, para cantidades fuera de este rango no es el caso. Para una menor cantidad, se considera que el tiempo de creación, destrucción y manejo de hilos es mayor al tiempo ahorrado con la paralelización. Este tiempo se llega a compensar al llegar a 5, pero a partir de 8 identificadores, el tiempo de transmisión de datos hacia los hilos incrementa más rápido que el tiempo ahorrado.

Por lo que es posible obtener una mejora en el tiempo de ejecución, sin embargo para

que esta se mantenga conforme se aumenta la cantidad de identificadores, se necesita una manera de disminuir la cantidad de datos enviados a cada hilo. Una opción es tener una variable global a la que acceda cada hilo en lugar de crear nuevas variables en cada hilo, pero debido a la manera que las funciones de paralelización funcionan en Matlab, no se puede tener una variable global a la que accedan todos los hilos.

Un gran obstáculo para la paralelización efectiva de esta herramienta es que el código resulta ser lineal. Con esto se refiere a que casi todas las operaciones dependen de los resultados de la operación anterior. Esto es más fácil de notar en el diagrama de flujo de la Figura 19 en la que se tiene un flujo en línea "recta" por un único camino. Por lo que estos resultados no implican que sea imposible el disminuir los tiempos de ejecución a través de la paralelización, sino que con las divisiones en hilos seleccionadas para esta implementación no resultó en esto. Es posible que limitando las condiciones de uso o seleccionando otra división en el código para los hilos se pueda llegar a un mejor resultado.

9.1.5. Precisión

En esta prueba, se hicieron mediciones con el programa y se tomaron las mismas mediciones con una cinta métrica marca Stanley para la posición y un transportador marca Artesco. Estas se compararon entre sí para verificar el grado de precisión de las mediciones que resultan en el programa. En el Cuadro 3 se muestran los resultados de dicha prueba.

ID	**Herramienta**			**Medidas teóricas**			**Porcentaje de error (%)**		
	x **(cm)**	y **(cm)**	θ **(°)**	x **(cm)**	y **(cm)**	θ **(°)**	x	y	θ
255	20.5425	10.5684	45.0343	20.7	10.7	45.3	0.761	1.230	0.587
30	82.4219	17.8128	64.7133	82.6	17.7	65	0.216	0.637	0.441
40	84.831	36.2762	63.4974	84.9	36.5	63.1	0.081	0.613	0.630
0	50.0799	43.8914	203.2723	50.5	43.3	201.4	0.832	1.366	0.930
50	26.2968	57.0546	337.4734	26.5	56.5	335	0.767	0.982	0.738

Cuadro 3: Cuadro de resultados de pruebas de precisión en Matlab

Se aprecia que los porcentajes de error para las medidas son todos menores a 1.5 % y los ángulos no sobrepasan el 1 % de error. Estos valores, aunque de por sí son bajos, se pueden mejorar fácilmente. Aparte del posible error durante las mediciones teóricas debido a los instrumentos de medición y el proceso de medición, la mayor fuente de error es la transformación proyectiva.

En la sección 8.2, se menciona que se utilizó una transformación de tipo proyectiva ya que es posible que las esquinas seleccionadas no se encuentren perfectamente perpendiculares entre sí. Durante esta prueba de precisión, no se procuró esto, ya que no todos los usuarios de esta herramienta lo harán. Por lo que de si se toma el tiempo necesario para colocar los marcadores de mejor manera, los resultados podrían mejorar.

9.1.6. Validación de concepto

Como mencionado en la sección 8.7, las siguientes pruebas se realizaron en conjunto con los trabajos de graduación [26] para el algoritmo PSO y [27] para el algoritmo ACO.

Algoritmo PSO

Para realizar esta validación se hizo que ambas herramientas (Visión por computadora y algoritmo PSO) funcionaran conjuntamente. En esta se colocaron cuatro identificadores en posiciones cercanas al origen (esquina superior izquierda). Ya que esta fue la primera prueba, se quiso hacer sencilla y rápida únicamente para comprobar qué errores o dificultades se podrían tener con esta metodología.

Esta primera prueba llevó un total de 30 iteraciones para que todos los identificadores lograran converger al origen. Entre cada iteración se realizaba una toma de pose y los resultados luego se envían a sus respectivas direcciones IP. En la Figura 38 se puede ver la captura de las consolas de los agentes final. En esta se puede ver que según las posiciones de la herramienta eran proveídas, el agente realizaba la iteración siguiente de su algoritmo. La pose recibida por el agente se despliega con el formato "Coordenadas agente n son: x, y, θ", donde n es el número del agente, y x, y, θ son los valores de la pose del agente en ese instante. En la Figura 39 se muestra la posición inicial de los identificadores.

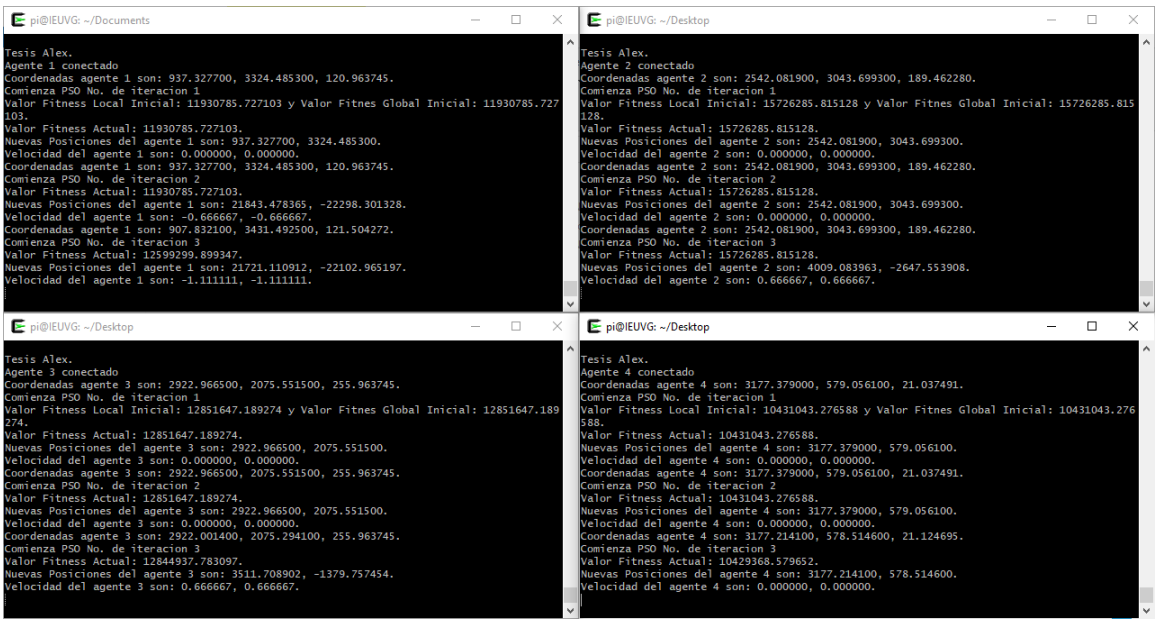


Figura 38: Captura de consolas finales de agentes (Prueba inicial PSO).

Figura 39: Imagen de posiciones iniciales de agentes (Prueba inicial PSO).

En la Figura 40 se puede ver la imagen con las posiciones finales. Es de notar que algunos de ellos se encontraban parcialmente fuera del área de trabajo o superpuestos entre sí. Esto causaba conflicto al momento de procesar la imagen, por lo que no se lograba detectar de manera correcta esos identificadores. Por esta misma razón, en la Figura 38 se puede notar que el número de iteración final no es el mismo en todas; al no poder detectar los identificadores, no se lograba actualizar el agente.

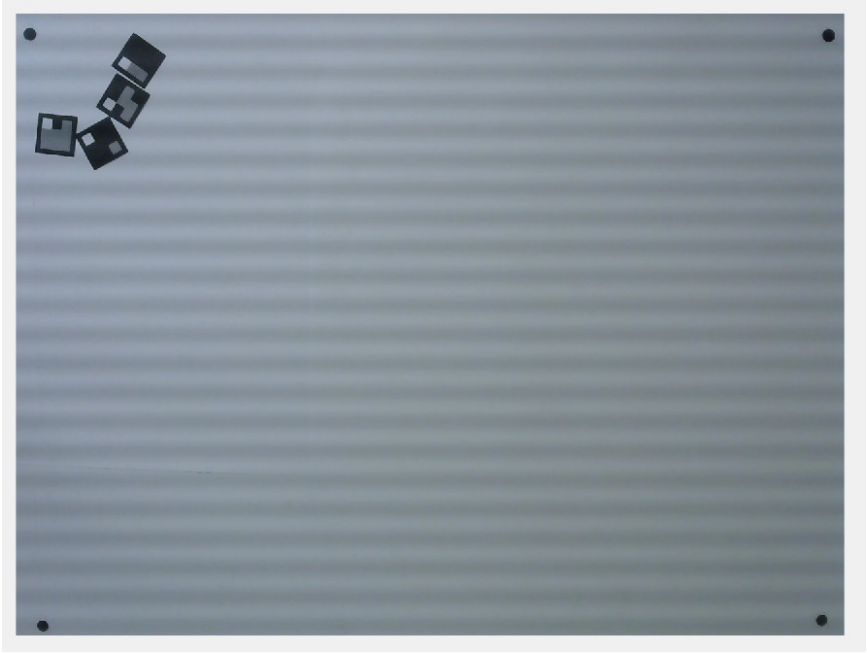


Figura 40: Imagen de posiciones finales de agentes (Prueba inicial PSO).

