



Instituto Politécnico Nacional



INSTITUTO POLITÉCNICO NACIONAL

ESCUELA SUPERIOR DE CÓMPUTO

DISEÑO DE UN SISTEMA AUTÓNOMO PARA MONITOREO

Versión Preliminar

Autores:
Hernández Vergara Eduardo
Rojas Cruz José Ángel

Noviembre 2024

Índice

1. Introducción	4
1.1. Objetivo	5
1.1.1. Objetivo General	5
1.1.2. Objetivos Específicos	5
1.2. Justificación	5
2. Marco Teórico	6
2.1. Introducción a los autómatas celulares	6
2.1.1. Teoría de Autómatas	6
2.1.2. Autómatas celulares de una dimensión	7
2.1.3. Condiciones frontera	8
2.1.4. Vecindario	9
2.1.5. Definición formal	10
2.1.6. Autómatas celulares de 2 Dimensiones	11
2.1.7. Ejemplo	13
2.1.8. Entropía de Shannon	14
2.1.9. Sistemas Dinámicos	14
2.2. Physarum Polycephalum	16
2.2.1. Mixomiceto	16
2.2.2. Ciclo de vida	16
2.2.3. Physarum Polycephalum	19
2.2.4. El Physarum Polycephalum visto desde la perspectiva computacional . .	20
2.3. Modo Gráfico	21
2.3.1. Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)	22
2.4. RasberryPi	23
2.4.1. Historia y Evolución	23
2.4.2. Comparativa	25
2.4.3. RasberryPi 4 Model B	26
2.5. Protocolos de comunicación	28
2.5.1. WebSocket	28
2.5.2. Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP)	29
2.5.3. Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC)	29
3. Estado del Arte	30
3.1. Physarum Polycephalum	30
3.1.1. Modelado de Adamatzky	31
3.1.2. Modelado Guillermo Olvera	32
3.1.3. Modelado de Yair Marin	33
3.1.4. Modelado de Jeff Jones	35
3.1.5. Modelado de Gunji	36
3.2. Robots para Monitoreo Poblacional	37
3.2.1. Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis	37
3.2.2. Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields	38

3.2.3. The Role of Robots in Environmental Monitoring	38
4. Propuesta a desarrollar	40
4.1. Requerimientos	40
4.1.1. Requerimientos Funcionales	40
4.1.2. Requerimientos no Funcionales	41
4.2. Diagramas	42
5. Implementación	51
5.1. Simulador del Physarum Polycephalum	51
5.1.1. Generación de rutas de nuestro simulador	57
5.1.2. Codificación e implementación de algoritmo en el robot en la primera iteración 1	64
5.1.3. Codificación e implementación de algoritmo en el robot en la primera iteración 2	65
5.1.4. Ajustes del algoritmo basados en pruebas unitarias 1	65
5.1.5. Ajustes del algoritmo basados en pruebas unitarias 2	66
5.1.6. Diseño inicial de interfaz para control y monitorear	66
5.1.7. Ajustes de rutas basado en resultado de pruebas de aceptación 1	67
5.2. Robot Propuesto	69
5.2.1. Desarrollo Inicial del Sistema del control del Robot	70
5.2.2. Ajustes de código en función de pruebas unitarias y de aceptación 1	78
5.2.3. Ajustes de código en función de pruebas unitarias y de aceptación 2	87
5.2.4. Ajustes de código en función de las pruebas de carga 1	96
5.2.5. Ajustes de código en función de pruebas unitarias y de aceptación 3	107
5.2.6. Ajustes de código en función de las pruebas de carga 2	124
6. Pruebas del sistema	143
6.1. Pruebas unitarias del algoritmo en simulación de mapas 1	144
6.2. Evaluación de desempeño del software en pruebas iniciales 1	144
6.3. Recopilación de datos del software y desempeño 1	150
6.4. Redacción del informe técnico inicial basado en la Iteración 1	155
6.5. Pruebas unitarias del algoritmo en simulación de mapas 2	155
6.6. Pruebas de aceptación en escenarios pequeños	156
6.6.1. 1ra Prueba de aceptación: Elección de estados a través del teclado.	156
6.6.2. 2da Prueba de aceptación: Colocación de los estados inicial y final.	156
6.6.3. 3ra Prueba de aceptación: Inicialización de la simulación.	157
6.7. Evaluación de desempeño del software en pruebas iniciales 2	157
6.8. Recopilación de datos del software y desempeño 2	160
6.9. Pruebas de aceptación en entornos medianos	169
6.9.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo	169
6.9.2. 2da Prueba de aceptación: Elección de estados a través del teclado.	169
6.9.3. 3ra Prueba de aceptación: Colocación de los estados inicial y final.	170
6.9.4. 4ta Prueba de aceptación: Inicialización de la simulación.	170
6.10. Pruebas de aceptación en entornos complejos simulados 1	171
6.10.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo	171
6.10.2. 2da Prueba de aceptación: Carga de mapas	171
6.10.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.	172
6.10.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.	172
6.10.5. 5ta Prueba de aceptación: Inicio de la simulación	173
6.11. Evaluación final del desempeño del software 1	173

6.12. Recopilación y análisis de datos del rendimiento del robot de IEEE 1872.1 e IEEE 2914	177
6.13. Redacción del informe con mejoras de la Iteración 2	177
6.14. Recopilación final de datos del software para el informe 1	178
6.15. Documentación del avance segunda iteración	178
6.16. Pruebas de aceptación en entornos complejos simulados 3	179
6.16.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo	179
6.16.2. 2da Prueba de aceptación: Carga de mapas	180
6.16.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.	180
6.16.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.	181
6.16.5. 5ta Prueba de aceptación: Inicio de la simulación	181
6.17. Evaluación final del desempeño del software 2	182
6.18. Recopilación final de datos del software para el informe 2	184
6.19. Informe final con resultados de la Iteración 3	184
6.20. Pruebas de aceptación en entornos complejos simulados 2	184
6.21. Análisis de Primera Iteración de Datos de Sensores	185
6.22. Diseño de módulos de hardware del robot para pruebas de integración	185
6.22.1. Módulo de actuadores (Motores)	186
6.22.2. Módulo de sensores (LiDAR)	187
6.22.3. Unidad de control (Raspberry Pi 4 B)	187
6.22.4. Módulo de visualización	187
6.22.5. Pruebas de integración	188
6.23. Desarrollo de módulos de hardware del robot para pruebas de integración	190
6.23.1. Módulo de actuadores	190
6.23.2. Módulo de sensores	191
6.23.3. Unidad de control	192
6.23.4. Sistema de visualización	201
6.23.5. Integración de módulos	201
6.24. Evaluación de primera iteración del desempeño del robot de pruebas integrales .	204
6.25. Pruebas unitarias e integración de sensores y motores en entornos controlados .	205
6.25.1. Pruebas unitarias	205
6.25.2. Pruebas de integración	210
6.26. Pruebas de Aceptación en escenarios controlados con retroalimentación de los sensores	212
6.27. Análisis de Segunda Iteración de Datos de Sensores para Implementación de Interfaz Gráfica	213
6.28. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos	214
6.29. Pruebas de aceptación en entornos complejos simulados y reales	215
6.30. Evaluación de segunda iteración del desempeño del robot de pruebas integrales .	215
6.31. Análisis tercera iteración de datos de sensores para implementación de aplicación móvil	216
6.32. Pruebas de aceptación en entornos complejos simulados y reales	218
6.33. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos	218
6.34. Evaluación de tercera iteración del desempeño del robot de pruebas integrales .	219
6.35. Redacción del Informe Final	219

Resumen - Este proyecto propone el diseño e implementación de un autómata capaz de determinar trayectos en espacios euclidianos. La aplicación de este sistema aborda desafíos en navegación autónoma y diseño de redes en tiempo real. Su relevancia se destaca en contextos como en sistemas autónomos. El autómata estará implementado en un modelo bidimensional no lineal. Este autómata podría servir para el monitoreo de entidades poblacionales y sistemas relacionados.

Palabras clave - Autómata, Control, Diseño de Redes, Ruteos, Tiempo Real.

Abstract - This project proposes the design and implementation of an automaton capable of determining trajectories in Euclidean spaces. The application of this system addresses challenges in autonomous navigation and real-time network design. Its relevance is highlighted in contexts such as in autonomous systems. The automaton will be implemented in a two-dimensional nonlinear model. This automaton could serve for the monitoring of population entities and related systems.

1. Introducción

En el complejo mundo actual, en el cual los procesos industriales se valen de la automatización y digitalización, los autómatas celulares emergen como sistemas dinámicos discretos de gran potencial. Dichos sistemas están constituidos por matrices de celdas; estas celdas son la unidad básica de los autómatas celulares y cada celda puede estar en un estado determinado, como 1 o 0, vivo o muerto. En realidad, pueden tener múltiples interpretaciones, pero básicamente es un sistema binario. Además, estas matrices de celdas se rigen por las reglas del autómata, que evolucionan conforme lo hacen las generaciones. En este proyecto nos enfocaremos en el desarrollo y aplicación de un autómata programable en una Raspberry Pi, con el objetivo de proporcionar a las organizaciones una solución integral y adaptable. Esta solución les permitirá el monitoreo y control eficiente de sistemas y procesos en tiempo real.

La necesidad de un monitoreo en tiempo real es muy alta en el entorno social, empresarial y en algunas aplicaciones para uso gubernamental, donde la agilidad y la eficacia son esenciales. En esta situación, los autómatas celulares se presentan como herramientas bastante versátiles a la hora de interactuar dinámicamente con su entorno. En términos simples, cada uno de los elementos en un autómata celular se relaciona con sus vecinos (celdas que se encuentran alrededor de nuestra celda actual), y su estado en la próxima generación se determina según el estado de sus vecinos en la generación actual. Esta capacidad única de interacción y cambio dinámico de estados los convierte en instrumentos poderosos para abordar una variedad de desafíos.

Este proyecto no solo se enfoca en la implementación técnica de un autómata celular, sino también en su aplicación práctica en entornos específicos. Los autómatas celulares han demostrado su valía en diversas áreas, desde el monitoreo y predicción del cambio de uso de la tierra hasta la planificación de rutas sin colisión para robots. Dos referencias particulares, [1] y [2], destacan por su relevancia directa a nuestra propuesta de Trabajo Terminal (TT), ya que se centran en tareas de monitoreo y la implementación de robots sin colisiones, respectivamente. Justamente en nuestro caso, es el monitoreo y la prevención de colisiones de robots.

En este enfoque buscamos resaltar la versatilidad de los autómatas celulares como herramientas de solución aplicables en situaciones del mundo real. A su vez, se espera que el autómata sea adaptable para que pueda ser implementado en sectores emergentes, como la Inteligencia

Artificial (IA), ampliando aún más su alcance y utilidad.

1.1. Objetivo

1.1.1. Objetivo General

Implementar un autómata que sea capaz de determinar sus trayectos en espacios bidimensionales para monitorear trazando rutas en tiempo real.

1.1.2. Objetivos Específicos

- Diseñar un autómata basado en el mixomiceto *Physarum polycephalum* que sea capaz de determinar trayectos en espacios bidimensionales.
- Implementar una simulación del autómata en un programa desarrollado en el lenguaje de programación C++.
- Implementar el autómata en robot cuyo controlador sea una Raspberry Pi 4.
- Diseñar un sistema de monitoreo que permita visualizar el estado del autómata y del robot.
- Realizar las pruebas en un entorno controlado.
- Realizar las pruebas en un entorno real.

1.2. Justificación

En el marco actual de la automatización constante de procesos y el desarrollo de herramientas que facilitan la realización de tareas, han surgido distintas tecnologías y procesos que han sabido atacar de la mejor manera las problemáticas que involucran a la automatización. Sin embargo, muchas veces es complicado darles un correcto seguimiento a las actividades y procesos que están involucrados en la realización de tareas automáticas, ocasionando distintos problemas que afectan en la solución del problema al que originalmente planteaban solucionar o complicar de manera innecesaria el proceso. Por lo que es necesario crear herramientas de monitorización con métodos más fiables a los ya existentes, y que, además de brindar un correcto seguimiento a los procesos a los cuales se les hace un análisis, tenga la capacidad de reaccionar ante los cambios relevantes e importantes que surjan durante la realización de las distintas tareas y procesos en los que se vea involucrado.

Es por eso por lo que se busca la implementación de un sistema robótico, el cual, por medio de la aplicación de la teoría de autómatas celulares, monitorice la realización de distintas tareas y procesos en los que se vea involucrado y, además, priorizando siempre que la monitorización sea efectiva, continua y confiable. Esto para las distintas industrias que realicen actividades en las cuales se vean involucradas la automatización de procesos y tareas.

Los autómatas celulares han sido usados para distintas disciplinas que van desde la antropología hasta los gráficos por computadora, sin embargo, ha sido muy poco visto en actividades que involucren sistemas robóticos debido al comportamiento y la manera en la que los autómatas celulares se comportan a partir de diversas entradas, siendo a veces complicado discernir el comportamiento que se tendrá, lo que añade complejidad al implementar uno de estos autómatas a sistemas robóticos. Pero gracias al conocimiento brindado por algunas materias como lo son Sistemas Operativos, Arquitectura de Computadoras, Diseño Digital y Teoría Computacional, se puede llegar a un procedimiento y tratamiento de la información tal que sea posible dirigir el autómata a la mejor solución posible.

2. Marco Teórico

Para avanzar en nuestro proyecto, conforme a los **Objetivos Específicos** previamente definidos, es esencial comprender los fundamentos de los *autómatas celulares*, incluyendo sus características y propiedades. Esta comprensión es clave para su implementación en nuestro proyecto. Por ello, dedicaremos esta sección a detallar estos conceptos básicos, características y propiedades de los autómatas celulares. Asimismo, proporcionaremos una introducción al mixomiceto *Physarum polycephalum*, destacando su conexión con los autómatas celulares.

Adicionalmente, subrayaremos el papel crucial de la *Raspberry Pi 4*, que se encarga de gestionar el robot y de aplicar el autómata celular. Incluirá también una descripción concisa de la librería gráfica *Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)* (o *Vulkan*), seleccionada para la simulación del autómata celular.

2.1. Introducción a los autómatas celulares

Primero es necesario conocer la teoría de autómatas y como se relaciona con los autómatas celulares, por ello daremos un breve repaso de la teoría de autómatas.

2.1.1. Teoría de Autómatas

La teoría de autómatas es el estudio de dispositivos de cálculo abstractos, es decir de las máquinas.^[3] Estos autómatas son modelos matemáticos fundamentales en el área de estudio de las ciencias de la computación, son usados para entender los procesos de cálculo y toma de decisiones. En la teoría de autómatas se estudian los autómatas finitos, los autómatas con pila, las máquinas de Turing, los autómatas celulares, etc. Los autómatas regulares pueden ser jerarquizados en una jerarquía de Chomsky, que es una jerarquía de lenguajes formales. La cual sería la siguiente^[4]:

- **Tipo 3 - Gramáticas regulares:** Estos generan los lenguajes regulares. Estas se restringen a producciones de la forma $A \rightarrow a\gamma$ y $A \rightarrow aB$. Son asociados a los autómatas finitos.
- **Tipo 2 - Gramáticas libres de contexto:** Estos generan los lenguajes independientes del contexto. Estas se restringen a producciones de la forma $A \rightarrow \gamma$. Son asociados a los autómatas con pila.
- **Tipo 1 - Gramáticas sensibles al contexto:** Estos generan los lenguajes sensibles al contexto. Estas se restringen a producciones de la forma $\alpha A \beta \rightarrow \alpha \gamma \beta$. Son asociados a las máquinas de Turing linealmente acotadas (significa que la cinta de la máquina de Turing tiene un límite determinado por un cierto número entero de veces sobre la longitud de entrada).
- **Tipo 0 - Gramáticas irrestrictas:** Estos generan los lenguajes recursivamente enumerables. Estas se restringen a producciones de la forma $\alpha A \beta \rightarrow \delta$. Son asociados a las máquinas de Turing.

En cambio los autómatas celulares, aunque diferentes en estructura y aplicación a las gramáticas formales, también son modelos matemáticos fundamentales en el área de estudio de las ciencias de la computación y forman parte de la teoría de autómatas. Mientras que los autómatas convencionales se centran en el procesamiento secuencial de cadenas de símbolos y operan basándose en estados y transiciones claramente definidos, los autómatas celulares utilizan una

red de células cuyos estados evolucionan en paralelo, siguiendo reglas locales. Esta diferencia fundamental en su enfoque los hace especialmente adecuados para modelar y explorar fenómenos que involucran procesos dinámicos y patrones espaciales. A pesar de estas diferencias, los autómatas celulares se alinean con los principios fundamentales de la teoría de autómatas en cuanto a la representación y manipulación de información, ofreciendo una perspectiva más amplia y diversa sobre lo que constituye un ‘autómata’ en el contexto de la computación y el procesamiento de información. Su inclusión en la teoría de autómatas subraya la amplitud y la profundidad de este campo, demostrando que la teoría de autómatas no solo se limita a las máquinas y lenguajes formales tradicionales, sino que también abarca modelos computacionales más generales y versátiles.

Una vez que hemos explicado la teoría de automatas podemos pasar a explicar en mas detalle los autómatas celulares.

2.1.2. Autómatas celulares de una dimensión

Los autómatas celulares de una dimensión consisten de una linea de celdas o estados, cada una con 2 estados posibles, 0 o 1, vivo o muerto, etc. Estas celdas se actualizan en cada generación, de acuerdo a una regla de evolución, la cual determina el estado de una celda en la siguiente generación, basándose en el estado de la celda y sus vecinos en la generación actual. La regla de evolución se aplica a todas las celdas de la misma manera, y en paralelo, es decir, todas las celdas se actualizan al mismo tiempo. Para ejemplificar esto, se puede ver la Figura 1. En donde tenemos la regla de evolución 30, de las *Reglas de Wolfram*[5], o *Automatas Celulares Elementales* (Elemental Cellular Automata, ECA), en la cual se puede ver que la celda de la generación $t + 1$ depende de la celda de la generación t y sus vecinos.

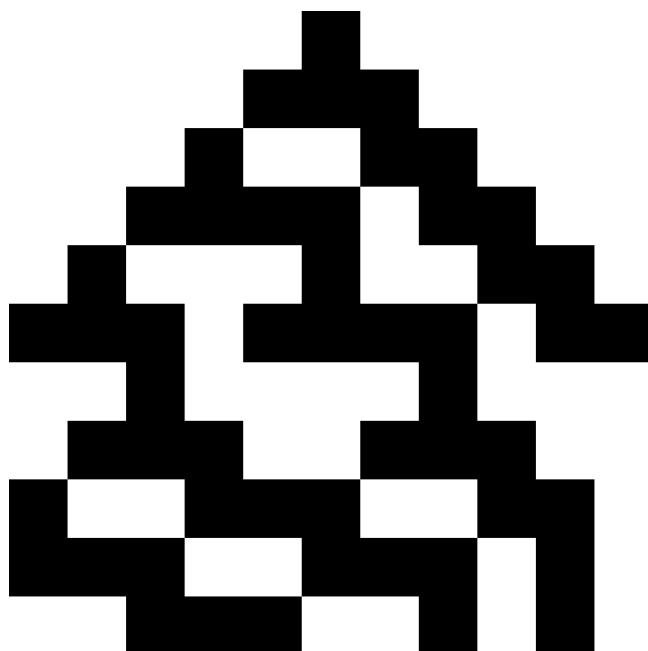


Figura 1: Regla de evolución 30 de las *Reglas de Wolfram* 10 generaciones o (t)

Como podemos ver en estos autómatas celulares de una dimensión los que podemos considerar los elementos más importantes son:

- **Vecindad:** Es el conjunto de celdas que se toman en cuenta para la actualización de una celda.

- **Estado:** Es el valor que puede tomar una celda, en este caso solo puede ser 0 o 1.
- **Regla de evolución:** Es la regla que determina el estado de una celda en la siguiente generación.

En este caso la vecindad la forman la celda y sus dos vecinos, pero puede ser de cualquier tamaño, siempre y cuando sea simétrica, es decir, que la celda se encuentre en el centro de la vecindad. El estado de la celda puede ser cualquier valor, pero en este caso solo puede ser 0 o 1. La regla de evolución es la que determina el estado de la celda en la siguiente generación, en este caso la regla de evolución 30, que se puede ver en la Figura 1, es la siguiente:

111	110	101	100	011	010	001	000
0	0	0	1	1	1	1	0

Cuadro 1: Regla de evolución 30

Como podemos ver en el Cuadro 1 la regla de evolución 30 es una regla de evolución local, es decir, que solo depende de la celda y sus vecinos, y no de toda la línea de celdas. En este caso la regla de evolución 30 es una regla de evolución determinista, es decir, que para una celda y sus vecinos siempre se obtiene el mismo resultado. Pero también existen las reglas de evolución no deterministas, en las cuales para una celda y sus vecinos se puede obtener más de un resultado. En este caso la regla de evolución 30 es una regla de evolución unidimensional, es decir, que la celda solo depende de sus vecinos de la izquierda y de la derecha, pero también existen las reglas de evolución bidimensionales, n-dimensionales, etc.

2.1.3. Condiciones frontera

Por definición los autómatas celulares son infinitos, pero en la práctica no se pueden tener autómatas celulares infinitos, por lo que se tienen que definir condiciones frontera, las cuales son las condiciones que se tienen en los extremos del autómata celular. Existen diferentes tipos de condiciones frontera, las cuales son:

- **Abierta** En este caso las celdas de los extremos no tienen vecinos, por lo que no se pueden actualizar.
- **Periódica** En este caso las celdas de los extremos tienen como vecinos a las celdas del otro extremo.
- **Reflejante** En este caso las celdas de los extremos tienen como vecinos a las celdas del otro extremo, pero invertidas.
- **Frontera** En este caso las celdas de los extremos tienen como vecinos a celdas con un valor fijo.

En nuestro caso, utilizaremos la condición frontera periódica; esta tiene forma de un toroide, como se puede ver en la Figura 2.

También podemos observar que en la Figura 2 se puede ver que las celdas de los extremos tienen como vecinos a las celdas del otro extremo, por lo que se puede decir que es una condición frontera periódica.

2.1.4. Vecindario

Un vecindario es un conjunto de celdas que se toman en cuenta para la actualización de una celda. Como vimos con anterioridad en los autómatas celulares de una dimensión, el vecindario de una celda es la celda y sus dos vecinos, pero en los autómatas celulares de dos dimensiones el vecindario de una celda puede ser de cualquier tamaño, siempre y cuando sea simétrico, es decir, que la celda se encuentre en el centro del vecindario. En la Figura 3 se puede ver un ejemplo de un vecindario de una celda. E incluso en los autómatas celulares de dos dimensiones se pueden tener vecindarios de diferentes formas, como se es común verlos en forma cuadrada y en forma hexagonal, así como se muestra en las Figuras 3 - 7.

A su vez los vecindarios más usados son los siguientes:

- **Vecindario de Moore** En este caso el vecindario de una celda es la celda y sus ocho vecinos.

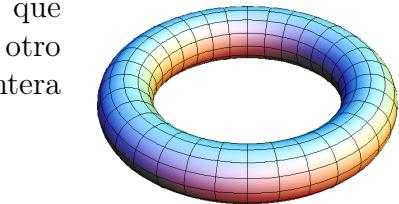


Figura 2: Representación de un toroide [6]

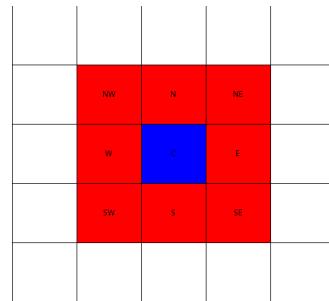


Figura 3: Vecindario de Moore

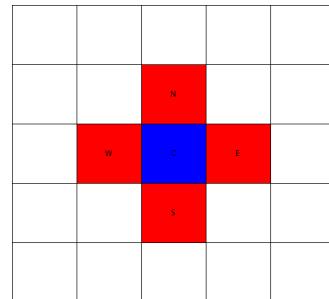


Figura 4: Vecindario de von Neumann

- **Vecindario de von Neumann** En este caso el vecindario de una celda es la celda y sus cuatro vecinos.

- **Vecindario de Moore extendido** En este caso el vecindario de una celda es la celda y sus veinticuatro vecinos.

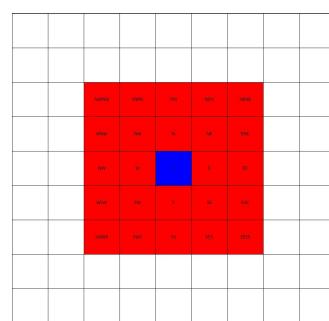


Figura 5: Vecindario de Moore extendido

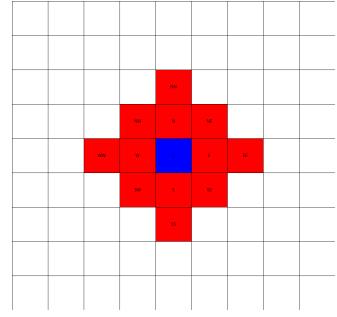


Figura 6: Vecindario de von Neumann extendido

- Vecindario de von Neumann extendido** En este caso el vecindario de una celda es la celda y sus doce vecinos.

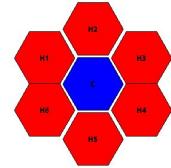


Figura 7: Vecindario de hexagonal

Una vez que hemos explicado eso podemos pasar a definir formalmente los autómatas celulares.

2.1.5. Definición formal

Primero denotemos \mathbb{Z} como el conjunto de los números enteros, es decir, $\mathbb{Z} = (-\infty, -1, 0, 1, \infty)$. y la longitud de cualquier tupla x como $|x|$. Para todas las tuplas x y y de la misma longitud, denotemos $x \oplus y$ como la tupla que resulta de la suma componente a componente de x y y , es decir, $(x \oplus y)_i = x_i + y_i$ para todo $i \in \mathbb{Z}$.

Entonces tenemos que un autómata celular es una tupla (\mathbb{Z}^n, S, N, f) tal que la n dimensión es al menos 1 donde $n \in \mathbb{Z}^+$, S es un conjunto finito no vacío de estados, N es un conjunto finito no vacío de vecindades perteneciente a \mathbb{Z}^n y f es una función de transición local, es decir, $f : S^N \rightarrow S$ donde S^N representa al conjunto de todas las posibles configuraciones de vecindad en N .

La configuración inicial de un autómata celular es una función $c : \mathbb{Z}^n \rightarrow S$ que asigna un estado a cada celda. La evolución de un autómata celular es una función $F : S^{\mathbb{Z}^n} \rightarrow S^{\mathbb{Z}^n}$ que asigna una configuración a la siguiente configuración, es decir, $F(c) = c'$, donde c' es la configuración resultante de aplicar la función de transición local a cada celda de la configuración c , es decir, $c'(x) = f(c|_{x+N})$ para todo $x \in \mathbb{Z}^n$, donde $c|_{x+N}$ es la restricción de c a la vecindad $x + N$. Esto también aplica n dimensionalmente, es decir, que se podría decir que $c'(x, y) = f(c|_{(x,y)+N})$ para todo $(x, y) \in \mathbb{Z}^2$. Otra notación que podemos usar, y de hecho es la que utilizaremos es $C(x, y : t)$ donde C es el centro de la vecindad, x, y son las coordenadas de la celda y t es el tiempo, o generaciones.

Cabe añadir que puede haber restricciones adicionales en el conjunto de vecindades N y en la función de transición local f . Por ejemplo, en el caso de los autómatas celulares de una dimensión, el conjunto de vecindades N es un conjunto de tuplas de longitud 3, donde la primera componente es la celda, la segunda componente es la celda de la izquierda y la tercera

componente es la celda de la derecha. Y la función de transición local f es una función de 8 variables booleanas, es decir, $f : \{0, 1\}^3 \rightarrow \{0, 1\}$.

Y en el caso de los autómatas celulares de dos dimensiones, el conjunto de vecindades N es un conjunto de tuplas de longitud variable, dependiendo del tipo de vecindad, donde la primera componente es la celda y las demás componentes son las celdas vecinas. Por ejemplo, en el caso de la vecindad de Moore, el conjunto de vecindades N es un conjunto de tuplas de longitud 9, donde la primera componente es la celda (x, y) el cual sería el centro de la vecindad, la segunda componente es la celda $(x - 1, y - 1)$, la tercera componente es la celda $(x - 1, y)$, la cuarta componente es la celda $(x - 1, y + 1)$, la quinta componente es la celda $(x, y - 1)$, la sexta componente es la celda $(x, y + 1)$, la séptima componente es la celda $(x + 1, y - 1)$, la octava componente es la celda $(x + 1, y)$ y la novena componente es la celda $(x + 1, y + 1)$. Aquí (x, y) es la celda central de la vecindad. Y la función de transición local f es una función de 512 variables booleanas, es decir, $f : \{0, 1\}^9 \rightarrow \{0, 1\}$.

Esta definición formal de autómata celular fue tomada de [7]. Una vez explicada la definición formal de autómata celular, podemos pasar a explicar a más detalle los autómatas celulares de 2 dimensiones, los cuales son los que se usan en este Trabajo Terminal (TT).

2.1.6. Autómatas celulares de 2 Dimensiones

Los autómatas celulares de 2 dimensiones son los que se usan en este Trabajo Terminal (TT), por ello es necesario explicarlos con más detalle. Primero recordando lo ya explicado en la sección 2.1.5, un autómata celular de 2 dimensiones es una tupla (\mathbb{Z}^2, S, N, f) y este necesita de 8 elementos para poder ser definido, los cuales son los siguientes:

1. **Celdas:** Es la unidad básica del autómata celular. Cada celda ocupa una posición en el espacio, para ser representados suelen usarse cuadrículas o redes, esta tiene un estado y una vecindad.
2. **Estados:** Cada celda puede estar en uno de varios estados posibles. En los autómatas celulares más simples, cada celda puede estar en uno de dos estados posibles (0 o 1, vivo o muerto, etc), pero en los autómatas celulares más complejos, cada celda puede estar en uno de varios estados posibles. En el caso en particular del Physarum polycephalum, cada celda puede estar en uno de nueve estados posibles $\mathbb{P} = \{x \in \mathbb{Z} | 0 \leq x \leq 8\}$ entonces es $S = \mathbb{P}$.
3. **Cuadrícula o Red:** Las celdas están dispuestas a lo largo del espacio euclíadiano, estas suelen disponerse en una cuadrícula o red, en donde cada celda ocupa una posición en el espacio. Es n-dimensional, pero en este caso es 2-dimensional, es decir, $n = 2$.
4. **Vecindad:** Es el conjunto de celdas que se toman en cuenta para la actualización de una celda. En el caso de la vecindad de Moore, que se puede ver en la sección 2.1.4, esta es $N = 8$
5. **Reglas de Transición:** Son un conjunto de reglas que determinan como cambia el estado de la celula en función del estado actual de ella y de sus vecinos. Estas reglas se aplican repetidamente a lo largo del tiempo, generalmente de manera síncrona, es decir, todas las celdas se actualizan al mismo tiempo. Estas están definidas por la función de transición local f . Ejemplificando lo anterior tenemos que en el juego de la vida de Conway [8] donde tenemos que $C(x, y : t)$ que es la celda central y $N(x, y : t)$ que es la vecindad de Moore

de la celda central, además tenemos que tiene $f : \{0, 1\}^9 \rightarrow \{0, 1\}$, entonces podemos deducir que la función de transición se define como:

$$f(C(x, y : t), N(x, y : t)) = \begin{cases} 1 & \text{si } C(x, y : t) = 0 \text{ y } N(x, y : t) = 3 \\ 1 & \text{si } C(x, y : t) = 1 \text{ y } N(x, y : t) = 2 \text{ o } N(x, y : t) = 3 \\ 0 & \text{en otro caso} \end{cases}$$

6. **Tiempo o Generaciones:** Es el número de veces que se aplica la función de transición local f . En este caso es $t \in \mathbb{Z}^+$.
7. **Condiciones Iniciales:** Antes de que el autómata celular comience a evolucionar, se debe especificar el estado de cada celda. En este caso es $c : \mathbb{Z}^2 \rightarrow S$. Estas condiciones iniciales pueden ser aleatorias o no.
8. **Condiciones Frontera:** Son las condiciones frontera previamente mencionadas en la sección 2.1.2.

2.1.7. Ejemplo

Una vez que hemos explicado los autómatas celulares de 2 dimensiones podemos pasar a explicar un ejemplo de un autómata celular de 2 dimensiones.

En este caso mostraremos un autómata celular de 2 dimensiones(binario) con vecindad de Moore y con una configuración totalística $B4678/S35678$, es decir, una celda nacerá si tiene 4, 6, 7 u 8 vecinos vivos y sobrevivirá si tiene 3, 5, 6, 7 u 8 vecinos vivos. Este autómata celular también es conocido con el nombre de *Anneal* y es mencionado en [9].

Para formalizar tenemos que este autómata $\mathbb{A} = (\mathbb{Z}^2, S, N, f)$ donde: $S = \{0, 1\}$, $N = \{0, 1\}^9$ y $f : \{0, 1\}^9 \rightarrow \{0, 1\}$ y C es la celda central, entonces podemos deducir que la función de transición se define como:

$$f(C(x, y : t), N(x, y : t)) = \begin{cases} 1 & \text{si } C(x, y : t) = 0 \text{ y } N(x, y : t) \in \{4, 6, 7, 8\} \\ 1 & \text{si } C(x, y : t) = 1 \text{ y } N(x, y : t) \in \{3, 5, 6, 7, 8\} \\ 0 & \text{en otro caso} \end{cases}$$

Dáandonos como resultado la siguiente evolución en 250 generaciones (véase la Figura 8).

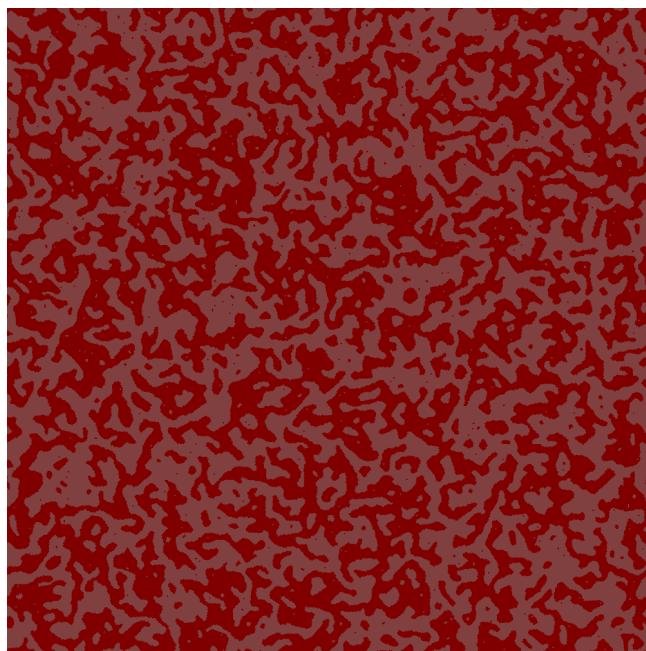


Figura 8: Evolución del autómata celular *Anneal*

2.1.8. Entropía de Shannon

La entropía de Shannon es un concepto fundamental en la teoría de la información, que se utiliza para medir la incertidumbre en una variable aleatoria. La entropía en si es el grado de información/desinformación que tenemos en un sistema. Es decir que cuanta más información tengamos de un sistema, menor será la entropía y viceversa. La entropía de Shannon, nombrada así por Claude Shannon[10], se define matemáticamente la suma negativa de las probabilidades de cada posible valor de la variable aleatoria multiplicada por el logaritmo de la probabilidad de ese valor. Esta definición es la siguiente:

$$H(X) = - \sum_{i=1}^n p(x_i) \log_b p(x_i) \quad (1)$$

Donde X es una variable aleatoria discreta, $p(x_i)$ es la probabilidad de que la variable aleatoria X tome el valor x_i y b es la base del logaritmo. En el caso de que la base del logaritmo sea 2, la unidad de medida de la entropía serán los bits. En el caso de que la base del logaritmo sea e , la unidad de medida de la entropía serán los nat. Y en el caso de que la base del logaritmo sea 10, la unidad de medida de la entropía serán los dits.