Es importante mencionar que aunque la comunicación entre la herramienta y los agentes fue exitosa y estos lograron ejecutar su algoritmo en todo momento, este no fue el caso de detección de los identificadores. Esto se dio en dos casos: cuando el identificador se encontraba parcialmente fuera del área de trabajo y cuando dos identificadores se superponían. En el primer caso, el filtro de bordes no logra detectar un borde en la parte del identificador que se encuentra fuera, por lo que el contorno lo logra cerrarse y rellenarse y es eliminado por el código. En el segundo caso se debe a que ambos se detectan como un único contorno, lo que causa que se no identifiquen por separado y el dato resultante sea incorrecto.

Tomando en consideración estas limitantes en consideración, para la segunda prueba se decidió mover el origen del sistema de coordenadas hacia el centro del área de trabajo. De esta manera, los identificadores no se saldrían del área de trabajo ya que el punto de meta será el centro. También se separaron los marcadores uniformemente para evitar que se superpongan entre sí. En las Figuras 41, 42, 43 y 44 se muestra una serie de imágenes que demuestran el movimiento que se tuvo de los agentes.

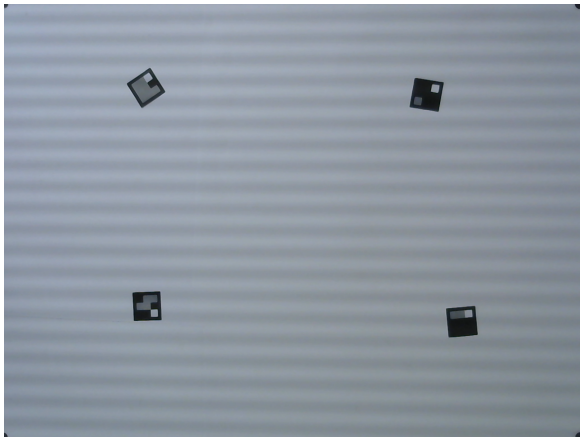


Figura 41: Imagen de posiciones iniciales de agentes (Prueba final PSO).

Figura 42: Imagen de posiciones en iteración 25 de agentes (Prueba final PSO).

Figura 43: Imagen de posiciones en iteración 64 de agentes (Prueba final PSO).

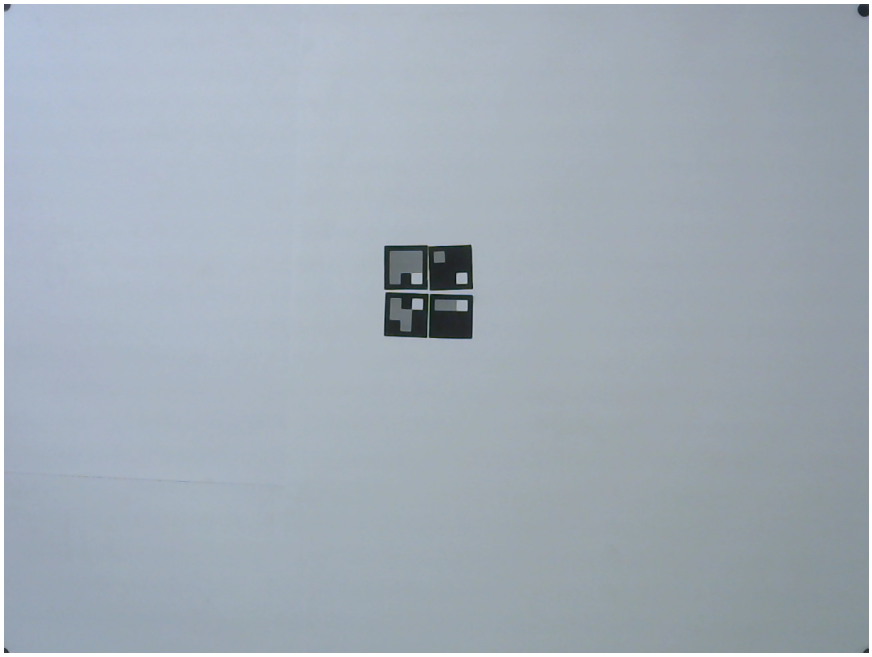


Figura 44: Imagen de posiciones finales de agentes (Prueba final PSO).

Para esta prueba, se hicieron 79 iteraciones en total. Los cambios de posición se realizaron moviendo los marcadores de manera manual. Y cabe mencionar que el cambio en las posiciones no se realizó con ningún método de medición. Esto ya que el propósito de esta prueba no era medir la exactitud del algoritmo PSO.

Algoritmo ACO

De la misma manera que para el algoritmo PSO, se hizo que las herramientas (en este caso Visión por Computadora y algoritmo ACO) funcionaran en conjunto. Primero se debe hacer mención que en todas las imágenes de esta sección se marcará con color azul el nodo destino, con color rojo el nodo de origen y con color verde la ruta calculada por el algoritmo ACO. En este caso se comenzó validando con el caso más sencillo en el que no existen obstáculos y la ruta óptima es una diagonal. En las Figuras 45, 46 y 47 se muestran capturas del proceso de movimiento del agente.

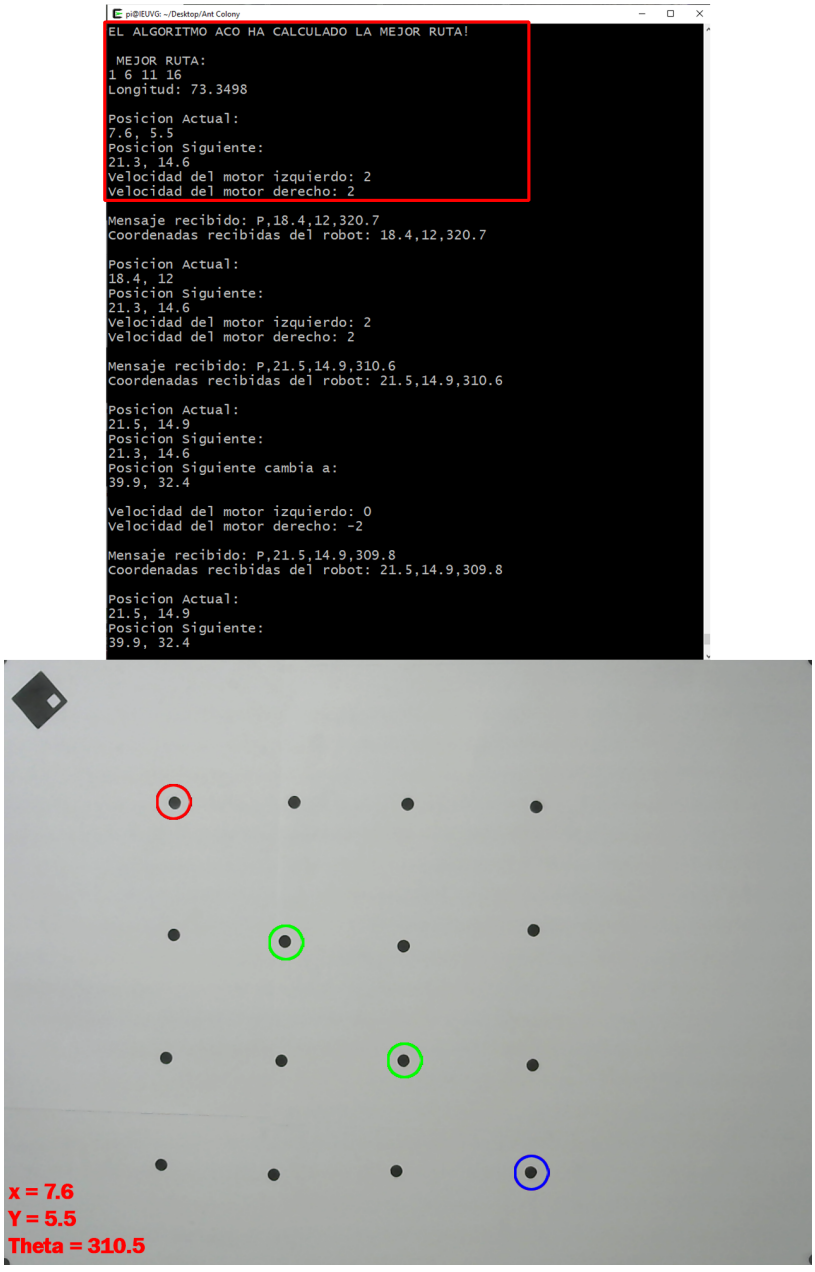


Figura 45: Imagen de posiciones iniciales de agentes (Prueba inicial ACO).