En nuestro caso en particular usamos bits como unidad de medida de la entropía, ya que lo usamos para saber que tipo de comportamiento tiene el autómata celular. Es decir, si la entropía es alta, entonces el autómata celular tiene un comportamiento caótico, y si la entropía es baja, entonces el autómata celular tiene un comportamiento ordenado.

Como vimos en ejemplo anterior, el autómata celular *Anneal* podemos observar rápidamente su entropía en la Figura 9:

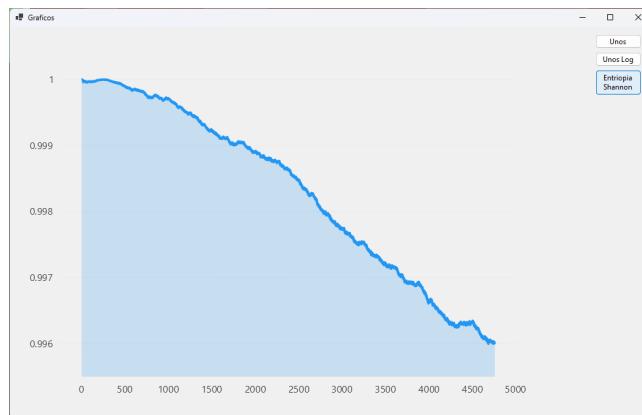


Figura 9: Entropía del autómata celular *Anneal* en 5000 generaciones

2.1.9. Sistemas Dinámicos

La teoría de Sistemas Dinámicos se puede concebir como un enfoque para describir cómo un estado dado se modifica o progresiona hacia otro a medida que transcurre el tiempo, es decir, cómo un sistema evoluciona temporalmente. La teoría de Sistemas Dinámicos se puede aplicar a cualquier área de la ciencia, como por ejemplo, la biología, la economía, la física, la química, etc. En nuestro caso en particular, la teoría de Sistemas Dinámicos se aplica a los autómatas celulares, ya que los autómatas celulares son sistemas dinámicos discretos.

Nos referimos a sistemas dinámicos discretos cuando el tiempo es discreto. Con los autómatas celulares podemos ver que el tiempo es discreto, ya que el tiempo se mide en generaciones. Es decir, en cada generación el autómata celular evoluciona de un estado a otro. En la Figura 10 podemos ver un ejemplo de la evolución de un autómata celular en el tiempo:

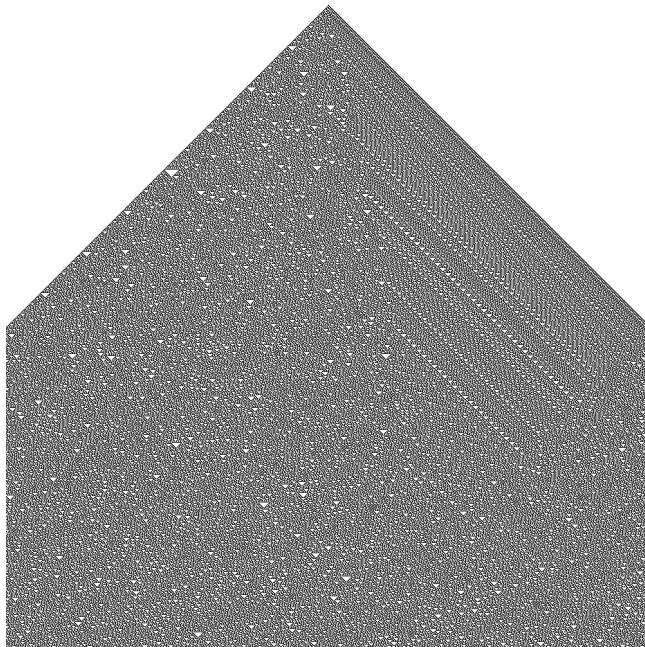


Figura 10: Evolución de la regla 30 en 1000 generaciones o pasos de tiempo

No daremos una explicación más en profundidad de los sistemas dinámicos, ya que no es el objetivo de este trabajo. Para más información sobre sistemas dinámicos y caos, se recomienda leer [11] y también [12]. Sin embargo, lo que sí es importante mencionar es que los sistemas dinámicos pueden ser clasificados, en este caso nos enfocaremos en los sistemas triviales, los sistemas complejos y los sistemas caóticos.

Esto lo hacemos con el propósito de poder clasificar el comportamiento de nuestro autómata celular (*Physarum Polycephalum*) en base a su comportamiento. Es decir, si el autómata celular se comporta de manera trivial, compleja o caótica. Para esto, primero daremos una breve explicación de cada uno de estos tipos de sistemas dinámicos. A su vez, quisiéramos destacar que para hacer nuestra clasificación o determinar el tipo de comportamiento tiene nuestro autómata celular usaremos una gráfica en la cual podamos observar la Entropía de Shannon en función del tiempo. Ya que para poder usar como método de clasificación los atractores, requerimos que la cantidad de estados en el *Physarum Polycephalum* es: $\mathbb{P} = \{x \in \mathbb{Z} | 0 \leq x \leq 8\}$ entonces es $S = \mathbb{P}$ y por lo tanto su función de transición es: $f : \mathbb{P} \rightarrow \mathbb{P}$ o sea $f : \mathbb{P}^N \rightarrow \mathbb{P}$ donde N es la vecindad del autómata celular. Por lo tanto, la función de transición es una función de \mathbb{P}^9 , dandonos como resultado un total de 9^9 o sea 387,420,489 estados posibles. Por lo tanto, no es posible graficar todos los estados posibles del autómata celular.

Ahora si, daremos una breve explicación de los sistemas mencionados anteriormente:

- **Sistemas Triviales:** Son sistemas que no tienen comportamiento complejo, es decir, son sistemas que tienen un comportamiento predecible. Por ejemplo, un péndulo simple, ya que su movimiento es periódico y predecible.
- **Sistemas Complejos:** Son sistemas que tienen un comportamiento complejo, es decir, son sistemas que tienen un comportamiento impredecible debido a su sensibilidad a las

condiciones iniciales y a la presencia de interacciones no lineales. Por ejemplo, el clima, ya que es imposible predecir el clima con exactitud.

- **Sistemas Caóticos:** Los sistemas caóticos son un subconjunto de sistemas complejos que son extremadamente sensibles a las condiciones iniciales. Pequeñas variaciones en las condiciones iniciales pueden llevar a resultados completamente diferentes en el tiempo. Los sistemas caóticos pueden parecer aleatorios y desordenados, pero en realidad están gobernados por ecuaciones matemáticas deterministas. Un ejemplo clásico de sistema caótico es el sistema de doble péndulo, donde el movimiento se vuelve impredecible y altamente sensible a las condiciones iniciales después de un corto período de tiempo.

2.2. Physarum Polycephalum

Para poder desarrollar este Trabajo Terminal (TT) es necesario conocer el organismo que se va a modelar, en este caso el *Physarum Polycephalum*. Por lo tanto, en esta sección daremos una breve introducción al *Physarum Polycephalum*, así como sus características y propiedades.

2.2.1. Mixomiceto

Los mixomicetos, también conocidos como myxo-mycetes o mohos mucilaginosos, son un grupo de fascinantes organismos unicelulares que se encuentran en el reino Protista, más concretamente protistas ameboídes o amobozoa. A pesar de su apariencia poco llamativa y su tamaño microscópico, los mixomicetos son organismos muy interesantes, por su ciclo de vida y su comportamiento biológico inusual.

A diferencia de las setas y otros hongos tradicionales, los mixomicetos no forman estructuras multicelulares visibles durante la mayor parte de su ciclo de vida. En cambio, existen como células individuales, generalmente microscópicas, denominadas myxamoebas, que se desplazan a través de ambientes húmedos y ricos en materia orgánica, buscando condiciones favorables para su crecimiento.[13]

Los mixomicetos según Rojas [13] toman 3 formas distintas durante el transcurso de su vida:

- **Amoeboídes:** Son células individuales que se mueven por medio de seudópodos o flagelos dependiendo principalmente de la cantidad de agua en el medio. Estas amebas se denominan *myxamoebas* y son las que se encuentran en el suelo.
- **Plasmodio:** Es una masa de citoplasma multinucleado sin separación de membranas celulares, que se mueve por medio de la contracción de sus fibras de actina. Este plasmodio es el que se encuentra en el interior de los troncos de los árboles.
- **Cuerpo fructífero:** Es la estructura que se forma cuando el plasmodio se transforma en esporas. Estas esporas son las que se encuentran en la parte superior de los troncos de los árboles.

2.2.2. Ciclo de vida

El ciclo de vida de los mixomicetos es muy complejo, sin embargo, un mixomiceto típico tiene un ciclo de vida que se puede dividir en dos etapas distintas, las cuales son: el plasmodio y uno o más cuerpos fructíferos.[14]

La secuencia de eventos que ocurren durante el ciclo de vida de un mixomiceto típico comienza con una espora microscópica que se formó dentro de uno de los cuerpos fructíferos, característicamente producidos por los mixomicetos, y luego es liberada de este. Bajo condiciones favorables, la espora germina para producir de uno a cuatro protoplastos haploides sin pared celular, denominados gametos. Estos son liberados a través de un pequeño poro que se forma en la pared de la espora aunque también pueden resultar de la espora abriéndose.

Algunos protoplastos son flagelados cuando se liberan, mientras que otros son ameboides. Estos últimos a veces pueden desarrollar flagelos después de un corto período de tiempo o, en algunos casos, simplemente permanecer ameboides. Las células flageladas se llaman células enjambre, mientras que las células no flageladas se llaman mixamoebas. Las mixamoebas y las células enjambre son interconvertibles, y la forma particular en la que existe una célula dada depende aparentemente en gran medida de la disponibilidad de agua libre en su entorno inmediato. En presencia de agua libre, la forma flagelada tiende a predominar, mientras que bajo condiciones de escasez de agua, las mixamoebas sin flagelos son más comunes.[14]

Las mixamoebas y las células enjambre se dividen en dos, y cuando las condiciones no son buenas, las mixamoebas pueden convertirse en estructuras inactivas llamadas microquistes, que les ayudan a sobrevivir por mucho tiempo. Cuando dos de estas células se juntan, forman un cigoto, que puede ser ameboide o flagelado, pero eventualmente se convierte en ameboide y crece para convertirse en un plasmodio, una estructura grande con muchos núcleos pero que funciona como una sola célula. Si las condiciones se vuelven difíciles, como por la falta de agua o frío, el plasmodio se puede transformar en un esclerocio, una forma resistente que puede volver a convertirse en plasmodio cuando las condiciones mejoren. En invierno, es posible encontrar esclerocios debajo de la corteza de troncos y tocones en descomposición.

El ciclo de vida de los mixomicetos se puede observar en la Figura 11, en donde:

- A) Espora
- B) Germinación de la espora
- C) Etapa unicelular, que es o una mixoamiba (izquierda) o una célula enjambre (derecha).
- D) Microquiste
- E) Fusión de dos mixamoebas o células enjambre para producir solo una célula.
- F) Fusión de dos mixamoebas o células enjambre para producir solo una célula.
- G) Cigoto
- H) Plasmodio temprano
- I) Esclerocio
- J) Plasmodio maduro
- K) Comienzo de la esporulación
- L) Cuerpos fructíferos maduros con esporas aún encerradas

Si desea profundizar en el tema, puede consultar el libro "Myxomycetes: Biology, Systematics, Biogeography, and Ecology" de Rojas [13].

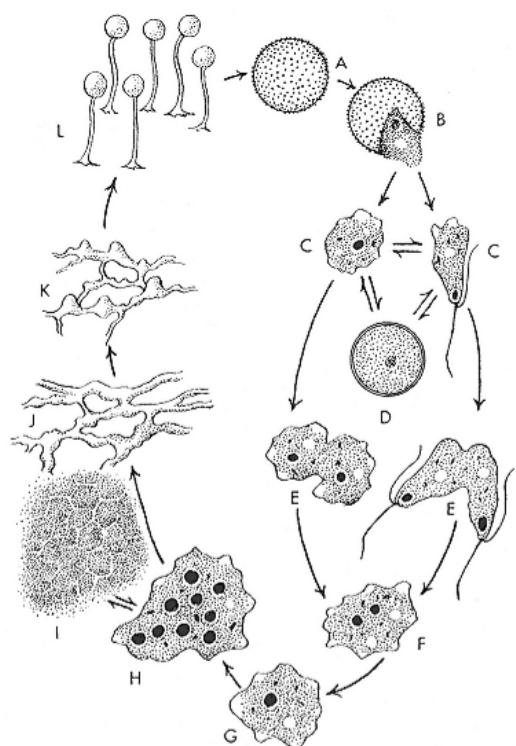


Figura 11: Ciclo de vida de los mixomicetos, extraído de "Myxomycetes: A Handbook of Slime Molds" de Stephenson y Stempf [14].

2.2.3. *Physarum Polycephalum*

El *Physarum Polycephalum*, también conocido como "The Blob", o "La Mancha", es un protista con formas celulares diversas. El *Physarum Polycephalum* es un mixomiceto acelular, esto proviene de la etapa plasmoidal de su ciclo de vida, en la cual el plasmodio es un coenocito multinucleado macroscópico de color amarillo brillante, formado en una red de tubos entrelazados. Esta etapa del ciclo de vida es la que se utiliza para el estudio de este organismo.[15] Podemos ver un ejemplo en la siguiente Figura 12.

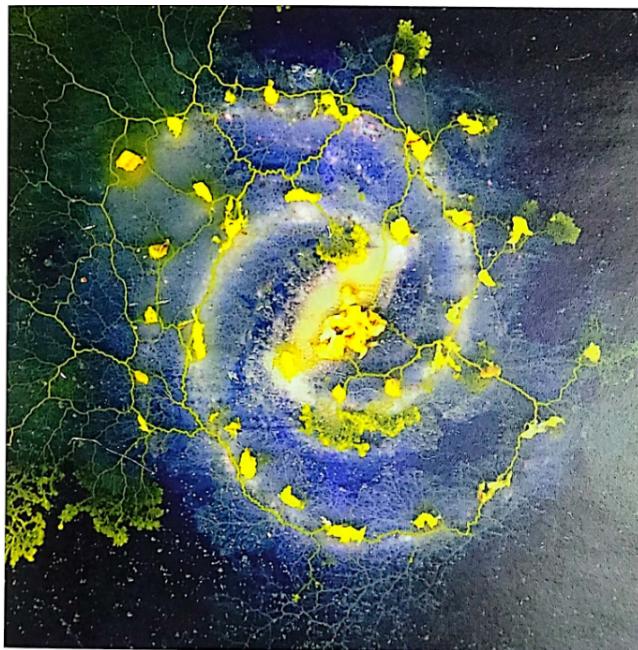


Figura 12: *Physarum* propagándose en una impresión artística de una galaxia. Imagen extraída de 'Atlas of *Physarum Computing*' de A. Adamatzky [16].

Como vimos con anterioridad, los mixomicetos se dividen en dos etapas, el plasmodio y los cuerpos fructíferos. El *Physarum Polycephalum* es un mixomiceto que se encuentra en la etapa plasmoidal de su ciclo de vida, en la cual el plasmodio es un coenocito multinucleado macroscópico de color amarillo brillante, formado en una red de tubos entrelazados. Esta etapa del ciclo de vida es la que se utiliza para el estudio de este organismo.[15]

El *Physarum Polycephalum* es un organismo que se encuentra en la naturaleza en lugares húmedos y oscuros, como en el interior de los troncos de los árboles en descomposición, en hojarasca húmeda, en suelos ricos en materia orgánica y en lugares oscuros y húmedos. Este organismo se alimenta de bacterias, hongos y otros microorganismos que se encuentran en su entorno, y se desplaza por medio de la contracción de sus fibras de actina, que le permiten moverse en busca de alimento.[15]

El *Physarum Polycephalum* es un organismo muy interesante para el estudio de la biología y la física, ya que tiene propiedades únicas que lo hacen un organismo muy especial. Por ejemplo, el *Physarum Polycephalum* es capaz de resolver laberintos como se observa en la Figura 13, encontrar la ruta más corta entre dos puntos, y tomar decisiones complejas basadas en la informa-

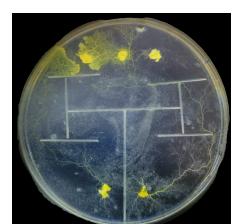


Figura 13:
Physarum Polycephalum resolviendo un laberinto. [16].

ción que recibe de su entorno. Además, el Physarum Polycephalum es capaz de aprender y recordar información, y de adaptarse a su entorno de una manera muy eficiente.

Una vez dada una breve introducción al Physarum Polycephalum, podemos pasar a la perspectiva de la computación, en donde el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización, simulación de redes de transporte, y modelado de sistemas complejos. En la siguiente sección veremos cómo el Physarum Polycephalum ha sido utilizado en la computación y en la modelación de sistemas complejos.

2.2.4. El Physarum Polycephalum visto desde la perspectiva computacional

Como se mencionó en la sección 2.2.3, el Physarum Polycephalum es un organismo notable que ha despertado un gran interés por parte de biólogos y matemáticos debido a su notable capacidad para exhibir comportamientos emergentes y resolver problemas de optimización de manera eficiente, demostrando una gran versatilidad. Entre sus comportamientos complejos se encuentran la locomoción, la formación de redes adaptativas y la toma de decisiones descentralizadas.

En la computación, el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización, simulación de redes de transporte, y modelado de sistemas complejos. En particular, el Physarum Polycephalum ha sido utilizado para resolver problemas de optimización de rutas, como el problema del camino más corto, el problema del flujo máximo, y el problema de la cobertura de sensores. Además, el Physarum Polycephalum ha sido utilizado para modelar sistemas complejos, como la formación de redes de transporte, la formación de patrones en sistemas biológicos, y la formación de estructuras en sistemas físicos.

Por mencionar algunos ejemplos de aplicaciones del Physarum Polycephalum en la computación, tenemos los siguientes:

- **A physarum-inspired prize-collecting steiner tree approach to identify subnetworks for drug repositioning:** En el artículo se detalla cómo un algoritmo, inspirado en el moho Physarum polycephalum, se aplica para descubrir medicamentos que podrían ser útiles en el tratamiento de enfermedades cardiovasculares. Mediante la construcción de Redes de Similitud de Fármacos (Drug Similarity Networks, DSNs), donde los nodos representan medicamentos y las conexiones reflejan similitudes entre ellos basadas en características como la estructura química y los efectos terapéuticos, cada medicamento recibe un 'premio' según su similitud con otros ya utilizados en afecciones cardiovasculares. El algoritmo busca dentro de estas redes para encontrar subredes que maximicen estos premios y minimicen los costos (disimilitudes), identificando así grupos de fármacos potencialmente reutilizables para tratar enfermedades cardiovasculares. Este método propone una forma innovadora de repensar el uso de medicamentos existentes, ofreciendo un camino acelerado hacia el descubrimiento de nuevas aplicaciones terapéuticas en el campo cardiovascular. [17]
- **A Novel Physarum-Based Ant Colony System for Solving the Real-World Traveling Salesman Problem:** Este artículo introduce un nuevo sistema de colonia de hormigas, inspirado en el modelo matemático de Physarum, para abordar el Problema del agente viajero (Traveling Salesman Problem, TSP). Este sistema ha demostrado ser más eficiente y robusto en comparación con los sistemas tradicionales de colonia de hormigas, algoritmos genéticos y optimización por enjambre de partículas. Este estudio se encuentra en un capítulo del libro 'Advances in Swarm Intelligence'. [18]

- **Composing Popular Music with Physarum polycephalum-based Memristors:** Este artículo investiga el uso de Physarum polycephalum, un moho mucilaginoso, como memristor para la composición de música popular, presentando una colaboración entre organismos biológicos y sistemas computacionales en la creación musical. Mediante una interfaz hardware-software, el estudio transforma datos musicales en voltajes y viceversa, utilizando el comportamiento no lineal del moho para influir en la composición. Aunque requiere ajustes para integrar las salidas del organismo en las piezas musicales, este enfoque innovador abre nuevas posibilidades en la creatividad computacional y la producción musical, instando a músicos y no expertos a explorar el cómputo no convencional en sus procesos creativos. El trabajo subraya el potencial de incorporar tecnologías biológicas en la composición musical, marcando un paso hacia la diversificación de las herramientas creativas en la música popular. [19]
- **Monte Carlo Physarum Machine: Characteristics of Pattern Formation in Continuous Stochastic Transport Networks:** El artículo introduce la Máquina de Physarum Monte Carlo (Monte Carlo Physarum Machine, MCPM), un modelo avanzado para reconstruir redes de transporte a partir de datos en 2D y 3D, ampliando un modelo previo de Jones sobre el moho Physarum polycephalum. La MCPM se evalúa por su capacidad para generar estructuras complejas denominadas poliformas y se aplica en la reconstrucción de la red cósmica, mostrando eficacia con datos cosmológicos simulados y observacionales. Los autores, afiliados a la Universidad de California, Santa Cruz y la Universidad Estatal de Nuevo México, exploran también aplicaciones futuras del MCPM en diversas disciplinas. [20]
- **Using an Artificial Physarum polycephalum Colony for Threshold Image Segmentation:** Este artículo presenta un innovador algoritmo basado en la simulación de una colonia de Physarum polycephalum artificial para abordar el problema de la segmentación de imágenes por umbral, un área clave en el procesamiento de imágenes. Tradicionalmente, los algoritmos de Inteligencia Artificial (IA) enfrentan desafíos en la selección del umbral óptimo, tendiendo a caer en óptimos locales. La metodología propuesta simula la expansión y contracción de hifas artificiales para buscar soluciones óptimas, facilitando el aprendizaje mutuo entre diferentes Physarum polycephalum y mejorando la capacidad de búsqueda global. Utilizando la entropía de Kapur como función de ajuste, el algoritmo propuesto demuestra una mayor precisión y velocidad de convergencia en comparación con métodos convencionales, validado a través de experimentos de referencia. Este enfoque abre nuevas perspectivas en el campo del procesamiento de imágenes, particularmente en aplicaciones de segmentación por umbral, ofreciendo una herramienta prometedora para resolver problemas complejos en esta área. [21]

Como se puede observar, el Physarum Polycephalum ha demostrado ser una fuente de inspiración para el desarrollo de algoritmos y sistemas computacionales innovadores, que han sido aplicados en una amplia variedad de campos, desde la biología y la medicina, hasta la música y la cosmología. Su capacidad para resolver problemas complejos de manera eficiente y su versatilidad para adaptarse a diferentes entornos lo convierten en un organismo único y valioso para la investigación científica y la computación.

2.3. Modo Gráfico

Para poder desarrollar el simulador de Physarum Polycephalum, es necesario conocer el modo gráfico, ya que es la interfaz que el usuario va a utilizar para interactuar con el simulador.

2.3.1. Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML)

Para el desarrollo de nuestro Trabajo Terminal (TT), se utilizó la Biblioteca Multimedia Simple y Rápida (Simple and Fast Multimedia Library, SFML), la cual es una biblioteca gráfica multiplataforma de código abierto, que proporciona una Interfaz de Programación de Aplicaciones (Application Programming Interface, API) simple y fácil de usar para el desarrollo de aplicaciones multimedia y videojuegos. SFML está escrita en C++ y proporciona una interfaz de programación orientada a objetos, que facilita la creación de aplicaciones gráficas y multimedia de alto rendimiento.

En nuestro caso lo estamos usando en C++, ya que buscamos un mejor rendimiento en tiempo de ejecución la comparación la podemos ver en la Figura 14. Además de que SFML es una biblioteca muy popular en la comunidad de desarrollo de videojuegos, por lo que es una buena opción para el desarrollo de nuestro simulador. Y no es tan complicada como Vulkan o DirectX, y ya que el propósito de nuestro trabajo no es el desarrollo del simulador sino del Robot, no lo consideramos vital el uso de una biblioteca más compleja.

SFML proporciona una serie de módulos que permiten el desarrollo de aplicaciones multimedia y videojuegos, incluyendo gráficos 2D, sonido, entrada de teclado y ratón, y redes. Además, SFML proporciona una serie de clases y funciones que facilitan la creación de aplicaciones gráficas y multimedia, como ventanas, sprites, texturas, fuentes, sonidos, y eventos. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

SFML es una biblioteca multiplataforma que está disponible para Windows, Linux y macOS, y es compatible con una amplia variedad de compiladores y entornos de desarrollo, como Visual Studio, Code::Blocks, y Xcode. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

SFML es una biblioteca de código abierto que está disponible bajo la licencia zlib/png, lo que significa que se puede utilizar de forma gratuita en proyectos comerciales y no comerciales, y se puede modificar y distribuir libremente. SFML también proporciona una serie de módulos que permiten la creación de aplicaciones multimedia y videojuegos, como gráficos 2D, sonido, entrada de teclado y ratón, y redes.

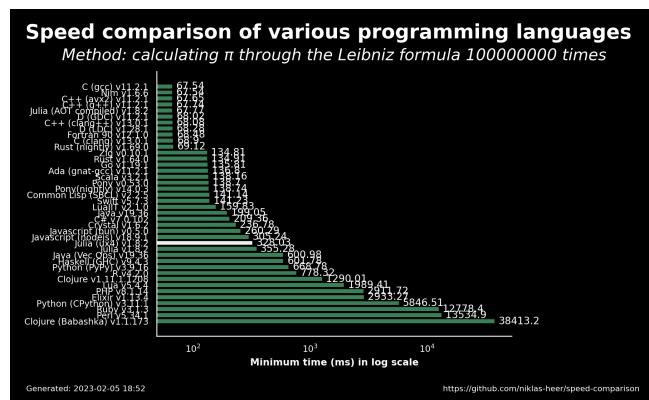


Figura 14: Comparación de tiempo de ejecución entre diferentes lenguajes de Programación. Esta gráfica fue generada Heer [22]

2.4. RasberryPi

En nuestro Trabajo Terminal (TT), la Raspberry Pi 4 es la encargada de gestionar el robot y la ruta que nos da el Physarum por medio de bluetooth. Por ello en esta sub sección, daremos una breve introducción a la Raspberry Pi 4, especificaciones técnicas, comparativas, etc.

2.4.1. Historia y Evolución

La Raspberry Pi nació en 2006 como un proyecto ideado por Eben Upton, Rob Mullins, Jack Lang y Alan Mycroft, quienes trabajaban en la Universidad de Cambridge. La idea principal era crear una computadora de bajo costo que permitiera a los estudiantes de la universidad mejorar sus habilidades de programación. [23] En 2009, el proyecto se convirtió en una fundación sin fines de lucro, la Raspberry Pi Foundation, con el objetivo de promover la enseñanza de la informática en las escuelas y países en desarrollo. [23]

La primera Raspberry Pi fue lanzada en febrero de 2012, con un procesador ARM11 de 700 MHz, 512 MB de RAM y un precio de 35 dólares. Desde entonces, la Raspberry Pi ha evolucionado hasta convertirse en una plataforma de desarrollo muy popular, con millones de unidades vendidas en todo el mundo.[23] La Raspberry Pi 4, lanzada en junio de 2019, es la versión más reciente de la placa y cuenta con un procesador ARM Cortex-A72 de 1.5 GHz, hasta 8 GB de RAM y soporte para pantallas 4K. [23]

La Raspberry Pi ha sido utilizada en una amplia variedad de proyectos, desde servidores web y centros multimedia hasta robots y sistemas de control. Su bajo costo y su flexibilidad la han convertido en una herramienta muy popular entre los aficionados a la informática y la electrónica. Además, la Raspberry Pi ha sido utilizada en proyectos educativos en todo el mundo, ayudando a enseñar a los jóvenes las habilidades necesarias para el siglo XXI.

- **Raspberry Pi Model B:** La Raspberry Pi Model B es la primera versión de la placa, lanzada en febrero de 2012. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 1 puerto USB tipo A, 1 conector GPIO de 8 pines, salida HDMI, salida de audio y un lector de tarjetas SD [23]
- **Raspberry Pi Model A+:** La Raspberry Pi Model A+ es una versión más pequeña y económica de la placa, lanzada en noviembre de 2014. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 1 puerto USB tipo A, 1 conector GPIO de 40 pines, salida HDMI y salida de audio 3.5 mm [23]
- **Raspberry Pi 2 Model B:** La Raspberry Pi 2 Model B es la segunda versión de la placa, lanzada en febrero de 2015. Cuenta con un procesador ARM Cortex-A7 de 900 MHz, 1 GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm y ethernet 10/100 [23]
- **Raspberry Pi Zero:** La Raspberry Pi Zero es una versión más pequeña y económica de la placa, lanzada en noviembre de 2015. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines y HAT compatible de 40 pines [23]
- **Raspberry Pi 3 Model B:** La Raspberry Pi 3 Model B es la tercera versión de la placa, lanzada en febrero de 2016. Cuenta con un procesador ARM Cortex-A53 de 1.2 GHz, 1 GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, ethernet 10/100, conexión Wifi y Bluethooth 4.1 LE [23]

- **Raspberry Pi Zero W:** La Raspberry Pi Zero W es una versión más pequeña y económica de la placa, lanzada en febrero de 2017. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluethooth 4.1 LE [23]
- **Raspberry Pi Zero WH:** La Raspberry Pi Zero WH es una versión más pequeña y económica de la placa, lanzada en febrero de 2018. Cuenta con un procesador ARM11 de 1 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluethooth 4.1 LE [23]
- **Raspberry Pi 3 Model B+:** La Raspberry Pi 3 Model B+ es la cuarta versión de la placa, lanzada en marzo de 2018. Cuenta con un procesador ARM Cortex-A53 de 1.4 GHz, 1 GB de RAM, 4 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, ethernet 10/100, conexión Wifi y Bluethooth 4.2 LE [23]
- **Raspberry Pi 3 Model A+:** La Raspberry Pi 3 Model A+ es una versión más pequeña y económica de la placa, lanzada en noviembre de 2018. Cuenta con un procesador ARM Cortex-A53 de 1.4 GHz, 512 MB de RAM, 1 puerto USB tipo A 2.0, 1 conector GPIO de 40 pines, salida HDMI, salida de audio 3.5mm, conexión Wifi y Bluethooth 4.2 LE [23]
- **Raspberry Pi 4 Model B:** La Raspberry Pi 4 Model B es la quinta versión de la placa, lanzada en junio de 2019. Cuenta con un procesador ARM Cortex-A72 de 1.5 GHz, hasta 8 GB de RAM, 2 puertos USB tipo A 3.0, 2 puertos USB tipo A 2.0, 1 conector GPIO de 40 pines, 2 salidas micro HDMI, salida de audio 3.5mm, ethernet Gigabit, conexión Wifi y Bluethooth 5.0 LE [23]
- **Raspberry Pi Compute Module 1:** La Raspberry Pi Compute Module 1 es una versión de la placa diseñada para su uso en sistemas embebidos, lanzada en abril de 2014. Cuenta con un procesador ARM11 de 700 MHz, 512 MB de RAM, 4GB eMMC Flash, Conector SODIMM DDR2 [23]
- **Raspberry Pi Compute Module 3:** La Raspberry Pi Compute Module 3 es una versión de la placa, lanzada en enero de 2017. Cuenta con un procesador BCM2837 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, 4GB eMMC Flash, Conector SODIMM DDR2 y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 3 Lite:** La Raspberry Pi Compute Module 3 Lite es una versión de la placa, lanzada en enero de 2017. Cuenta con un procesador BCM2837 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, Conector SODIMM DDR2 y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 3+:** La Raspberry Pi Compute Module 3+ es una versión de la placa, lanzada en enero de 2019. Cuenta con un procesador BCM2837B0 de cuatro núcleos a 1.2 GHz, 1 GB de RAM, 8GB, 16GB y 32 GB eMMC Flash, slot MicroSDHC y Conector GPIO 46 pines [23]
- **Raspberry Pi Compute Module 4:** La Raspberry Pi Compute Module 4 es una versión de la placa, lanzada en octubre de 2020. Cuenta con un procesador ARM a 1.5 GHz, 1 GB, 2 GB, 4 GB, 8 GB de RAM, 2 puertos Gigabit Ethernet, Conectividad Wi-Fi (opcional), 1 USB C y conector GPIO de 28 pines [23]
- **Raspberry Pi 400:** La Raspberry Pi 400 es una versión de la placa, lanzada en noviembre de 2020. Cuenta con un procesador ARM Cortex-A72 de 1.5 GHz y soporte de 64 bits, 1-8 GB de RAM, 2 puertos USB tipo A 3.0, 2 puertos USB tipo A 2.0, Conector GPIO

de 40 pines, 2 salidas micro HDMI, salida de audio 3.5mm, ethernet Gigabit, conexión Wifi y Bluethooth 5.0 LE [23]

- **Raspberry Pi Pico:** La Raspberry Pi Pico es una placa de desarrollo, lanzada en enero de 2021. Cuenta con un procesador RP2040 de doble núcleo ARM Cortex-M0+ a 133 MHz, 264 KB de RAM, 2 MB de memoria flash QSPI, 26 pines GPIO, 3 pines analógicos, 2 UART, 2 SPI, 2 I2C, 16 canales PWM, 1 temporizador de 12 bits y 1 temporizador de 16 bits [23]
- **Raspberry Pi Zero 2 W:** La Raspberry Pi Zero 2 W es una versión de la placa, lanzada en octubre de 2021. Cuenta con un procesador BCM2710A1 de cuatro núcleos a 1.0 GHz, 512 MB de RAM, 1 puerto mini HDMI, 1 puerto micro USB OTG, 1 conector GPIO de 40 pines, HAT compatible de 40 pines, conexión Wifi y Bluethooth 4.2 LE [23]
- **Raspberry Pi 5:** La Raspberry Pi 5 es una versión de la placa, lanzada en octubre de 2023. Cuenta con un procesador ARM Cortex-A73 de 2.4 GHz, hasta 8 GB de RAM, Doble salida micro HDMI 4K60p, gpu VideoCore VII con soporte de OpenGL ES 2.1 y Vulkan 1.2, decodificador HEVC 4K60, Bluethooth 5.0, WiFi 802.11ac, Ranura microSD de alta velocidad con soporte de SDR104, 2 puertos USB 3.0, 2 puertos USB 2.0, 1 puerto Gigabit Ethernet, interfaz PCIe 2.0, conexiones GPIO de 40 pines y botón de encendido y apagado [23]

2.4.2. Comparativa

Una vez visto los modelos de Raspberry Pi, es necesario hacer una comparativa entre ellos para poder elegir el modelo que mejor se adapte a nuestras necesidades. En el Cuadro 2 se muestra una comparativa entre los modelos de Raspberry Pi.