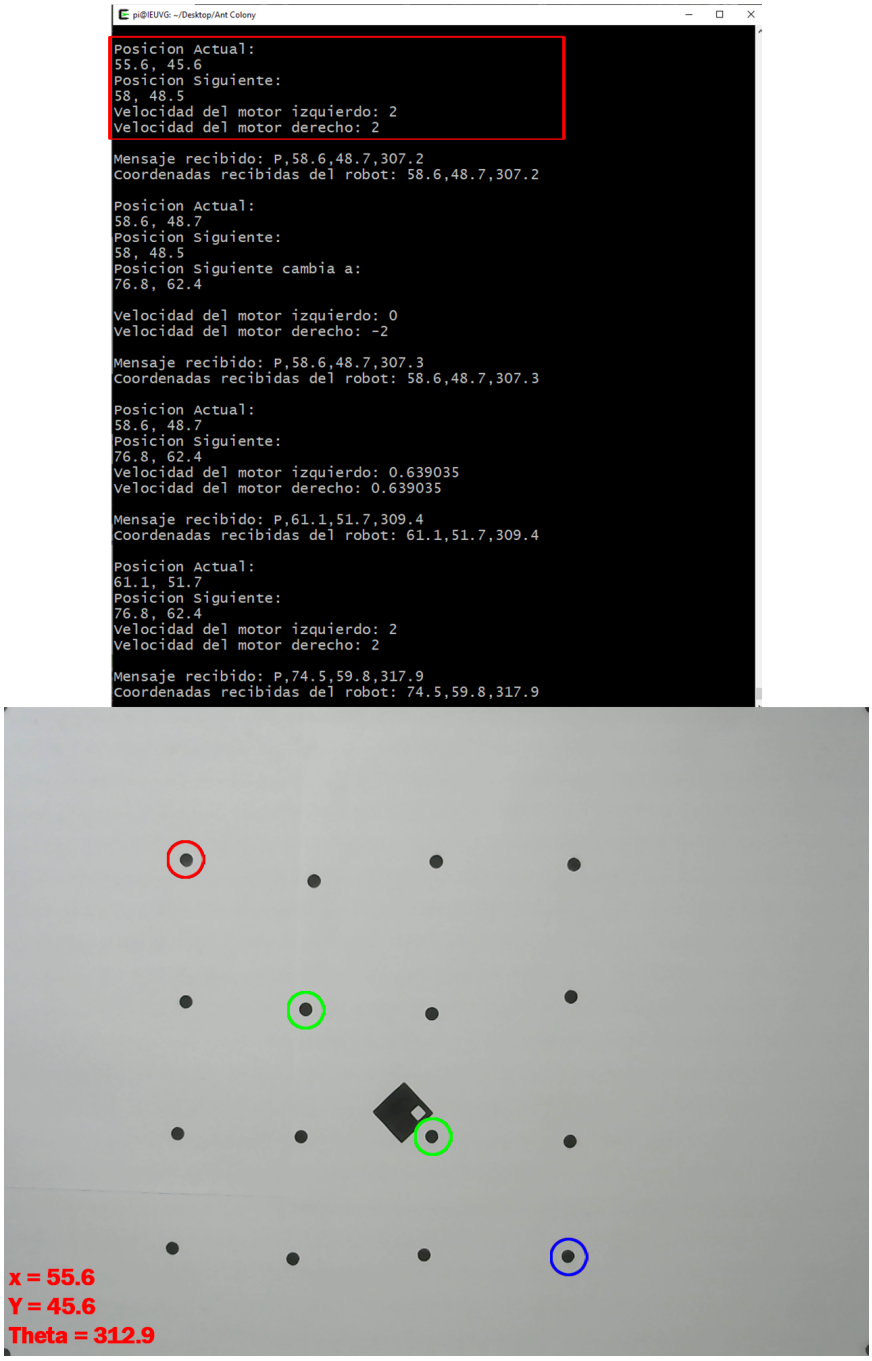


Figura 46: Imagen de posiciones intermedias de agentes (Prueba inicial ACO).

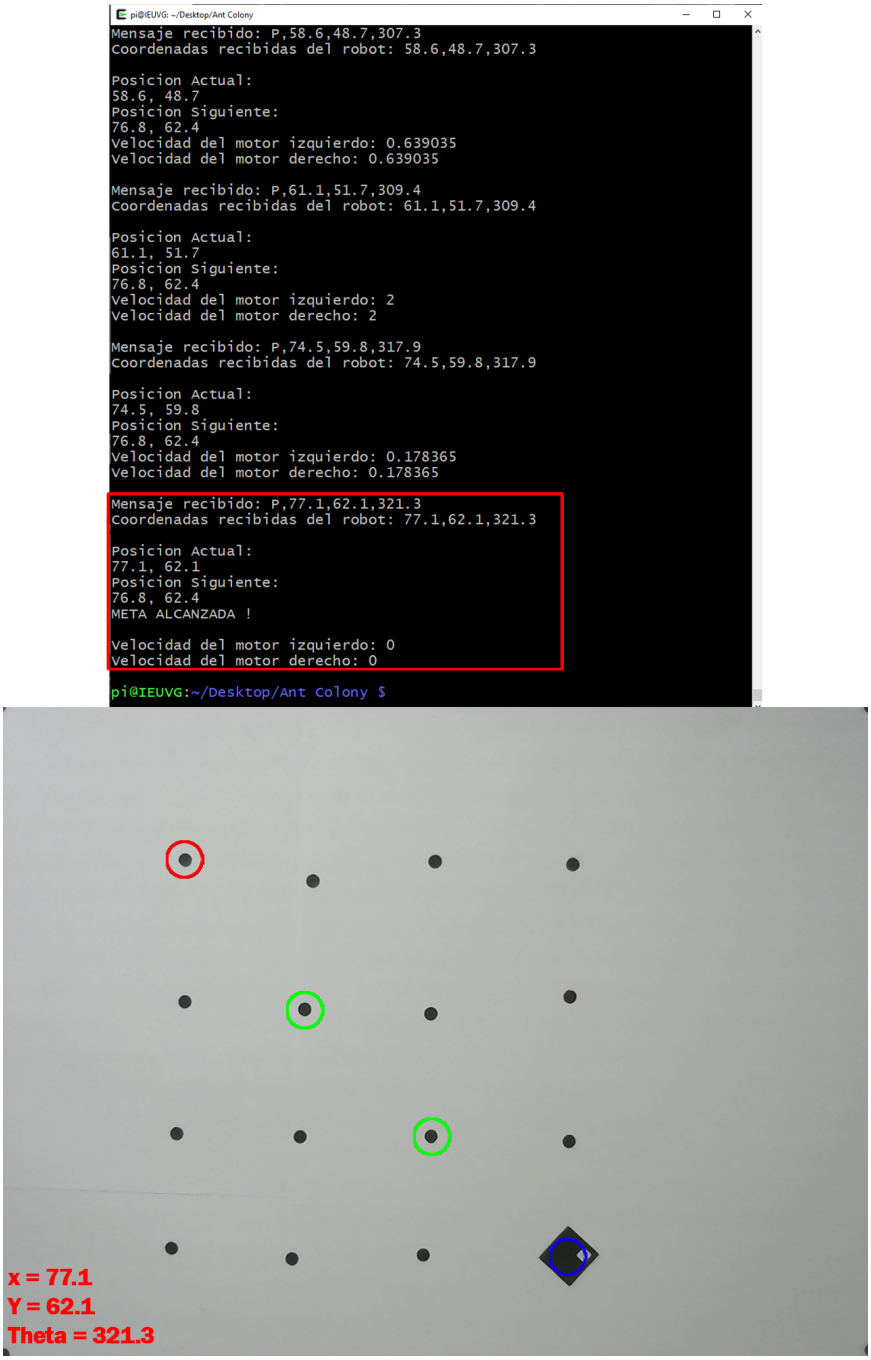


Figura 47: Imagen de posiciones finales de agentes (Prueba inicial ACO).

En estas imágenes también se muestra una captura de la consola y se resalta la iteración actual. Se puede ver que existe una correspondencia entre lo que se detecta con la herramienta y lo que recibe el agente. También que el algoritmo logra cambiar de nodo objetivo conforme se acerca al objetivo actual, siguiendo la trayectoria mostrada. Lo siguiente fue realizar una serie de pruebas similares pero cambiando los nodos iniciales y finales para formar distintas trayectorias. No se incluirán imágenes de estas pruebas ya que son similares a las ya mostradas únicamente cambiando la trayectoria recorrida.

Teniendo validada la integridad de la transmisión de datos, la reconstrucción del mapa, las conexiones y el movimiento del agente, se procedió a realizar pruebas con obstáculos. Se comenzó con un caso simple, comenzando con la base de la prueba demostrada con anterioridad, pero para evitar que siga esta ruta diagonal, se interrumpe una de las conexiones. Esto forzará al agente a rodear esa conexión. En las Figuras 48, 49 y 50 se muestra la serie de imágenes del movimiento realizado por el agente. En este caso se utilizó un celular como obstáculo como primer objeto.

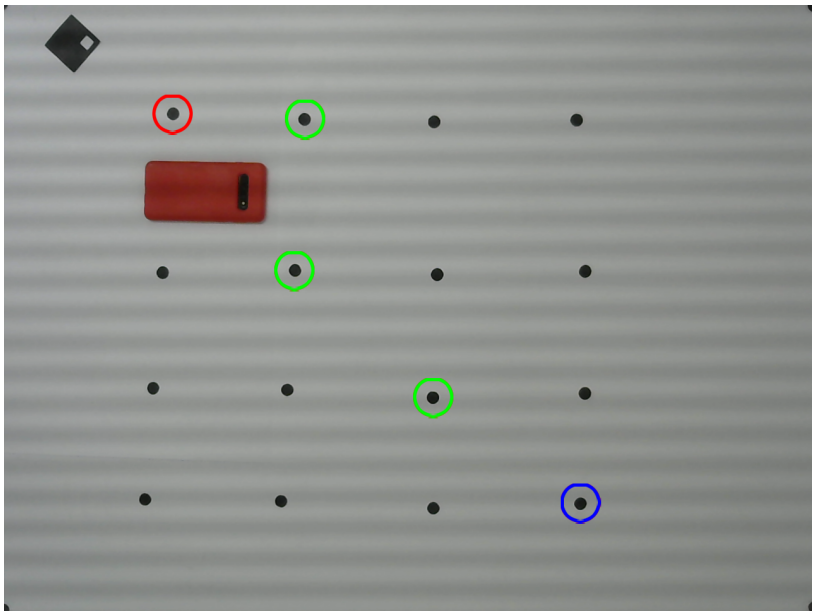


Figura 48: Imagen de posiciones iniciales de agentes (Prueba inicial de obstáculos ACO).