Cuadro 2: Comparación de los modelos de Raspberry Pi

Modelo	CPU	RAM	Puertos USB	GPIO	Salida Video	Red	Memoria
Model B	ARM11 700 MHz	512 MB	1 tipo A	8 pines	HDMI	No	Lector SD
Model A+	ARM11 700 MHz	512 MB	1 tipo A	40 pines	HDMI, Audio 3.5mm	No	No
2 Model B	ARM Cortex-A7 900 MHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100	No
Zero	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	No	No
3 Model B	ARM Cortex-A53 1.2 GHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100, Wifi, BT 4.1	No
Zero W	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.1	No

Continúa en la siguiente página

Cuadro 2 – continuación de la página anterior

Modelo	CPU	RAM	Puertos USB	GPIO	Salida Video	Red	Memoria
Zero WH	ARM11 1 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.1	No
3 Model B+	ARM Cortex-A53 1.4 GHz	1 GB	4 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Ethernet 10/100, Wifi, BT 4.2	No
3 Model A+	ARM Cortex-A53 1.4 GHz	512 MB	1 tipo A 2.0	40 pines	HDMI, Audio 3.5mm	Wifi, BT 4.2	No
4 Model B	ARM Cortex-A72 1.5 GHz	hasta 8 GB	2 tipo A 3.0, 2 tipo A 2.0	40 pines	2 micro HDMI	Ethernet Gigabit, Wifi, BT 5.0	MicroSD
Compute Module 1	ARM11 700 MHz	512 MB	No	SODIMM DDR2	No	No	4GB eMMC
Compute Module 3	BCM2837 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	4GB eMMC
Compute Module 3 Lite	BCM2837 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	No
Compute Module 3+	BCM2837B0 1.2 GHz	1 GB	No	GPIO 46 pines	No	No	8/16/32 GB eMMC
Compute Module 4	ARM 1.5 GHz	1/2/4/8 GB	USB C	GPIO 28 pines	No	2x Ethernet Gigabit, Wifi (opc)	No
Pi 400	ARM Cortex-A72 1.5 GHz	1-8 GB	2 tipo A 3.0, 2 tipo A 2.0	40 pines	2 micro HDMI	Ethernet Gigabit, Wifi, BT 5.0	No
Pico	RP2040 ARM Cortex-M0+ 133 MHz	264 KB	No	26 GPIO	No	No	2 MB flash
Zero 2 W	BCM2710A1 1.0 GHz	512 MB	mini HDMI, micro USB OTG	40 pines	mini HDMI	Wifi, BT 4.2	No
Pi 5	ARM Cortex-A73 2.4 GHz	hasta 8 GB	2 USB 3.0, 2 USB 2.0	40 pines	Doble micro HDMI 4K60p	Ethernet Gigabit, Wifi, BT 5.0	MicroSD SDR104

2.4.3. RasberryPi 4 Model B

Como ya vimos con anterioridad en la sección 2.4.2, la Raspberry Pi 4 Model B es una computadora de placa única (Single Board Computer, SBC) desarrollada por la Fundación Raspberry Pi. Es la cuarta generación de la serie Raspberry Pi y fue lanzada en junio de 2019.

Nosotros enfatizaremos sus usos que tienen para el desarrollo de un robot autónomo, como el que estamos desarrollando en nuestro Trabajo Terminal (TT).

Ventajas de la Raspberry Pi 4 Model B:

- **Alto Rendimiento:** El procesador Broadcom BCM2711 de cuatro núcleos a 1.5GHz ofrece un rendimiento significativamente mayor que las versiones anteriores de la Raspberry Pi, lo que permite ejecutar algoritmos de control y navegación más complejos.
- **Conectividad Avanzada:** La Raspberry Pi 4 Model B cuenta con dos puertos USB 3.0, dos puertos USB 2.0, un puerto Gigabit Ethernet, conexión Wi-Fi 802.11ac y Bluetooth 5.0, lo que facilita la comunicación con otros dispositivos, sensores y redes.
- **Flexibilidad de E/S:** La placa ofrece una amplia variedad de opciones de entrada y salida, incluyendo GPIO, HDMI, USB, Ethernet, Wi-Fi, Bluetooth, cámaras y pantallas, lo que permite conectar una gran variedad de sensores, actuadores y periféricos.
- **Soporte de Software:** La Raspberry Pi 4 Model B es compatible con una amplia variedad de sistemas operativos, incluyendo Raspbian, Ubuntu, Windows 10 IoT Core y otros, lo que facilita el desarrollo de aplicaciones y la integración con otros dispositivos.
- **Bajo Costo:** La Raspberry Pi 4 Model B es una placa de bajo costo, lo que la hace accesible para estudiantes, aficionados y profesionales que deseen desarrollar proyectos de robótica y automatización.

Usos de la Raspberry Pi 4 Model B en robótica:

- **Control de Robots:** La Raspberry Pi 4 Model B se puede utilizar para controlar robots móviles, drones, brazos robóticos y otros dispositivos autónomos, gracias a su alto rendimiento, conectividad avanzada y flexibilidad de E/S.
- **Visión por Computadora:** La Raspberry Pi 4 Model B se puede utilizar para procesar imágenes y vídeos en tiempo real, lo que permite a los robots detectar objetos, seguir líneas, evitar obstáculos y realizar otras tareas de visión por computadora.
- **Aprendizaje Automático:** La Raspberry Pi 4 Model B se puede utilizar para ejecutar algoritmos de aprendizaje automático y redes neuronales, lo que permite a los robots aprender de su entorno, adaptarse a nuevas situaciones y mejorar su rendimiento con el tiempo.
- **Interacción con el Entorno:** La Raspberry Pi 4 Model B se puede utilizar para interactuar con el entorno físico a través de sensores y actuadores, lo que permite a los robots medir la temperatura, la humedad, la luz, la distancia, la velocidad y otras variables, así como controlar motores, luces, pantallas y otros dispositivos.
- **Comunicación Inalámbrica:** La Raspberry Pi 4 Model B se puede utilizar para comunicarse de forma inalámbrica con otros dispositivos, sensores y redes, lo que permite a los robots enviar y recibir datos, comandos y actualizaciones de forma remota.

Por todo lo anteriormente mencionado, la Raspberry Pi 4 Model B es una excelente opción para el desarrollo de un robot autónomo, ya que ofrece un alto rendimiento, conectividad avanzada, flexibilidad de E/S, soporte de software y bajo costo, lo que la hace accesible para estudiantes, aficionados y profesionales que deseen desarrollar proyectos de robótica y automatización.

2.5. Protocolos de comunicación

La comunicación entre dispositivos es un aspecto fundamental en la robótica, ya que permite la interacción entre los diferentes componentes de un sistema. En nuestro Trabajo Terminal (TT), la comunicación entre el robot y el Physarum se realiza por medio de una aplicación móvil que se comunica mediante diversos protocolos de comunicación. En esta subsección, se describirán los protocolos de comunicación utilizados en nuestro Trabajo Terminal (TT).

2.5.1. WebSocket

El protocolo de WebSocket fue desarrollado ya que el protocolo HTTP no es adecuado para aplicaciones en tiempo real, esto por que el protocolo HTTP es de petición-respuesta, lo que significa que el cliente debe solicitar información al servidor y el servidor debe responder a la solicitud. En cambio, el protocolo WebSocket permite una comunicación bidireccional entre el cliente y el servidor, en otras palabras, existe una conexión persistente entre el cliente y el servidor, lo que permite que el servidor envíe información al cliente sin que este lo solicite. [24]

A diferencia de los protocolos de comunicación tradicionales, WebSocket permite reducir la latencia en aplicaciones que requieren una actualización constante de datos, como juegos multijugador, chats en tiempo real, o plataformas de trading financiero. Esta capacidad es posible gracias al establecimiento de una conexión persistente a través de un único canal Protocolo de Control de Transmisión (Transmission Control Protocol, TCP), que permanece abierta hasta que alguna de las partes decide cerrarla. Esto reduce la sobrecarga de establecer conexiones repetidas y mejora significativamente el rendimiento de las aplicaciones que requieren actualizaciones constantes. [25]

El proceso de establecimiento de una conexión WebSocket comienza con un "handshake" basado en HTTP, donde el cliente solicita la apertura de una conexión WebSocket al servidor utilizando un encabezado específico, y el servidor responde aceptando o rechazando la conexión. Una vez completado el "handshake", la conexión se actualiza y ambos pueden intercambiar mensajes en formato binario o texto sin necesidad de seguir el ciclo de solicitud-respuesta. Esto hace que WebSocket sea altamente eficiente para aplicaciones en tiempo real que manejan grandes cantidades de datos o requieren baja latencia. [26]

Además, WebSocket proporciona ventajas en cuanto a la reducción del uso de ancho de banda. Al evitar la necesidad de crear múltiples conexiones y al eliminar los encabezados HTTP innecesarios en cada intercambio de mensajes, se logra una transmisión de datos más ligera. Esto es especialmente útil en entornos donde los recursos de red son limitados, como dispositivos móviles o redes con baja velocidad. [27]

Sin embargo, aunque WebSocket ofrece muchas ventajas en términos de rendimiento y latencia, su implementación puede presentar desafíos de seguridad, como la exposición a ataques de secuestro de WebSocket entre sitios (Cross-Site WebSocket Hijacking, CSWSH) o vulnerabilidades de inyección. Por esta razón, es importante integrar medidas de seguridad, como el uso de WebSockets sobre el protocolo de Seguridad de la Capa de Transporte (Transport Layer Security, TLS), conocido como WSS para cifrar las comunicaciones, y políticas adecuadas de validación del origen de las conexiones [28].

2.5.2. Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP)

El protocolo Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP) es un protocolo de comunicación utilizado en la World Wide Web para la transferencia de información entre un cliente y un servidor. Fue diseñado para ser un protocolo simple y flexible, que permitiera la transferencia de datos de manera eficiente y segura. [29]

HTTP opera bajo el modelo petición-respuesta, donde el cliente envía una petición al servidor solicitando un recurso específico, y el servidor responde con el recurso solicitado o un código de estado que indica si la petición fue exitosa o no. Las peticiones y respuestas en HTTP están compuestas por un conjunto de encabezados y opcionalmente un cuerpo de mensaje, que contiene la información a transferir. [29]

HTTP es un protocolo sin estado, lo que significa que cada petición se procesa de manera independiente, sin tener en cuenta las peticiones anteriores. Esto permite que el servidor sea más escalable y flexible, ya que no necesita mantener un estado de sesión con cada cliente. Sin embargo, esta característica también implica que el servidor no puede recordar información sobre el cliente entre peticiones, lo que puede limitar la interacción entre el cliente y el servidor. [29]

HTTP utiliza el Protocolo de Control de Transmisión (Transmission Control Protocol, TCP) como su capa de transporte, lo que garantiza una comunicación fiable y ordenada entre el cliente y el servidor. Las conexiones HTTP se establecen mediante un *handshake* entre el cliente y el servidor, donde se negocian los parámetros de la conexión, como el tipo de contenido aceptado, la codificación de transferencia, y la longitud del cuerpo del mensaje. Una vez establecida la conexión, el cliente y el servidor pueden intercambiar mensajes de manera eficiente y segura. [29]

A pesar de su simplicidad y flexibilidad, HTTP tiene algunas limitaciones en términos de rendimiento y eficiencia. Por ejemplo, HTTP es un protocolo de texto plano, lo que significa que los mensajes enviados a través de HTTP deben ser codificados en texto legible por humanos, lo que puede aumentar el tamaño de los mensajes y reducir la eficiencia de la transferencia de datos. Además, HTTP no es adecuado para aplicaciones en tiempo real, ya que su modelo petición-respuesta puede introducir latencia en la comunicación entre el cliente y el servidor. [29]

A pesar de estas limitaciones, HTTP sigue siendo uno de los protocolos de comunicación más utilizados en la World Wide Web, debido a su simplicidad, flexibilidad y compatibilidad con una amplia variedad de plataformas y tecnologías. Sin embargo, en aplicaciones que requieren una comunicación más eficiente y en tiempo real, es posible que sea necesario utilizar protocolos más especializados, como WebSocket o el Protocolo de Telemetría de Colas de Mensajes (Message Queuing Telemetry Transport, MQTT), que están diseñados específicamente para este propósito. [29]

2.5.3. Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC)

Comunicación en Tiempo Real en la Web (Web Real-Time Communication, WebRTC) es un conjunto de tecnologías que permite la comunicación en tiempo real entre navegadores web y aplicaciones móviles. Fue desarrollado por Google en 2011 con el objetivo de facilitar

la creación de aplicaciones de comunicación en tiempo real, como videollamadas, conferencias web y transmisión de datos en tiempo real. [30]

WebRTC se basa en varios estándares abiertos, como el Protocolo de Transporte en Tiempo Real (Real-Time Transport Protocol, RTP), el Protocolo de Control de Transporte en Tiempo Real (Real-Time Transport Control Protocol, RTCP) y Protocolo de Descripción de Sesión (Session Description Protocol, SDP), que permiten la transmisión de datos en tiempo real a través de la web. Estos estándares están diseñados para ser compatibles con una amplia variedad de dispositivos y plataformas, lo que facilita la creación de aplicaciones de comunicación en tiempo real que funcionan en diferentes entornos. [30]

Una de las características más importantes de WebRTC es su capacidad para establecer conexiones punto a punto entre los clientes, lo que permite una comunicación directa y segura entre los usuarios sin necesidad de pasar por un servidor centralizado. Esto reduce la latencia y mejora la calidad de la comunicación, ya que los datos se transmiten directamente entre los clientes sin intermediarios. Además, al utilizar cifrado de extremo a extremo, WebRTC garantiza la privacidad y seguridad de las comunicaciones, protegiendo los datos de posibles ataques o interceptaciones. [30]

WebRTC es compatible con una amplia variedad de dispositivos y plataformas, incluyendo navegadores web, aplicaciones móviles y dispositivos Internet de las Cosas (Internet of Things, IoT), lo que lo convierte en una solución versátil para la creación de aplicaciones de comunicación en tiempo real en diferentes entornos. Además, al ser un estándar abierto, WebRTC está respaldado por una amplia comunidad de desarrolladores y empresas, lo que garantiza su compatibilidad y soporte a largo plazo. [30]

3. Estado del Arte

En esta sección, presentamos un resumen de los trabajos previos relacionados con algoritmos previamente implementados del Physarum Polycephalum. Además, se incluyen trabajos relacionados con autómatas celulares y su aplicación en espacios euclidianos. Finalmente, se presentan trabajos relacionados al monitoreo de sistemas poblacionales y sistemas relacionados.

3.1. Physarum Polycephalum

En esta sección nos concentraremos principalmente en los diferentes modelos que se han propuesto para modelar el Physarum Polycephalum, así como en las aplicaciones que se han desarrollado a partir de estos modelos. Principalmente son 5 los modelos que se han propuesto para modelar el Physarum Polycephalum, los cuales son: el modelado de Adamatzky, el modelado de Olvera, el modelado de Marín, el modelado de Jones y el modelado de Gunji. A continuación, se describirán brevemente cada uno de estos modelos.

3.1.1. Modelado de Adamatzky

El modelo de *Physarum polycephalum* de Andrew Adamatzky destaca sus capacidades computacionales. Manipulando *Physarum* con alimentos y repelentes, demuestra la creación de puertas lógicas y la resolución de problemas de optimización como el camino más corto y el problema del agente viajero [31]. En la Figura 15 se muestra un ejemplo de puertas lógicas creadas por *Physarum*.

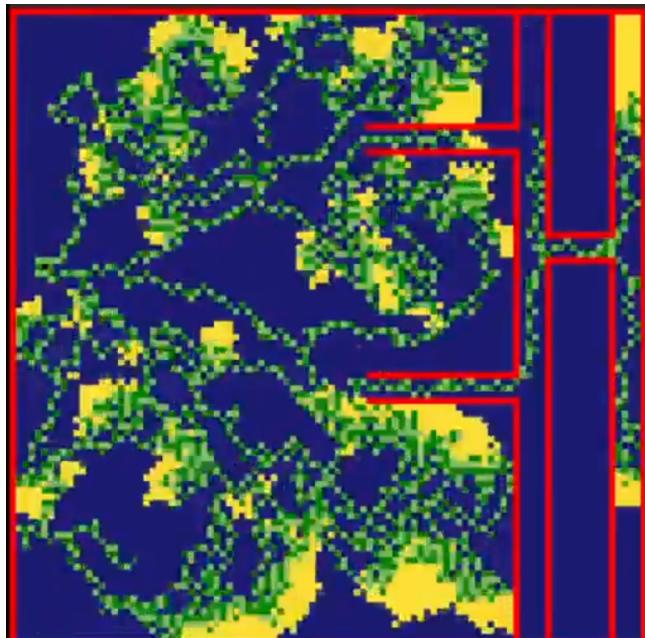


Figura 15: Ejemplo de puertas lógicas (OR) creadas por *Physarum*.

Adamatzky profundiza en la red protoplasmática de *Physarum*, comparándola con sistemas humanos, y muestra propiedades memristivas similares a los memristores electrónicos [31]. Su investigación también explora la dinámica no lineal y la formación de patrones complejos, significativos para la computación no convencional [31].

El modelo utiliza partículas de enjambre en un entorno bidimensional (2D) para simular el movimiento ameboide de *Physarum*. Estas partículas tienen etapas sensitiva y motora, interactuando con un quimioatrayente y generando patrones complejos. Además, se considera la resistencia al movimiento y la influencia de estímulos externos como quimioatrayentes y luz, controlando el comportamiento del colectivo.

Una característica destacada es la capacidad del colectivo para cambiar y recuperar su forma, navegar obstáculos y dividirse en fragmentos independientes que pueden fusionarse nuevamente, lo cual es deseable en aplicaciones robóticas [31]. La Figura 16 ilustra cómo el colectivo de *Physarum* se adapta a obstáculos.

Para más detalles, ver '*Emergence of self-organized amoeboid movement in a multi-agent approximation of physarum polycephalum*' [33].



Figura 16: Colectivo de *Physarum* adaptándose a obstáculos. [32]

3.1.2. Modelado Guillermo Olvera

El modelo utiliza autómatas celulares con la vecindad de Moore para simular la propagación y búsqueda de rutas del organismo *Physarum Polycephalum*. Este modelo funciona en una cuadrícula bidimensional donde cada célula puede asumir uno de varios estados: campo libre, nutriente no encontrado, repelente, punto inicial, gel en contracción, gel en compuesto, nutriente hallado, expansión del *Physarum* y gel sin compuesto.

El algoritmo está implementado en Python, lo cual introduce ciertas limitaciones en términos de velocidad y escalabilidad. Debido a la naturaleza interpretada de Python, el modelo es lento y poco escalable cuando se incrementa el número total de células e hilos utilizados.

El proceso comienza con la designación de una célula inicial que representa el punto de inicio del *Physarum*. Las reglas de transición determinan cómo cambian los estados de las células en función de sus vecinos. Por ejemplo, una célula de campo libre se convierte en una célula de expansión del *Physarum* si está adyacente a una célula en estado de punto inicial, gel en contracción o nutriente hallado. Las células de expansión del *Physarum* se propagan por la cuadrícula, y al encontrarse con nutrientes, estas células cambian su estado a nutriente hallado, lo que refuerza la ruta. En la Figura 17 se muestra la configuración inicial del *Physarum* en la cuadrícula.

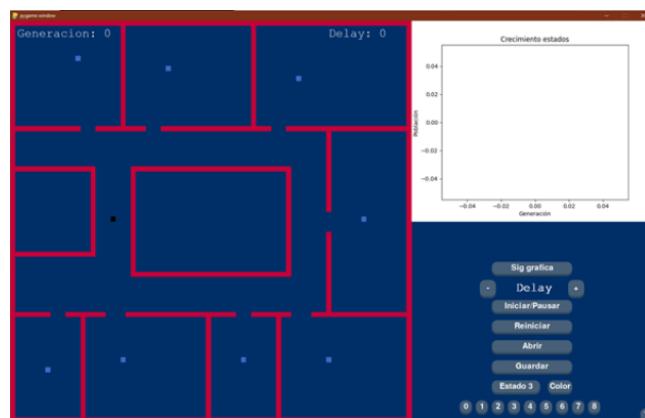


Figura 17: Configuración inicial del *Physarum* en la cuadrícula. [34]

A medida que el *Physarum* se expande, se generan rutas que conectan las fuentes de nutrientes, adaptándose dinámicamente a la presencia de obstáculos y asegurando la conectividad en el entorno. Aunque el algoritmo garantiza la formación de al menos una ruta viable, la trayectoria del *Physarum* no es realmente aleatoria, ya que sigue patrones determinados por las reglas de transición. Estas reglas, sin embargo, no siempre son claras ni consistentes. En la Figura 18 se muestra la expansión del *Physarum* y la formación de rutas.

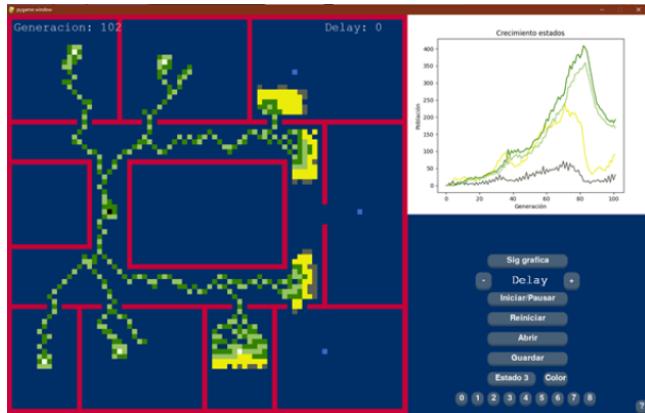


Figura 18: Expansión del *Physarum* y formación de rutas. [34]

Las reglas de transición se definen para cada estado de la célula, asegurando que el *Physarum* pueda encontrar y seguir rutas hacia los nutrientes, adaptándose en tiempo real a cambios en el entorno. Sin embargo, estas reglas no siempre son claras ni están bien documentadas en la referencia [34].

3.1.3. Modelado de Yair Marin

El algoritmo utilizado para modelar el comportamiento del *Physarum Polycephalum* en el documento se basa en autómatas celulares y se define por un conjunto de estados y reglas de transición específicas. Los estados incluyen campo libre, nutriente no encontrado, repelente, punto inicial, gel en contracción, gel con compuesto, nutriente hallado, expansión del *Physarum* y gel sin compuesto. Estos estados evolucionan de acuerdo con la vecindad de von Neumann, que considera las células adyacentes en las direcciones norte, sur, este y oeste.

Las reglas de transición determinan cómo cambia el estado de cada célula en función de sus vecinos. Por ejemplo, una célula en estado de campo libre (q_0) puede pasar al estado de expansión del *Physarum* (q_7) si está adyacente a un punto inicial (q_3), gel en contracción (q_4) o nutriente hallado (q_6). Del mismo modo, una célula en estado de nutriente no encontrado (q_1) cambia a estado de nutriente hallado (q_6) si está cerca de un gel con compuesto (q_5) o otro nutriente hallado (q_6). Estas reglas permiten que el modelo emule el comportamiento del *Physarum* en la búsqueda y exploración de su entorno, formando redes eficientes para el transporte de nutrientes y adaptándose a cambios en el entorno [35].

En la Figura 19, se muestra una ejecución del algoritmo, demostrando cómo el *Physarum* se expande y encuentra nutrientes en un entorno simulado.

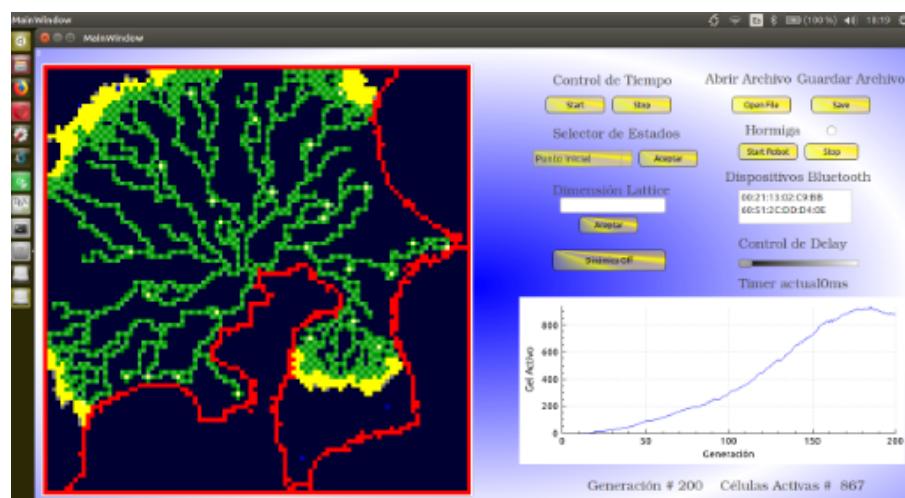


Figura 19: Ejecución del algoritmo de *Physarum Polycephalum* encontrando nutrientes. [35]

3.1.4. Modelado de Jeff Jones

El algoritmo propuesto en 'From Pattern Formation to Material Computation: Multi-agent Modelling of *Physarum Polycephalum*' [36] aprovecha el comportamiento natural de *Physarum polycephalum* para resolver problemas computacionales mediante un marco de sistema multi-agente (MAS). El núcleo del algoritmo involucra un gran número de agentes simples que imitan el comportamiento del plasmodio de *Physarum*. Cada agente opera basado en reglas locales, moviéndose e interactuando dentro de un entorno virtual que simula el espacio físico donde reside el moho del limo.

Los agentes se mueven hacia las fuentes de nutrientes siguiendo gradientes químicos, representando las señales atrayentes utilizadas por *Physarum*. Dejan rastros similares a feromonas que refuerzan los caminos exitosos, de manera similar a cómo *Physarum* fortalece sus tubos protoplasmáticos. Este mecanismo de retroalimentación de feromonas permite que los agentes se adapten dinámicamente a los cambios en el entorno, optimicen caminos y encuentren soluciones a problemas como el camino más corto o la resolución de laberintos. El comportamiento colectivo de estos agentes simples conduce a la emergencia de redes complejas y eficientes que pueden ser utilizadas para tareas de computación no convencional.

La fortaleza del algoritmo radica en su capacidad de autoorganización y adaptación sin control central. Demuestra capacidades robustas de resolución de problemas incluso en entornos dinámicos e inciertos. Al aprovechar los principios de autoorganización e interacción local, el algoritmo ofrece un enfoque novedoso para la computación distribuida y la optimización, inspirado en el comportamiento natural de *Physarum polycephalum*.

En la Figura 20 se muestra una representación del agente de acuerdo con el modelo de Jeff Jones.

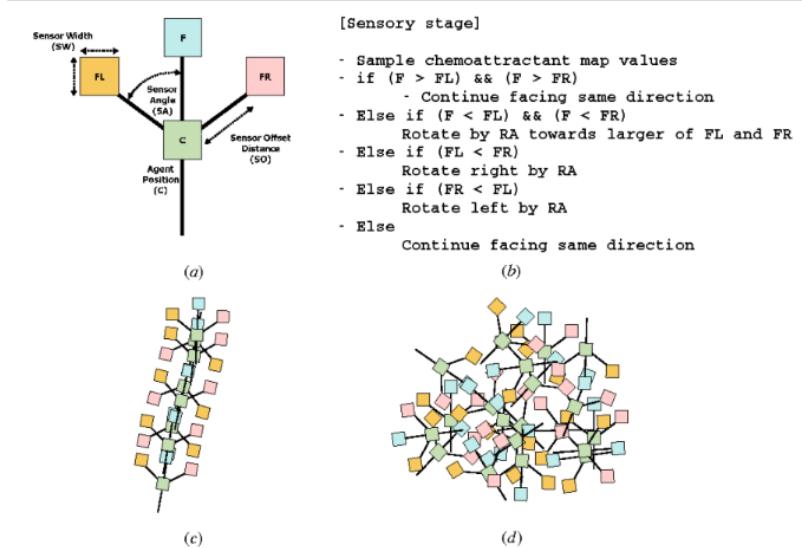


Figura 20: Representación del agente de acuerdo con el modelo de Jeff Jones. [36]

3.1.5. Modelado de Gunji

El modelo de célula mínima inspirado en el comportamiento del moho del limo *Physarum polycephalum* simula la capacidad de la célula para moverse y resolver problemas complejos como laberintos y configuraciones de árboles generadores a través de mecanismos simples pero efectivos. La célula está representada en una rejilla plana, donde cada sitio puede estar en uno de varios estados: externo (0), interno (1), límite (2) o estado final (-1). El modelo presenta dos fases principales: desarrollo y búsqueda de alimento. Durante la fase de desarrollo, la célula crece desde una semilla inicial hasta formar una agregación estructurada, mientras que en la fase de búsqueda de alimento, modifica activamente su forma y se mueve 'comiendo' sitios externos, causando flujo citoplasmático y reorganización de los límites.

Un aspecto clave del modelo es el proceso de 'comer 0', donde un sitio en estado 0 (externo) invade la célula, convirtiéndose en una 'burbuja' que es transportada dentro de la célula sin cruzar su propio camino (flujo memorizado). Este proceso conduce a la formación y eliminación de tentáculos, creación de redes adaptativas y optimización de caminos para resolver problemas como laberintos y configuraciones de árboles generadores. La interacción entre el flujo citoplasmático local y la forma global de la célula, impulsada por la alternancia entre endurecimiento y ablandamiento citoplasmático, permite que la célula se adapte dinámicamente y mantenga su estructura, exhibiendo comportamientos similares a la resolución inteligente de problemas observada en *Physarum polycephalum* [37].

En la Figura 21, se muestra cómo se aplica el algoritmo para resolver un laberinto. Este ejemplo ilustra la capacidad del modelo para adaptarse y optimizar caminos en tiempo real.

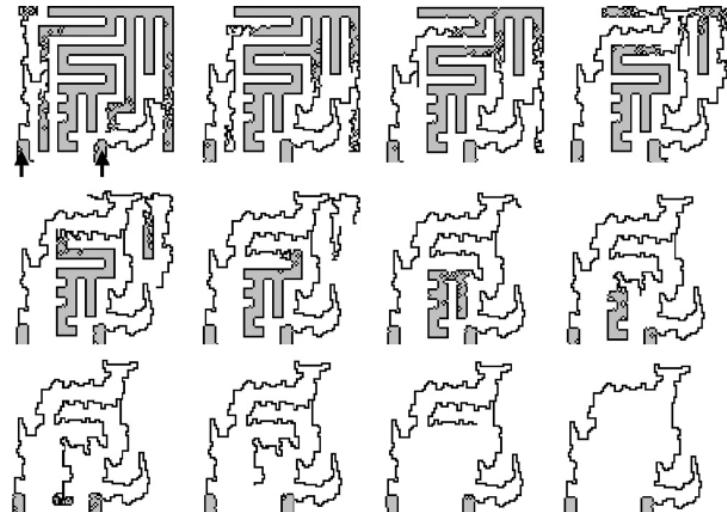


Figura 21: Aplicación del algoritmo para resolver un laberinto utilizando el modelo de *Physarum polycephalum*. [37]

Las reglas del modelo se describen en la Figura 22, mostrando los diferentes estados de los sitios y cómo interactúan durante las fases de desarrollo y búsqueda de alimento.

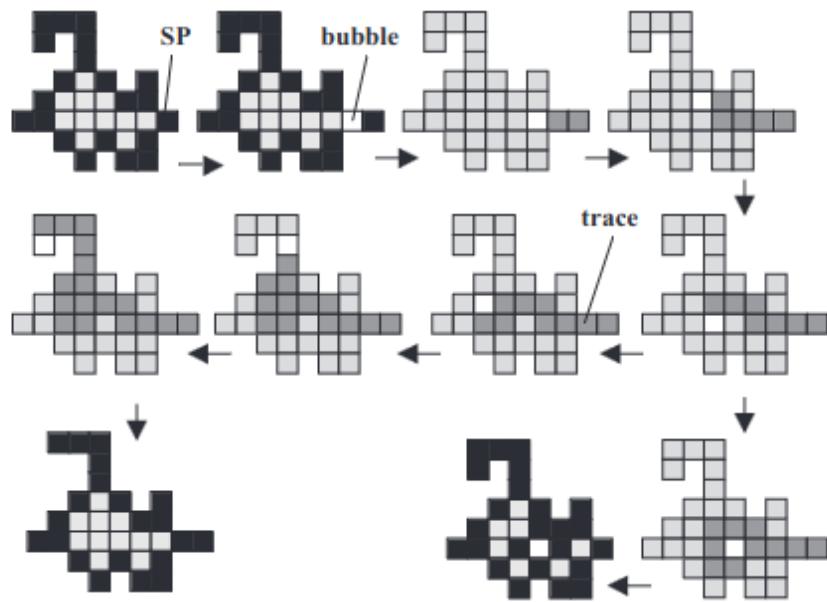


Figura 22: Reglas del modelo de célula mínima, mostrando los estados de los sitios y las interacciones. [37]

3.2. Robots para Monitoreo Poblacional

En esta sección, presentamos un resumen de los trabajos previos relacionados con robots o dispositivos aéreos para monitoreo poblacional, ya sea de animales o de humanos.

3.2.1. Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis

El artículo '*Uso de vehículos aéreos no tripulados (VANT's) para el monitoreo y manejo de los recursos naturales: una síntesis*' proporciona una base robusta para contextualizar la tesis titulada "*Diseño de un autómata para monitoreo*" en el estado del arte. La revisión exhaustiva que presenta este artículo sobre la utilización de drones en diversas aplicaciones de monitoreo y manejo de recursos naturales en América Latina es directamente relevante para esta investigación, ya que ambos trabajos se enfocan en el desarrollo y aplicación de tecnologías autónomas para la recolección de datos ambientales [38].

En primer lugar, el artículo destaca cómo los drones, equipados con una variedad de sensores como RGB, infrarrojos, multiespectrales, hiperespectrales y LIDAR, han revolucionado la capacidad de los investigadores para monitorear con precisión diversos aspectos del medio ambiente. Estos avances permiten obtener datos de alta resolución espacial de manera rápida y a un costo reducido, lo cual es crucial para la efectividad y eficiencia del monitoreo ambiental. En el contexto de la tesis, el diseño de un autómata para monitoreo puede beneficiarse enormemente de estos conocimientos y tecnologías, aplicando principios similares de autonomía y precisión en la recolección de datos.

Además, el artículo proporciona una visión detallada de las ventajas y limitaciones de diferentes tipos de drones y sensores, ofreciendo información valiosa que puede influir en las decisiones de diseño y selección de componentes para el autómata. Por ejemplo, la comprensión de las capacidades y restricciones de los sensores hiperespectrales y LIDAR puede guiar la integración de tecnologías adecuadas en el sistema de monitoreo, asegurando una recopilación de datos eficiente y precisa. Esta información es crucial para el diseño de sistemas que requieran

alta resolución y precisión en la medición de variables ambientales.