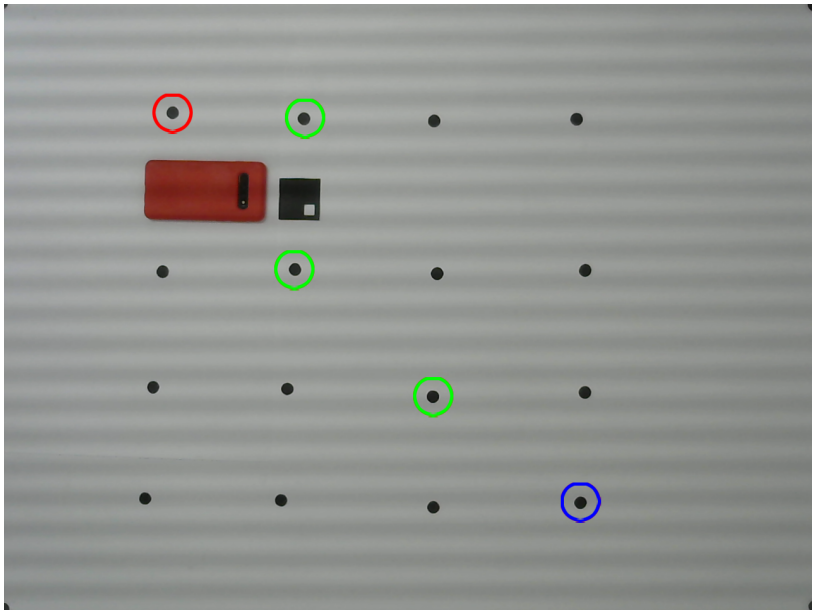


Figura 49: Imagen de posiciones intermedias de agentes (Prueba inicial de obstáculos ACO).

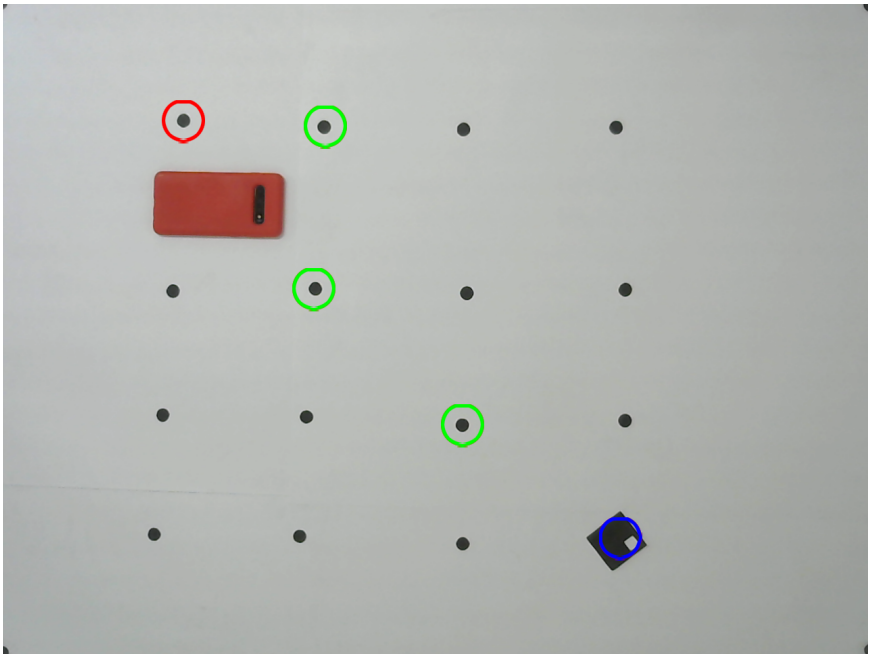


Figura 50: Imagen de posiciones finales de agentes (Prueba inicial de obstáculos ACO).

Con esta prueba se confirmó que las conexiones obstruidas por el obstáculo no se realizaron. Por lo que se procedió a hacer una prueba con mayor cantidad de obstáculos de manera que únicamente existiera una ruta posible procurando que esta no fuera tan sencilla como las pruebas anteriores. Ya que se requerían múltiples obstáculos para eliminar conexiones, se imprimieron rectángulos color negro que abarcan un área más grande. En las Figuras 51, 52 y 53 se muestra la secuencia de imágenes del movimiento del agente.

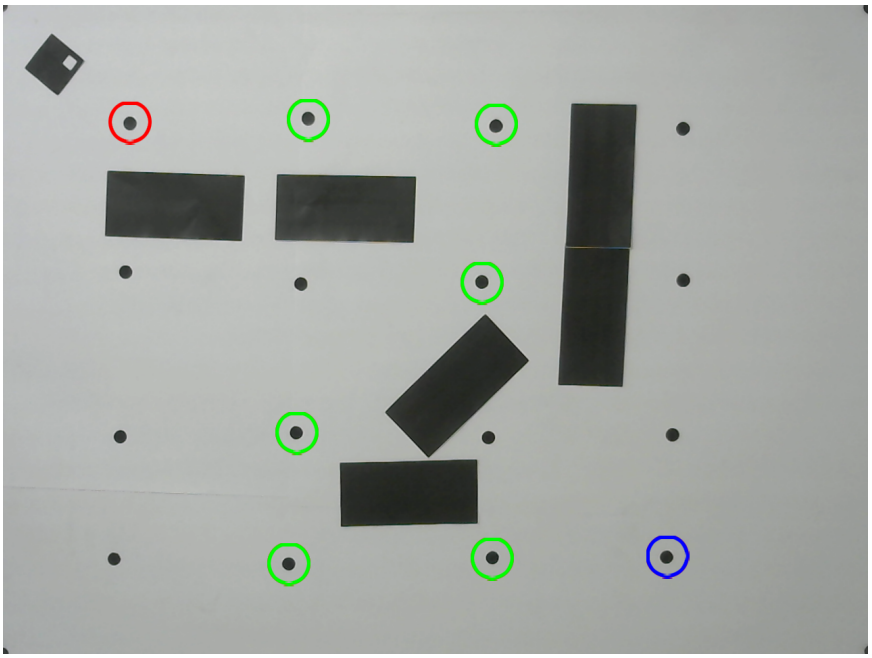


Figura 51: Imagen de posiciones iniciales de agentes (Ruta limitada por múltiples obstáculos).

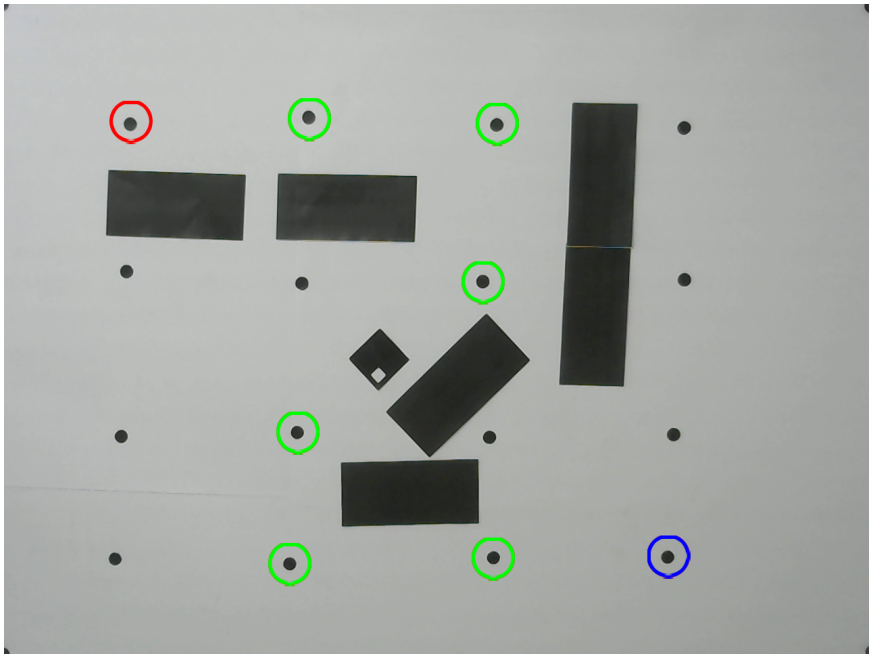


Figura 52: Imagen de posiciones intermedias de agentes (Ruta limitada por múltiples obstáculos).

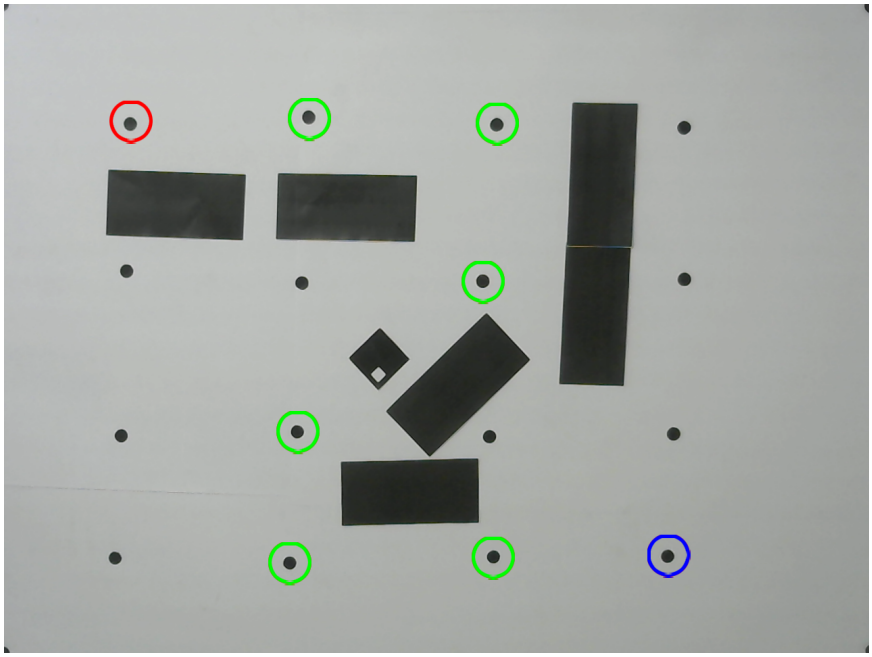


Figura 53: Imagen de posiciones finales de agentes (Ruta limitada por múltiples obstáculos).