3.2.2. Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields

El artículo '*Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields*' proporciona una base sólida para contextualizar la tesis titulada '*Diseño de un autómata para monitoreo*' en el estado del arte. La investigación presentada en este documento aborda el desarrollo y evaluación de varios algoritmos de muestreo para robots móviles, destinados a la detección de insectos en campos agrícolas, una problemática directamente relevante para la tesis, que se centra en el diseño de un autómata para el monitoreo ambiental. [39]

En primer lugar, el artículo destaca la importancia de los algoritmos de muestreo para maximizar la eficiencia en la detección de plagas, considerando las limitaciones de recursos como el tiempo y la energía. Los algoritmos desarrollados, tanto aquellos que operan sin información previa como los que utilizan datos en tiempo real, ofrecen estrategias para optimizar la recolección de datos en entornos agrícolas. En el contexto de la tesis, estos conocimientos pueden ser aplicados al diseño del autómata, integrando algoritmos de muestreo dinámico que prioricen puntos de muestreo estratégicos basados en patrones de distribución de plagas, mejorando así la eficiencia y precisión del monitoreo. [39]

Además, el artículo proporciona una evaluación detallada de la efectividad de estos algoritmos en diferentes escenarios de simulación, considerando variables como el tamaño del campo y la tasa de propagación de insectos. Esta información es crucial para la tesis, ya que ofrece una comprensión profunda de cómo los diferentes algoritmos pueden ser implementados y ajustados según las condiciones específicas del entorno de monitoreo. La integración de estas estrategias en el diseño del autómata permitirá una adaptación más rápida y precisa a las condiciones cambiantes del campo, asegurando una detección temprana y gestión eficaz de plagas.

El artículo también incluye estudios de caso basados en datos reales de infestaciones de insectos, proporcionando ejemplos prácticos y lecciones aprendidas que pueden ser directamente aplicables a la tesis. Estos estudios demuestran cómo la implementación de algoritmos de muestreo dinámico ha llevado a mejoras significativas en la eficiencia y precisión del monitoreo, validando la relevancia y aplicabilidad del diseño del autómata en contextos agrícolas reales. [39]

3.2.3. The Role of Robots in Environmental Monitoring

El artículo "The Role of Robots in Environmental Monitoring" por Robert Bogue, publicado en Industrial Robot: An International Journal, detalla la creciente utilización de sistemas robóticos en la monitorización ambiental. El artículo comienza con una introducción sobre la importancia creciente de los robots en este campo, seguida de un examen exhaustivo de varios tipos de sistemas robóticos utilizados para fines ambientales. Destaca los roles de los robots aéreos en la monitorización de la contaminación atmosférica y discute las capacidades de los robots de superficie y submarinos en la monitorización de ambientes acuáticos. Además, proporciona ejemplos de aplicaciones de monitorización robótica terrestre. El documento concluye resumiendo los avances y contribuciones significativas de los robots en la provisión de una cobertura de datos espaciales y temporales mejorada, la detección de contaminación, la caracterización de condiciones ambientales y la localización de actividades ilícitas.

Los robots han mejorado significativamente los métodos tradicionales de monitorización ambiental, ofreciendo datos con mayor precisión y cobertura. El uso de drones equipados con dispositivos de imagen y sensores pequeños y ligeros ha revolucionado la detección de contaminantes en el aire y la caracterización de entornos acuáticos y terrestres. Además, la integración de técnicas de IA ha mejorado la eficiencia y efectividad de las imágenes de drones ambientales. El artículo también enfatiza la importancia de los robots acuáticos en la monitorización de entornos de agua dulce y marinos, desde despliegues locales a corto plazo hasta misiones oceánicas de larga duración. En general, el artículo ofrece una visión general exhaustiva de las diversas y crecientes aplicaciones de los robots en la monitorización ambiental, subrayando su papel crítico en la ciencia ambiental moderna. [40]

4. Propuesta a desarrollar

El proyecto propuesto consiste en el desarrollo de un sistema de monitoreo poblacional basado en la implementación de un autómata en un modelo bidimensional no lineal. En este caso, como mencionamos anteriormente, el algoritmo esta basado en el modelo de Physarum Polycephalum. Este modelo es un organismo unicelular que se comporta como un autómata celular, y es capaz de resolver problemas de optimización y ruteo.

A su vez, el sistema propuesto se basa en la utilización de robots autónomos, los cuales se encargarán de recolectar información de la población y de los entornos en los que se encuentran. Estos robots estarán equipados con cámaras y sensores que les permitirán detectar y clasificar entidades poblacionales. Además, los robots estarán conectados a una red de comunicación que les permitirá compartir información en tiempo real.

El sistema funcionará de la siguiente manera: los robots autónomos recibirán la ruta a seguir por parte de nuestro simulador del Physarum Polycephalum, el cual se encargará de determinar la ruta óptima para recolectar información de la población. Una vez que los robots recolecten la información, esta sera enviada a un servidor central, el cual se encargara de procesar la información y de generar reportes en tiempo real.

La implementación de este sistema permitirá a los investigadores y a las autoridades locales monitorear poblaciones de manera eficiente y en tiempo real. Además, el sistema permitirá la detección de cambios en las poblaciones y en los entornos en los que se encuentran.

Por ello tendremos principalmente dos 'productos' a desarrollar, el primero será el simulador del Physarum Polycephalum, el cual será un sistema que permitirá determinar rutas óptimas para recolectar información de la población. El segundo producto será el sistema de monitoreo poblacional, el cual será un sistema que permitira a los robots autónomos recolectar información de la población y de los entornos en los que se encuentran.

Se detallará la implementación de estos sistemas en las siguientes secciones.

4.1. Requerimientos

Para el desarrollo de nuestro Trabajo Terminal (TT), es necesario establecer los requerimientos que debe cumplir el robot que simulará el comportamiento del Physarum Polycephalum. En las siguientes secciones, se describirán los requerimientos funcionales y no funcionales que se deben cumplir en nuestro Trabajo Terminal (TT).

4.1.1. Requerimientos Funcionales

En esta sección se presentan los requerimientos funcionales que definen las características y capacidades específicas que el sistema debe proporcionar para cumplir su propósito en el contexto de simulación y monitoreo de rutas automatizadas. Estos requerimientos aseguran que el sistema cumpla con las funciones clave necesarias para la interacción y operación de la simulación. Los requerimientos funcionales se listan en el Cuadro 3.

ID	Requerimiento Funcional
RF1	El sistema debe permitir al usuario seleccionar estados en el simulador mediante el teclado y el ratón.
RF2	El sistema debe permitir al usuario colocar los estados inicial y final en el lienzo antes de iniciar la simulación.
RF3	El sistema debe iniciar la simulación de rutas al presionar la tecla ENTER.
RF4	El sistema debe permitir la carga de un mapa o imagen en el lienzo para definir el entorno inicial de la simulación.
RF5	El sistema debe permitir al usuario visualizar la ruta generada en tiempo real durante la simulación.

Cuadro 3: Requerimientos Funcionales del Sistema

4.1.2. Requerimientos no Funcionales

En esta sección se describen los requerimientos no funcionales, los cuales establecen los criterios de calidad y desempeño que el sistema debe cumplir para garantizar una operación robusta, eficiente y compatible en distintos entornos. Estos requerimientos no funcionales aseguran que el sistema sea eficiente, adaptable y compatible, proporcionando una experiencia de usuario satisfactoria. Los requerimientos no funcionales se detallan en el Cuadro 4.

ID	Requerimiento No Funcional
RNF1	El sistema debe ser eficiente en términos de tiempo de simulación para optimizar el tiempo de generación de cada iteración.
RNF2	El sistema debe ser capaz de manejar simulaciones en mapas grandes y con obstáculos sin pérdida significativa de rendimiento.
RNF3	El sistema debe ser compatible tanto con sistemas operativos Windows como Linux.

Cuadro 4: Requerimientos No Funcionales del Sistema

4.2. Diagramas

En esta sección se presentan todos los diagramas necesarios para la implementación del sistema propuesto. En primer lugar, se presenta un diagrama de arquitectura del sistema propuesto, el cual muestra la interacción entre los diferentes componentes del sistema.

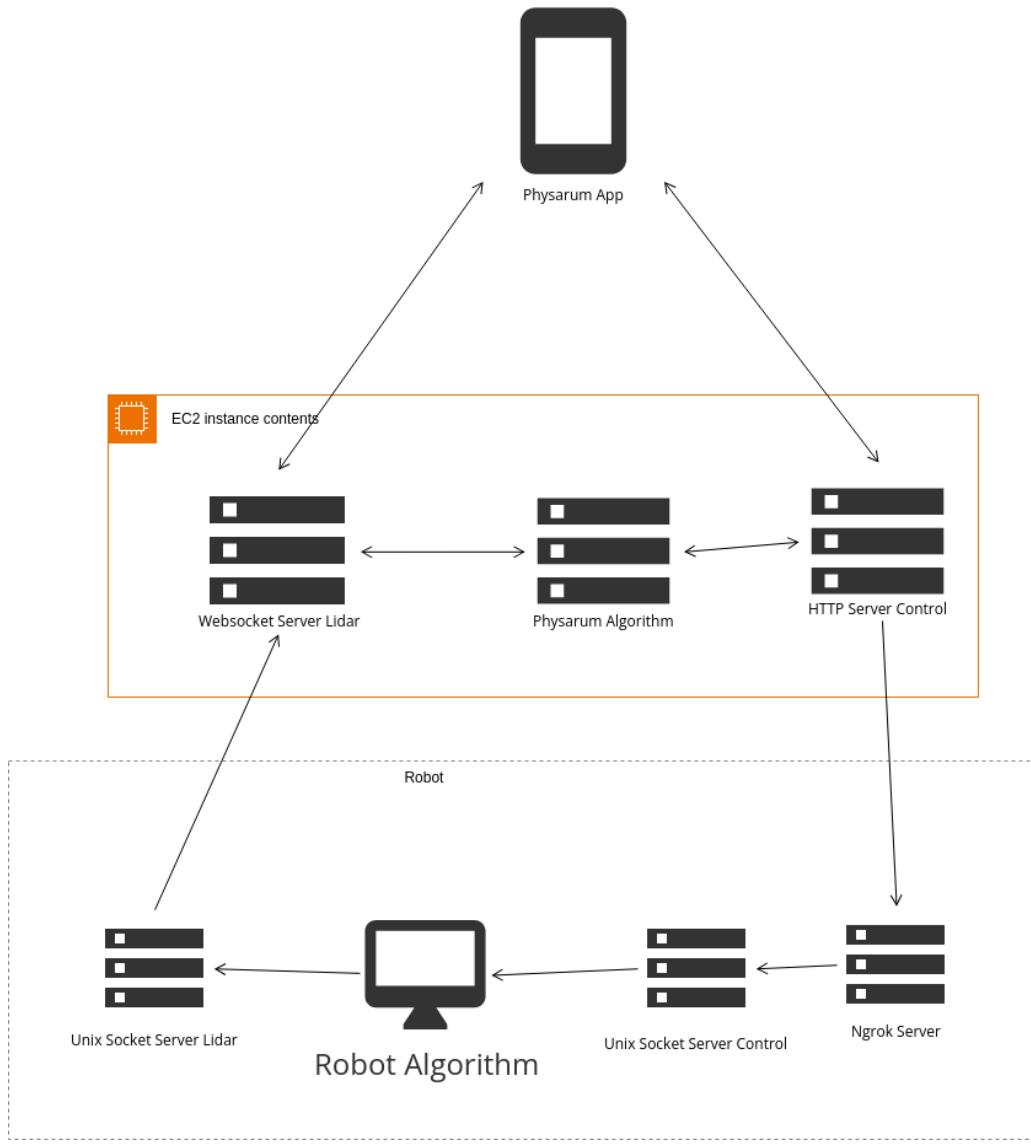


Figura 23: Diagrama de arquitectura del sistema propuesto.

Como se muestra en la Figura 23, el sistema mostrado en el diagrama representa una arquitectura distribuida implementada para el control remoto de un robot mediante un algoritmo basado en **Physarum polycephalum** y la utilización de datos obtenidos por un sensor LiDAR. La arquitectura se compone de tres capas principales: la capa de la aplicación móvil, la capa de procesamiento en la nube y la capa física del robot.

La **Physarum App** es la interfaz de usuario a través de la cual se envían los comandos al sistema, los cuales son gestionados por un servidor Protocolo de Transferencia de Hipertexto (Hypertext Transfer Protocol, HTTP) ubicado en una instancia EC2 de Amazon Web Services. Esta instancia contiene tres servidores: un servidor HTTP que recibe los comandos de control, un servidor WebSocket que maneja la recepción de los datos del sensor LiDAR enviados des-

de el robot, y el núcleo del sistema, el algoritmo de **Physarum**, encargado de procesar esta información para la toma de decisiones en tiempo real sobre el comportamiento del robot.

En la capa física, el robot está controlado mediante dos servidores Unix Socket. El **Unix Socket Server LiDAR** recibe y transmite los datos del sensor LiDAR al servidor WebSocket en la instancia EC2, mientras que el **Unix Socket Server Control** se encarga de recibir los comandos procesados y enviarlos al robot. Además, se utiliza un servidor Ngrok que permite la conexión remota segura, facilitando el control del robot desde la aplicación móvil.

Esta arquitectura distribuida permite la integración fluida de los componentes del sistema, asegurando una correcta interacción entre los datos sensoriales, el procesamiento en la nube y el control remoto del robot.

Después de la presentación de la arquitectura del sistema, se presentan los diagramas de flujo de las Figuras 24 - 26.

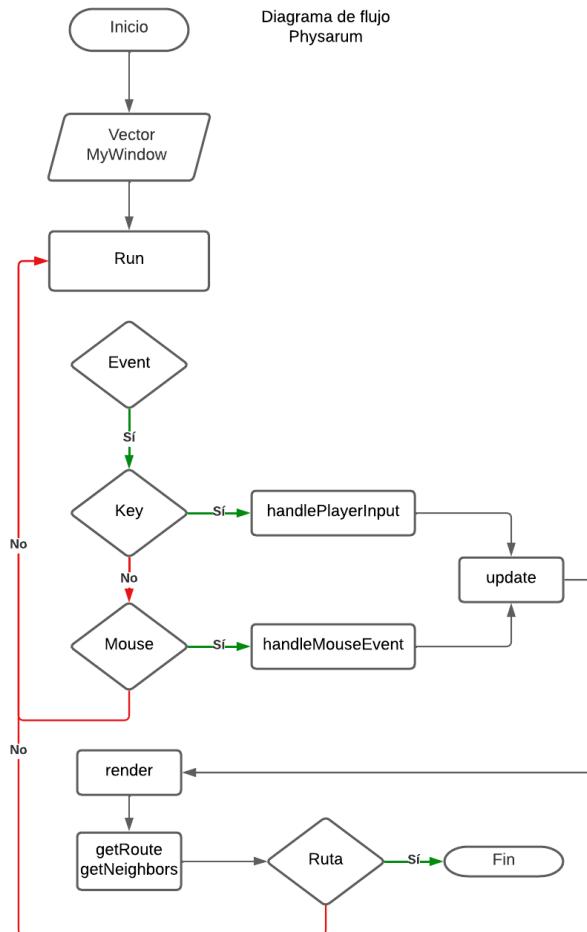


Figura 24: Diagrama de flujo del simulador de **Physarum polycephalum**.

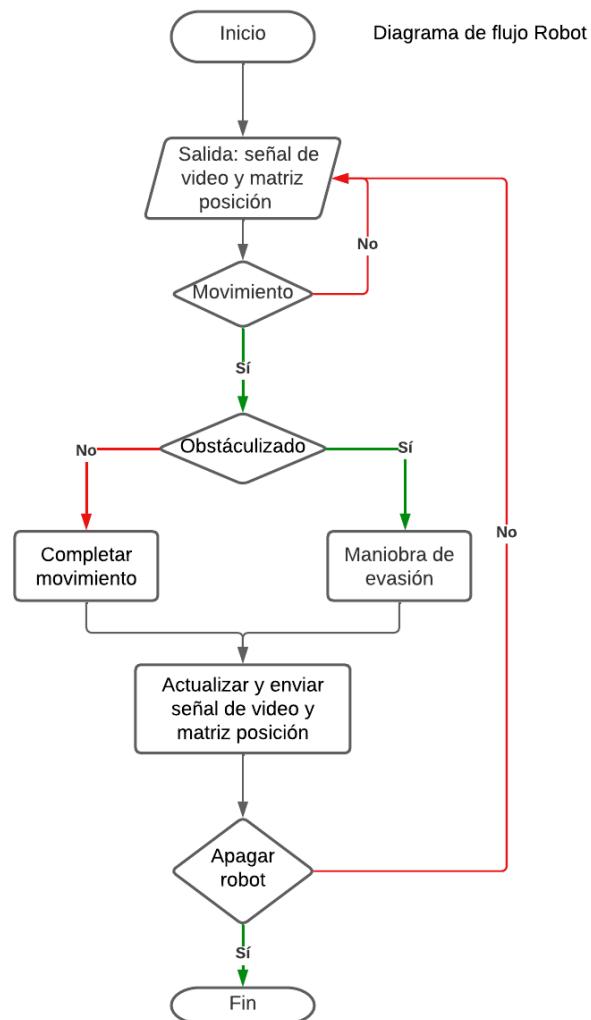


Figura 25: Diagrama de flujo del sistema del robot autónomo.