Por último, se realizó una prueba en la que se tenían cuatro posibles rutas que el agente podía elegir. No todas tenían la misma distancia a recorrer, por lo que el algoritmo debía de seleccionar la óptima. En las Figuras 54, 55 y 56 se observa en color verde la ruta que el algoritmo eligió, y en otros colores las demás rutas que se podían elegir.

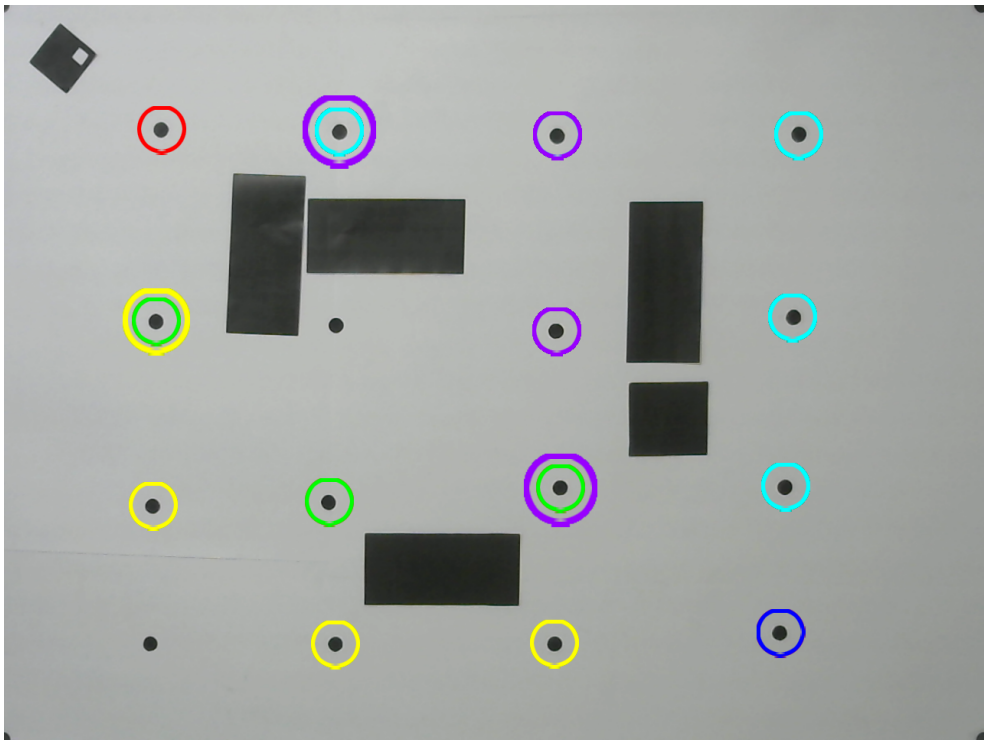


Figura 54: Imagen de posiciones iniciales de agentes (Múltiples rutas disponibles).

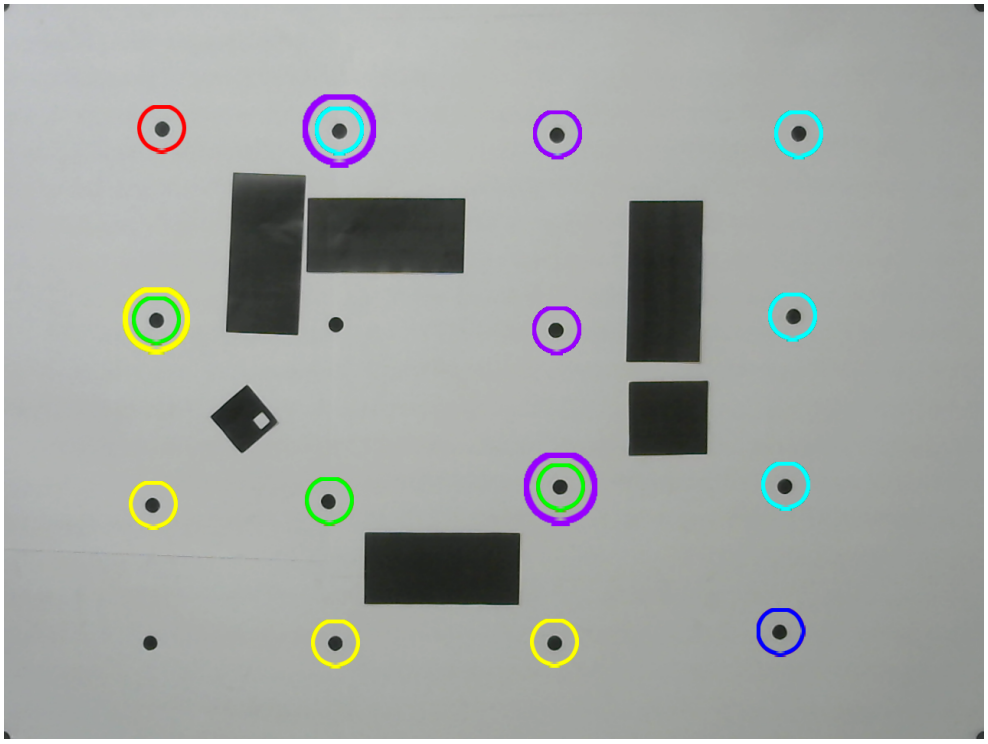


Figura 55: Imagen de posiciones intermedias de agentes (Múltiples rutas disponibles).

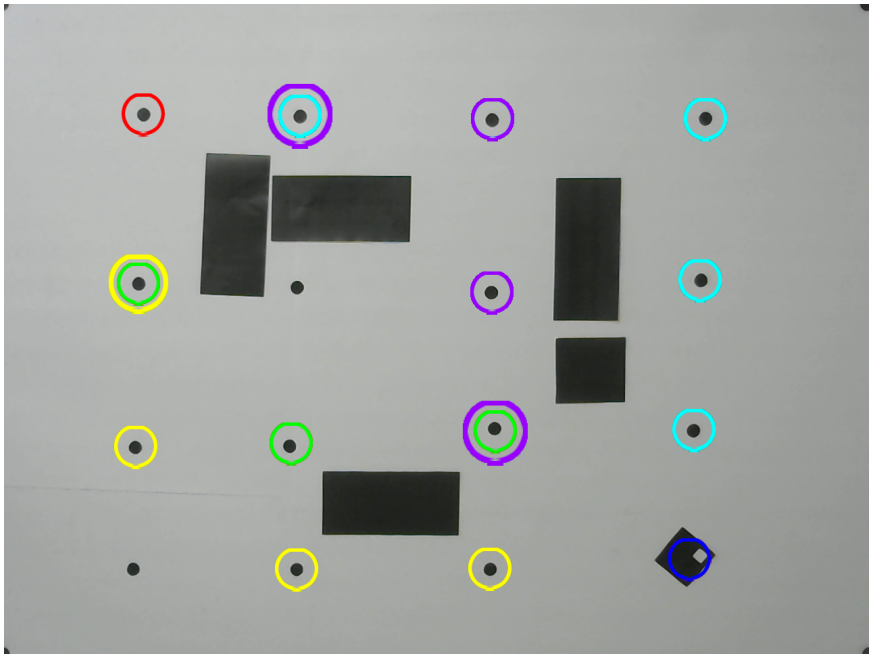


Figura 56: Imagen de posiciones finales de agentes (Múltiples rutas disponibles).

Es fácil de reconocer que el agente efectivamente tomó la ruta más corta para llegar a su meta. Por lo que se puede decir que al momento de realizar la integración, tanto el mapa, las conexiones, la detección de pose del agente, la detección de obstáculos estáticos y el algoritmo ACO siguen funcionando de manera correcta.

9.1.7. Pruebas con robots en movimiento

Para realizar esta validación, se utilizó un área de aproximadamente 67×50 cm. Para la calibración no se incluyeron obstáculos aunque esto no debería de ser problema para la herramienta. Luego de la calibración, se colocaron 6 identificadores en el área de trabajo. Y por último, se ejecutó la herramienta en bucle.

Esta prueba se grabó en un vídeo que se colocó en una carpeta compartida en el siguiente enlace. En este vídeo se puede ver una figura de Matlab en la que se despliegan los resultados de pose. Y se observa como se realiza el movimiento de los identificadores. Como antes mencionado, estos movimientos se realizaron con la mano pero esto de igual manera demuestra un movimiento en los identificadores.

Algo importante a resaltar es que los tiempos de ejecución aumentan ligeramente en cada iteración. Esto únicamente ocurre si se despliega la imagen con los resultados. Esto se verificó realizando la misma prueba en bucle pero únicamente guardando los tiempos de ejecución sin mostrar los resultados.

Cuando se tiene la herramienta en bucle, luego de 300 iteraciones, el tiempo medio de ejecución es de 0.2115 seg, con desviación estándar de 0.0420 seg y revisando manualmente los resultados, los incrementos son de 0.001 seg por iteración aproximadamente. Esto resulta

en una diferencia entre el último valor y el primero de 0.1716 *seg*. Por el otro lado, si no se despliegan los resultados, se tiene una media de 0.1068 *seg*, desviación estandar de 0.0212 *seg*, diferencia entre extremos de 0.0116. Al revisar los datos individuales, no se nota ningún aumento constante entre iteraciones.

De los resultados anteriores se puede ver que el despliegue de los resultados duplica el tiempo medio de ejecución y que los valores aumenta progresivamente. Sin embargo, si se desea una frecuencia de muestreo más constante, se puede utilizar la herramienta sin mostrar los resultados y únicamente transmitir los datos o guardarlos en un archivo para el uso en otro programa.

9.2. Python

Este proyecto de investigación comenzó en el 2019 con la tesis de André Rodas [4]. En esta se desarrolló la herramienta en el lenguaje de C y se utilizó la mesa de pruebas que se encuentra en la UVG para comprobar y desarrollarla. Lamentablemente, debido a la pandemia del Covid-19, la herramienta en Python no se pudo comprobar en esta mesa, sino se utilizó un prototipo de mucho menor tamaño. Por esta razón, en la tesis [5] se recomendaba verificar esta en la mesa UVG en caso que el aumento de tamaño generar algún inconveniente. Por esta razón, uno de los objetivos de este trabajo fue realizar estas pruebas.

Se replicaron las pruebas que se realizaron en [5] pero en la mesa de pruebas de la UVG. No se realizó ningún cambio a la herramienta existente para las siguientes pruebas, por lo que los resultados de la toma de pose se muestran en la terminal de ejecución del programa en lugar de en la interfaz.

9.2.1. Calibración

En la Figura 57, se puede ver un ejemplo de la calibración teniendo un marcador circular extra en el interior del área (encerrada con color rojo para para facilitar la distinción). El resultado es una calibración tomando en consideración las esquinas adecuadas.

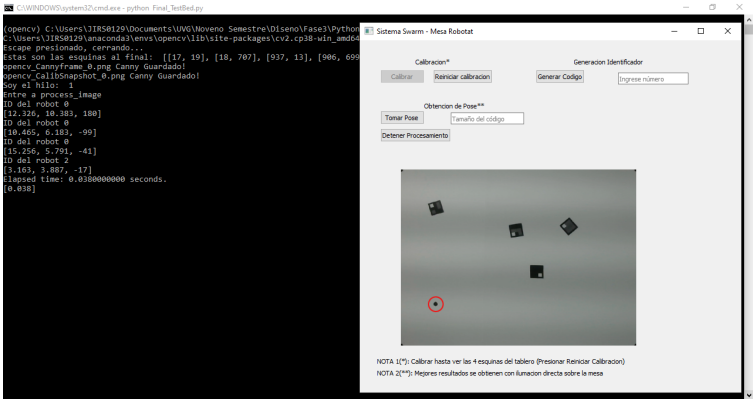


Figura 57: Imagen resultante de la toma de pose en Python con marcador circular interno extra.

En la Figura 58 se realizó un experimento similar pero esta vez, se colocó el marcador circular en el exterior del área de trabajo. En el trabajo de Guerra, se menciona que este puede llegar a ser un problema por la manera en que se seleccionan las esquinas de manera que las más cercanas a las esquina de la imagen original son seleccionadas. En la Figura 59 se puede ver el resultado. A primera vista se puede pensar que es correcta, pero en realidad, se seleccionó el marcador circular externo en lugar del interno. Esto se puede ver ya que en el resultado se puede apreciar un marcador circular interno (que debió ser seleccionada pero no lo fue); aparte, los identificadores se encuentran deformados debido a que las esquinas seleccionadas no se encuentran de manera perpendiculares.

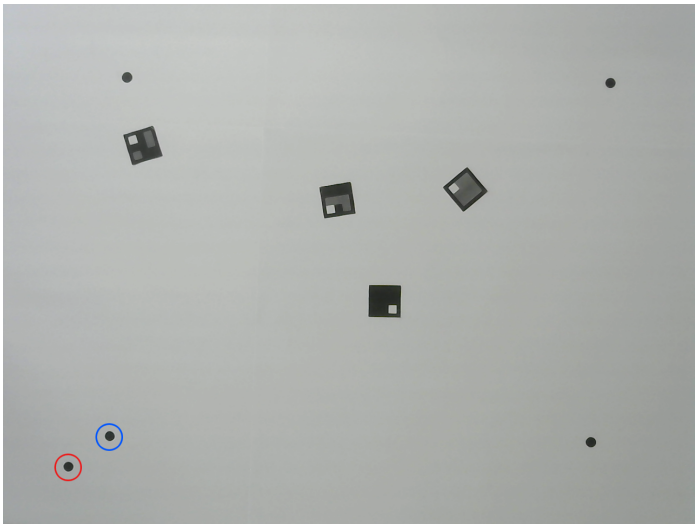


Figura 58: Imagen resultante de la toma de pose en Python con marcador circular interno extra sin calibrar.

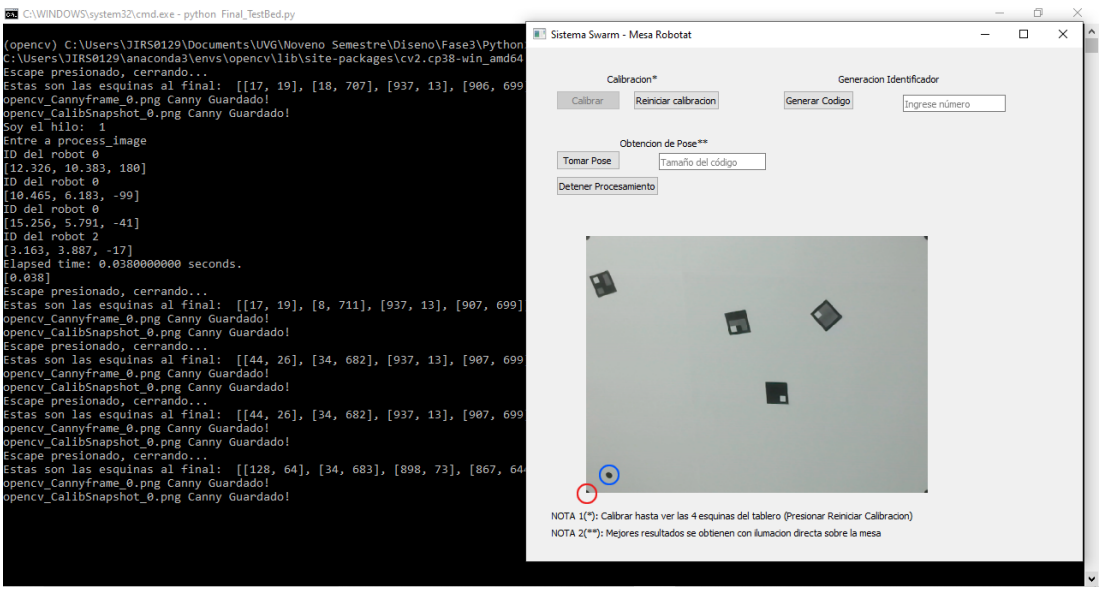


Figura 59: Imagen resultante de la toma de pose en Python con marcador circular externo extra calibrado.

En la Figura 60 se puede ver un ejemplo de una toma de pose exitosa. Se agregaron manualmente los valores de los códigos que se esperaban en la imagen de la interfaz. En el cuadrado rojo, se encerraron los resultados. Estos son el ID del robot y debajo un vector de números en el formato $[X, Y, \theta]$. Se puede ver que reconoce todos los identificadores con los valores de ID que corresponden y valores de posición y ángulo aproximadamente correctos.

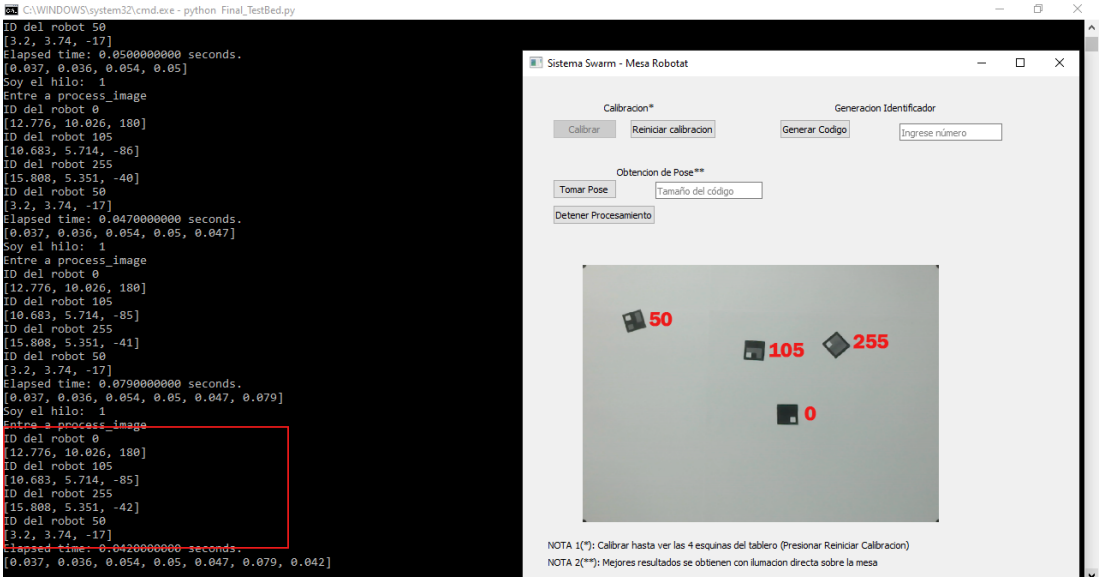


Figura 60: Imagen resultante de la toma de pose en Python.

Luego, se agregó un obstáculo al igual de los de Matlab y en la Figura 61 se puede ver que los resultados siguen siendo los correctos y no toma en cuenta el obstáculo en ningún momento.

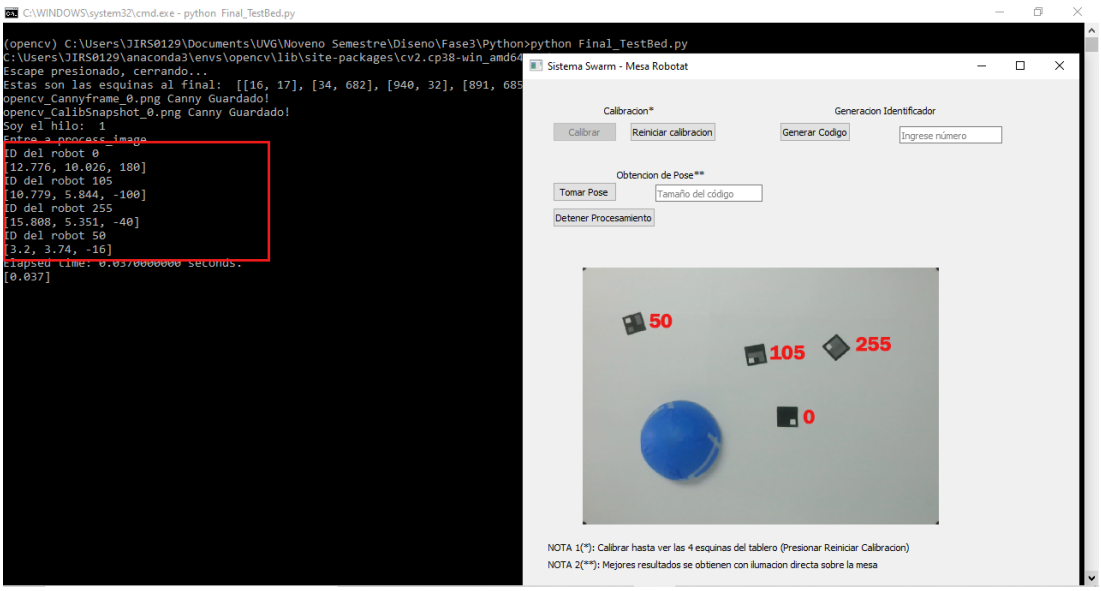


Figura 61: Imagen resultante de la toma de pose en Python con un obstáculo.

Las pruebas anteriores, al igual que se hizo mención en los resultados de Matlab, se iluminó con una lámpara con una única bombilla LED de 12W. En la Figura 62 se muestra un resultado del intento de toma de pose con las lámparas del laboratorio encendidas. De nuevo, se pueden ver las franjas negras que causan problemas en el programa. Esto nos ayuda a verificar que el problema en realidad es parte del equipo utilizado y no de la implementación en código de Matlab. En la Figura 63 se muestra un intento con la lámpara de la mesa encendida. En esta se puede ver que las franjas se encuentra más resaltadas aún.

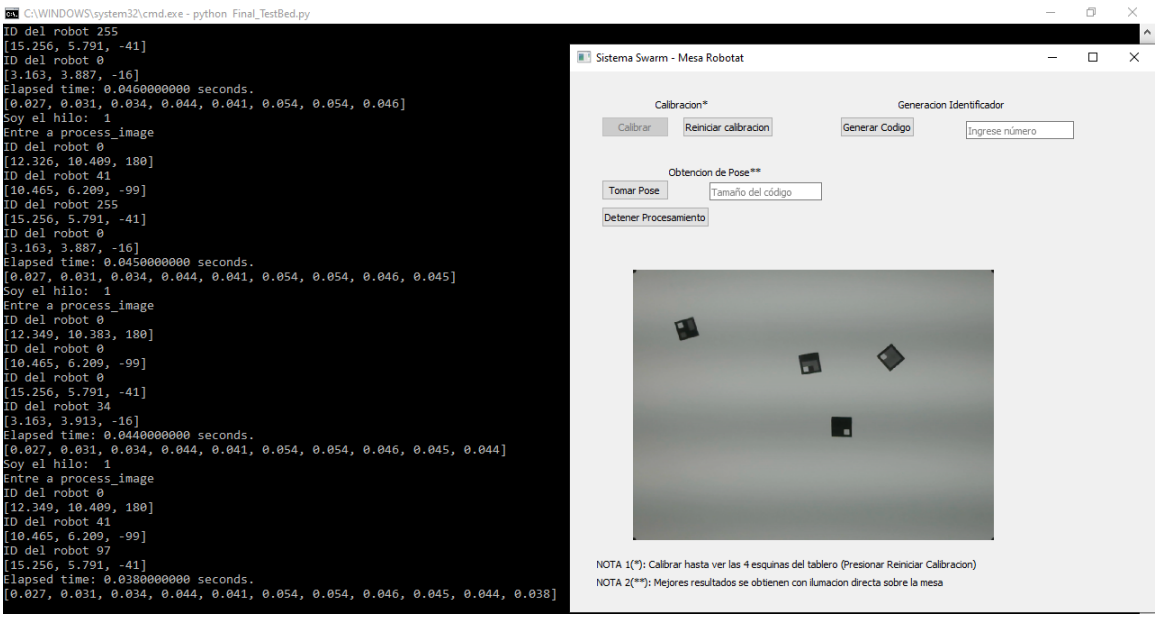


Figura 62: Imagen resultante de la toma de pose en Python con lámparas del laboratorio encendidas.

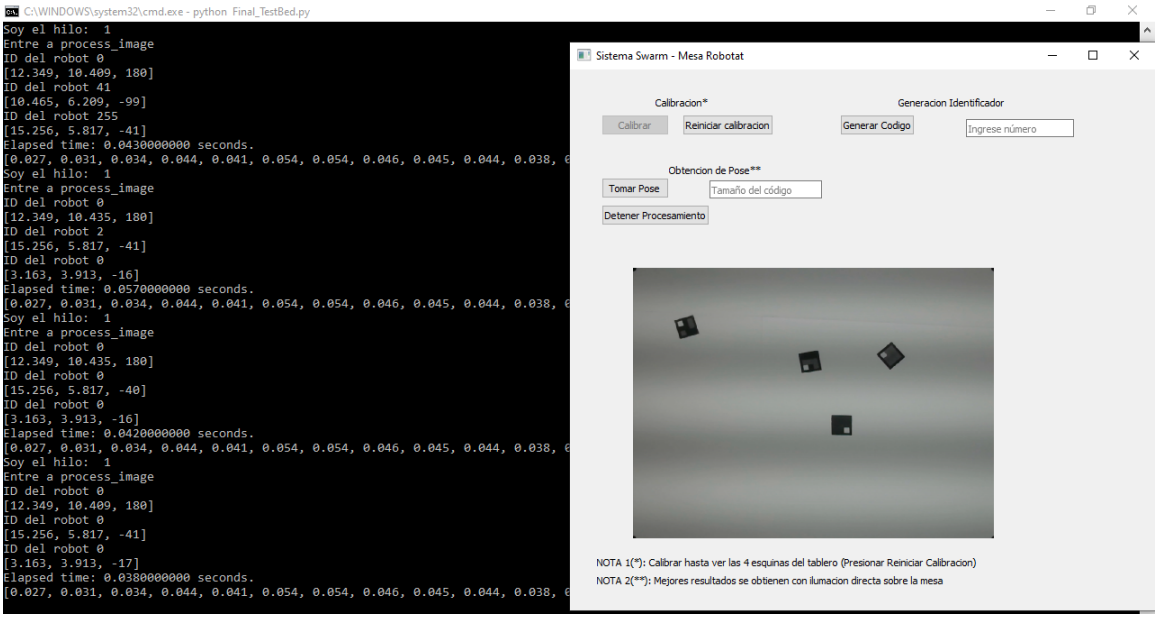


Figura 63: Imagen resultante de la toma de pose en Python con lámparas de la mesa encendidas.

9.2.2. Precisión

Similar a lo hecho en la subsección 9.1.5, se realizó una prueba de precisión con los resultados dados por la herramienta en Python. Estos luego se midieron con una cinta métrica marca Stanley para la posición y un transportador marca Artesco. Los resultados se presentan en el Cuadro 4.

ID	**Herramienta**			**Medidas teóricas**			**Porcentaje de error (%)**		
	x **(cm)**	y **(cm)**	θ **(°)**	x **(cm)**	y **(cm)**	θ **(°)**	x	y	θ
50	19.071	51.441	87	19	52.2	86.5	0.374 %	1.454 %	0.578 %
0	35.147	42.933	244	34.6	42.8	244.8	1.581 %	0.311 %	0.327 %
40	70.293	35.316	27	70.8	35.7	27.4	0.716 %	1.076 %	1.460 %
30	67.797	17.213	25	67.5	17.7	25.3	0.440 %	2.751 %	1.186 %
255	20.669	10.09	43	20.8	10.8	42.3	0.630 %	6.574 %	1.655 %

Cuadro 4: Cuadro de resultados de pruebas de precisión en Python

Se puede notar que los porcentajes de error en las medidas son menores a 2.8 % y menor a 1.7 % para los ángulos. Similar a los resultados de la subsección 9.1.5, estos se pueden mejorar de gran manera tomando el tiempo de colocar los marcadores circulares con mayor precisión.

CAPÍTULO 10

Conclusiones

1. Se realizó la migración de los algoritmos de visión por computadora desarrollados en fases previas al lenguaje de Matlab.

2. En el proceso de desarrollo se agregaron nuevas funcionalidades. Entre las más importantes se encuentran la detección de obstáculos, la detección de nodos para creación de mapas utilizados en algoritmos de robótica de enjambre, y la transmisión de datos entre dispositivos usando protocolos de red (UDP y TCP).

3. Es posible realizar el mapeo del área de trabajo detectando obstáculos estáticos sobre la misma sin importar la forma que tengan, siempre y cuando creen el suficiente contraste de color con la superficie de la mesa.

4. De manera similar a los resultados en [5], al paralelizar los algoritmos de visión por computadora se determinó que el tiempo de *overhead* supera al tiempo ahorrado en el procesamiento de datos.

5. Es posible realizar la detección de identificadores en movimiento con tasas de refresco moderadas.

6. Los tiempos de ejecución no se ven afectados de gran manera al aumentar el número de identificadores en el área de trabajo. Sin embargo, debido a que para aplicaciones de robótica de enjambre se puede llegar a tener grandes cantidades de identificadores, esta diferencia puede acumularse y llegar a ser significativa.

7. La herramienta en Python funciona de la misma manera en la mesa UVG que en las pruebas realizadas en la fase anterior.

8. Al aumentar el tamaño del área de trabajo en la herramienta de Python, el error aumenta a un máximo de 2.8 % para posiciones y 1.7 % para ángulos.

9. El error porcentual máximo de la herramienta de Matlab es de 1.5 % para posiciones y 1 % para ángulos.

10. La herramienta logró ser utilizada y validada con aplicaciones simples de robótica de enjambre bajo condiciones controladas.

11. Utilizar lámparas con múltiples fuentes de iluminación puede resultar en patrones de interferencia en las imágenes. Esto puede resultar en reconocimiento errónea del número de los identificadores.

12. La herramienta debe de ser calibrada según las condiciones en las que se le dará uso, esto puede resultar en un error en el número de los identificadores.

CAPÍTULO 11

Recomendaciones

1. De manera similar a las pruebas de integración que se realizaron con proyectos utilizando algoritmos PSO y ACO, realizar integración con otros algoritmos o variantes de estos.

2. Dar seguimiento a la integración con los algoritmos PSO y ACO con una plataforma móvil.

3. Plantear una implementación distinta de la paralelización del código.

4. Modificar la detección de obstáculos para tener un sistema completamente dinámico que pueda responder a cambios en los obstáculos. Esto puede realizarse detectando cuadrados y otras formas para distinguir los marcadores del resto, de manera similar a la que se hizo para los nodos de los mapas utilizados en los algoritmos ACO.

5. Tener opción de eliminar conexiones en los mapas de algoritmos ACO sin necesidad de colocar un obstáculo entre los nodos, permitiendo realizar mapas de conexiones más complejos sin alterar el terreno al incluir obstáculos extra. En las pruebas realizadas en este trabajo se necesitaba de obstáculos para eliminar conexiones porque no existe una opción de eliminar una conexión sin esto. Para esto se puede utilizar un sistema de colores para realizar las conexiones en los mapas ACO.

CAPÍTULO 12

Bibliografía

[1] P. Prasanna, K. Dana, N. Gucunski y B. Basily, "Computer-vision based crack detection and analysis," en *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2012*, M. Tomizuka, C.-B. Yun y J. P. Lynch, eds., International Society for Optics y Photonics, vol. 8345, SPIE, 2012, págs. 1143-1148. DOI: 10.1117/12.915384. dirección: https://doi.org/10.1117/12.915384.

[2] E. Bermejo, O. Deniz, G. Bueno y R. Sukthankar, "Violence Detection in Video Using Computer Vision Techniques," *Computer Analysis of Images and Patterns*, vol. 6855, págs. 332-339, 2011.

[3] J. Lizarazo y M. Ramos, "Visión artificial y comunicación en robots cooperativos omnidireccionales," *Ingeciencia*, vol. 1, n.º 1, págs. 5-11, 2016. dirección: http://editorial.ucentral.edu.co/ojs_uc/index.php/Ingeciencia/article/view/159.

[4] A. Rodas, "Desarrollo e implementación de algoritmo de visión por computadora en una mesa de pruebas para la experimentación con micro-robots móviles en robótica de enjambre," Tesis de licenciatura, Universidad del Valle de Guatemala, 2019.

[5] J. Guerra, "Algoritmos de Visión por Computadora para el Reconocimiento de la Pose de Agentes Empleando Programación Orientada a Objetos y Multihilos," Tesis de licenciatura, Universidad del Valle de Guatemala, 2020.

[6] RobotPlatform, *Type of Robot Sensors*, http://www.robotplatform.com/knowledge/sensors/types_of_robot_sensors.html, Accesado: 2021-08-07.

[7] B. Siciliano, L. Sciavicco, L. Villani y G. Oriolo, *Robotics: Modeling, Panning and Control*. Springer-Verlag London Limited, 2009.

[8] Y. Tan y Z.-y. Zheng, "Research Advance in Swarm Robotics," *Defence Technology*, vol. 9, n.º 1, págs. 18-39, 2013.

[9] C. Perry, *A self-organizing thousand-robot swarm*, https://www.seas.harvard.edu/news/2014/08/self-organizing-thousand-robot-swarm, Accesado: 2021-21-04, 2014.

[10] T. Goddard, T. Do, M. Riel-Mehan, G. Johnson, A. Ling, K. Helkme, M. Broeker y W. Lim, *Kilobots, T-cells and Cancer*, https://www.cgl.ucsf.edu/chimera/data/kilobots-jan2015/cancerbots.html, Accesado: 2021-03-06, 2015.

[11] S. Hauert, *Thousand-robot swarm self-assembles into arbitrary shapes*, https://robohub.org/thousand-robot-swarm-self-assembles-into-arbitrary-shapes/, Accesado: 2021-03-06, 2014.

[12] W. Savoie, T. A. Berrueta, Z. Jackson, A. Pervan, R. Warkentin, S. Li, T. D. Murphey, K. Wiesenfeld y D. I. Goldman, "A robot made of robots: Emergent transport and control of a smarticle ensemble," *Science Robotics*, vol. 4, n.º 34, 2019.

[13] A. Sakharkar, *A robot built from 'smarticles' to achieve tasks like locomotion*, https://www.inceptivemind.com/robot-built-smarticle-achieve-tasks-locomotion/9192/, Accesado: 2021-03-06, 2019.

[14] P. Corke, *Robotics, Vision and Control Fundamental Algorithms in MATLAB*. Springer Nature, 2017.

[15] R. Szeliski, *Computer Vision: Algorithms and Applications*. London: Springer Science Business Media, 2010.

[16] K. Potdar, C. Pai y S. Akolkar, "A Convolutional Neural Network based Live Object Recognition System as Blind Aid," nov. de 2018. DOI: 10.13140/RG.2.2.34494.54085.

[17] R. H. Carver y K.-C. Tai, *Modern multithreading: implementing, testing, and debugging multithreaded Java and C++/Pthreads/Win32 programs*. John Wiley Sons, Inc., 2006.

[18] T. G. Mattson, B. Sanders y B. Massingill, *Patterns for Parallel Programming*. Pearson Education Inc., 2005.

[19] M. I. A. T. Team, *MATLAB Support Package for USB Webcams*, https://www.mathworks.com/matlabcentral/fileexchange/45182-matlab-support-package-for-usb-webcams, Accesado: 2021-09-16, 2021.

[20] P. Corke, *ROBOTICS TOOLBOX*, https://petercorke.com/toolboxes/robotics-toolbox/, Accesado: 2021-09-16, 2017.

[21] ——, *MACHINE VISION TOOLBOX*, https://petercorke.com/toolboxes/machine-vision-toolbox/, Accesado: 2021-09-16, 2005.

[22] R. Orellana, *Formato JPEG vs. PNG: elige el que mejor se adapte a tus necesidades*, https://es.digitaltrends.com/computadoras/formatos-jpeg-vs-png/, Accesado: 2021-09-15, 2020.

[23] P. Garcia, *JPG vs JPEG: ¿hay alguna diferencia entre ellos?* https://www.imymac.es/mac-cleaner/jpg-vs-jpeg.html, Accesado: 2021-09-15, 2021.

[24] J. Diener, *2D minimal bounding box*, https://www.mathworks.com/matlabcentral/fileexchange/31126-2d-minimal-bounding-box, *MATLAB Central File Exchange*. Accesado: 2021-09-28, 2021.

[25] J. E. Diener y A. Z. Elsherbeni, "FDTD Acceleration using MATLAB Parallel Computing Toolbox and GPU.," *Applied Computational Electromagnetics Society Journal*, vol. 32, n.º 4, págs. 283-288, 2017.

[26] A. D. M. Esquivel, "Implementación y Validación del Algoritmo de Robotica de Enjambre *Particle Swarm Optimization* en Sistemas Físicos," Tesis de licenciatura, Universidad del Valle de Guatemala, 2021.

[27] W. A. S. Azurdia, "Implementación y Validación del Algoritmo de Robotica de Enjambre *Ant Colony Optimization* en Sistemas Físicos," Tesis de licenciatura, Universidad del Valle de Guatemala, 2021.