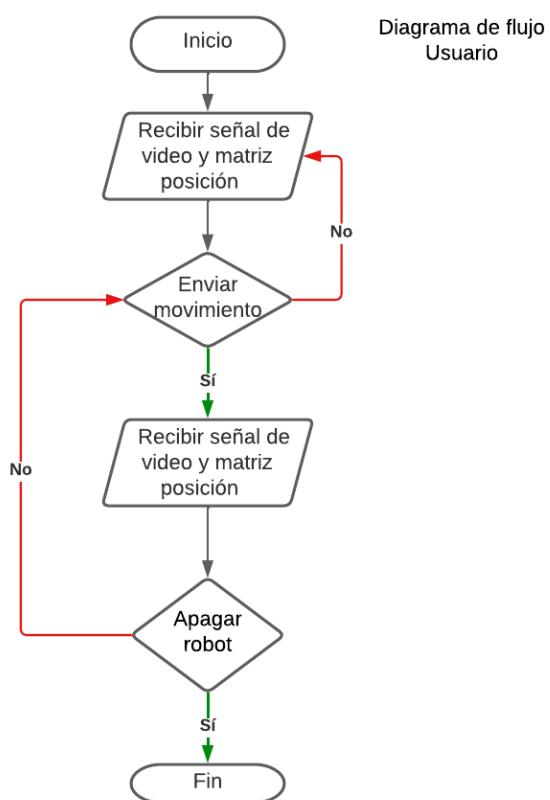


Figura 26: Diagrama de flujo de la aplicación móvil.

Después de la presentación de los diagramas de flujo, en la Figura 27 se presenta el diagrama de clases del sistema propuesto.

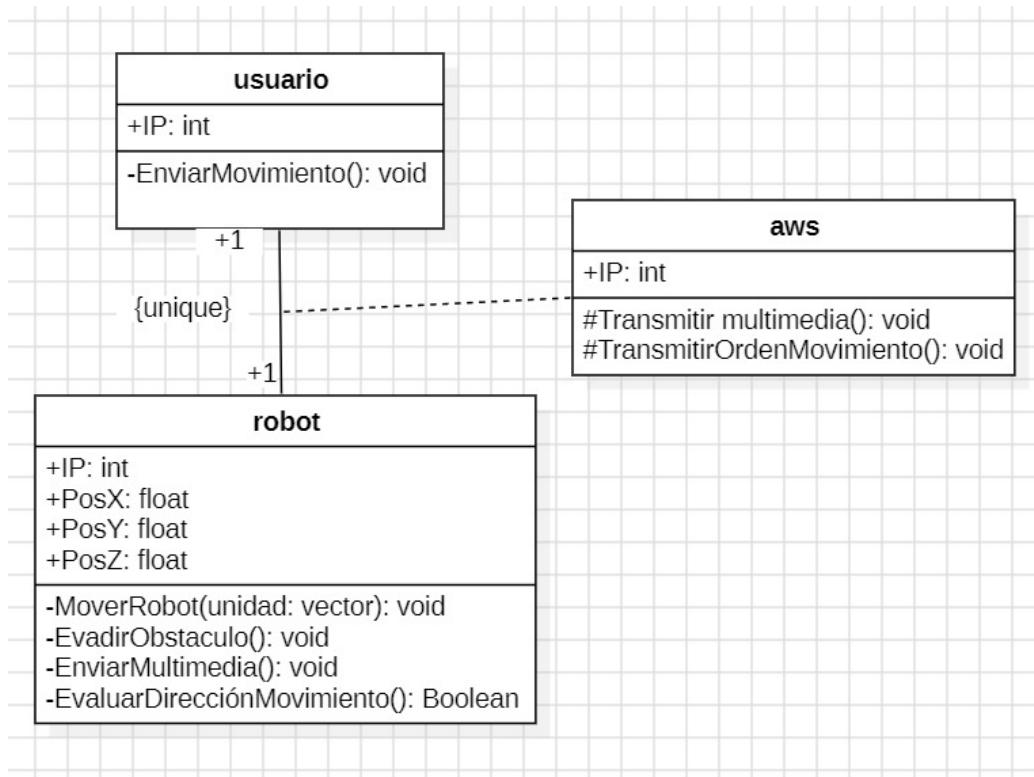


Figura 27: Diagrama de clases del sistema propuesto.

El diagrama de clases muestra la interacción entre el usuario, el robot y un servidor en Amazon Web Services (AWS). El **usuario** tiene un atributo para su dirección IP y un método *EnviarMovimiento()* que le permite enviar órdenes al **robot**. Este último, con atributos para su IP y su posición en los ejes X, Y y Z, tiene métodos para moverse (*MoverRobot()*), evadir obstáculos (*EvadirObstaculo()*), transmitir multimedia (*EnviarMultimedia()*) y evaluar su dirección de movimiento (*EvaluarDirecciónMovimiento()*).

El **servidor en AWS** cuenta con métodos para transmitir multimedia y órdenes de movimiento hacia el robot. El servidor actúa como intermediario entre el robot y el sistema, facilitando la transmisión de datos y el control remoto de manera segura y eficiente. El sistema asegura una comunicación única y directa entre cada usuario y su robot, manteniendo una interacción fluida y controlada.

En la figura 28 se presenta el diagrama de casos de usos del sistema propuesto.

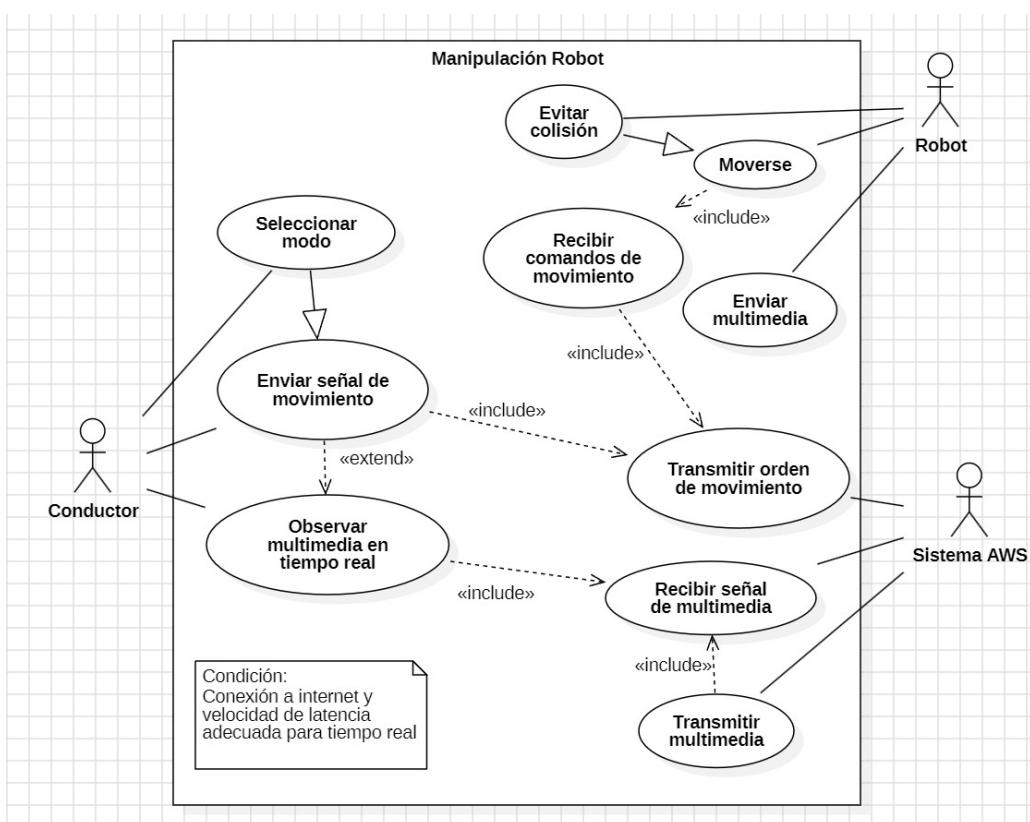


Figura 28: Diagrama de casos de uso del sistema propuesto.

Y el Cuadro 5 muestra la descripción del caso de uso **Manipulación del Robot**.

Cuadro 5: Descripción del Caso de Uso: Manipulación del Robot

Identificador	CU-01 Manipulación del Robot
Descripción	El conductor controla los movimientos del robot, seleccionando el modo de operación, enviando comandos de movimiento y observando la transmisión multimedia en tiempo real mientras AWS transmite las órdenes y multimedia.
Actores	Conductor, Robot, Sistema AWS
Precondiciones	<ul style="list-style-type: none"> ■ El sistema debe estar conectado a internet. ■ La latencia de la red debe ser adecuada para la transmisión en tiempo real.
Postcondiciones	<ul style="list-style-type: none"> ■ El robot ejecuta las órdenes de movimiento enviadas por el conductor. ■ El conductor puede observar la transmisión multimedia en tiempo real.
Secuencia Normal	<ol style="list-style-type: none"> A) El conductor selecciona el modo de operación del robot. B) El conductor envía una señal de movimiento. C) El sistema AWS recibe la señal y la transmite al robot. D) El robot recibe la señal y ejecuta el movimiento según las instrucciones. E) El robot evita colisiones mientras se mueve. F) El robot transmite la señal multimedia en tiempo real. G) El conductor observa la multimedia transmitida en tiempo real.

Excepciones	<p>A) Si la conexión a internet se pierde o es inestable, el sistema notifica al conductor sobre la interrupción en la transmisión.</p> <p>B) Si el robot detecta un obstáculo ineludible, detiene su movimiento y espera nuevas instrucciones.</p>
Rendimiento	<ul style="list-style-type: none"> ■ El sistema debe procesar las órdenes de movimiento en menos de 1 segundo. ■ La transmisión de la señal multimedia debe tener un máximo de 300ms de latencia.
Frecuencia	Se espera que este caso de uso se realice continuamente durante la operación del robot.
Importancia	Vital
Urgencia	Inmediata, ya que el sistema debe responder en tiempo real.
Comentarios	El sistema depende de la calidad de la conexión a internet para mantener la comunicación en tiempo real entre el conductor y el robot.

Finalmente, en la Figura 29 se presenta el diagrama de secuencia del sistema propuesto.

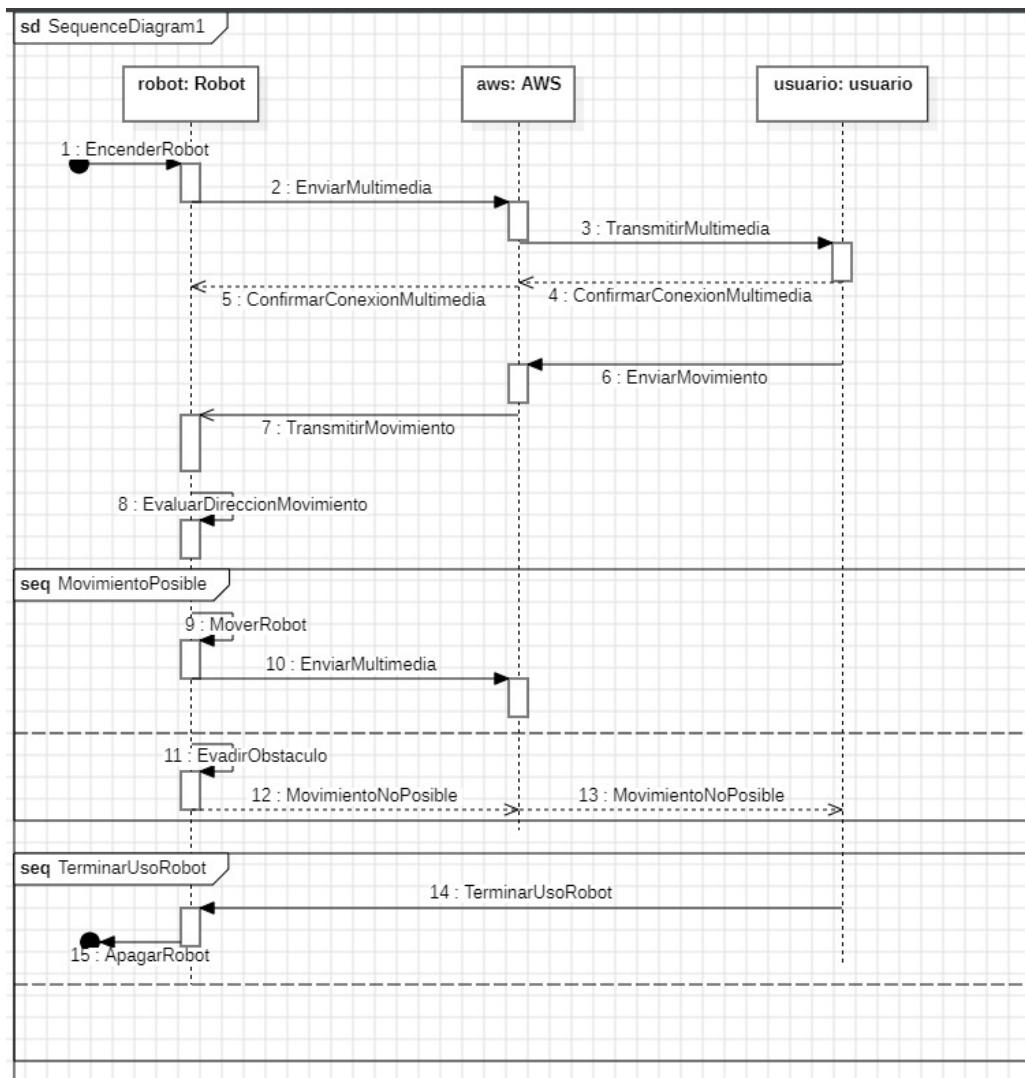


Figura 29: Diagrama de secuencia del sistema propuesto.

El diagrama de secuencia muestra la interacción entre el **robot**, el **servidor AWS**, y el **usuario**. Comienza cuando el usuario enciende el robot, lo que desencadena la transmisión de multimedia desde el robot hacia AWS, que luego retransmite al usuario. Tras esto, se confirma la conexión de multimedia entre el robot y AWS, asegurando que el sistema esté listo para recibir comandos.

El usuario puede entonces enviar órdenes de movimiento al robot a través de AWS. El robot evalúa la dirección del movimiento, y si es posible, procede a moverse mientras sigue enviando multimedia. En caso de encontrar un obstáculo, el robot intenta evadirlo. Si el movimiento no es posible, el robot notifica al usuario sobre la situación.

Finalmente, cuando el usuario desea terminar la sesión, se envía la señal para cerrar el uso del robot, lo que lleva al apagado del mismo. Así, el diagrama cubre desde el encendido y transmisión de multimedia hasta la ejecución de movimientos y finalización del uso del robot.

5. Implementación

Como mencionamos en el Marco Teórico, nos centraremos en el simulador del *Physarum Polycephalum* y en el robot con una Raspberry Pi. En este caso, el simulador del *Physarum Polycephalum* se encargará de determinar la ruta óptima para recolectar información de la población. Por otro lado, el robot con una Raspberry Pi se encargará de recolectar información de la población y de los entornos en los que se encuentran.

En este capítulo, se detallará la implementación de estos sistemas. Primero, se describirá la implementación del simulador del *Physarum Polycephalum*. Luego, se describirá la implementación del robot con una Raspberry Pi. Finalmente, se describirá la integración de estos sistemas.

5.1. Simulador del *Physarum Polycephalum*

El algoritmo propuesto en este documento está inspirado en el modelo de agentes de Jones [36]. El algoritmo es un autómata celular que simula el comportamiento de *Physarum polycephalum* en un laberinto, por lo que necesitamos definir algunos conceptos. Sea \mathbb{Z} el conjunto de los números enteros, y definamos la longitud de una tupla x como $|x|$. Para todas las tuplas x y y donde $|x| = |y|$, denotamos $x \oplus y$ como el resultado de la suma de cada componente de x y y , es decir, $(x \oplus y)_i = x_i + y_i$ para todo $i \in \mathbb{Z}$.

Un autómata celular se define como una tupla (\mathbb{Z}^n, S, N, f) donde n es la dimensión tal que $n \in \mathbb{Z}^+$, S es un conjunto de estados finito y no vacío, N es un conjunto no vacío y finito de vecindarios pertenecientes a \mathbb{Z}^n , y f es una función de transición local, es decir, $f : S^N \rightarrow S$ donde S^N representa el conjunto de todas las configuraciones posibles de vecindarios en N .

Así, el algoritmo propuesto en este trabajo se define como un autómata celular (\mathbb{Z}^2, S, N, f) donde $n = 2$, $S = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $N = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}^9$, y $f : \{0, 1, 2, 3, 4, 5, 6, 7, 8\}^9 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$.

Sea $P = (C(x, y : t), N(x, y : t), M(x, y : t))$ el estado combinado, vecindario y memoria de la célula en la posición (x, y) en el tiempo t . La función de transición f , que actualiza el estado de la célula central en la siguiente generación, se define de la siguiente manera:

$$f(P) = \begin{cases} 7 & \text{if } C(x, y : t) = 0 \\ & \wedge \exists N_i \in \{3, 4, 6\} \\ & \wedge M(x, y : t) = 0 \\ 6 & \text{if } C(x, y : t) = 1 \\ & \wedge \exists N_i \in \{5, 6\} \\ 5 & \text{if } C(x, y : t) = 4 \\ & \wedge \exists N_i \in \{3, 5, 6\} \\ & \wedge M(x, y : t) = 0 \\ & \wedge N_i \notin \{0, 7\} \\ 0 & \text{if } C(x, y : t) = 5 \\ & \wedge M(x, y : t) \notin \{5, 8\} \\ & \wedge N_i \notin \{1, 3, 4, 6\} \\ 8 & \text{if } C(x, y : t) = 5 \\ & \text{and the above condition is not met} \\ 4 & \text{if } C(x, y : t) = 7 \\ & \wedge \exists N_i \in \{3, 4, 6\} \\ 5 & \text{if } C(x, y : t) = 8 \\ C(x, y : t) & \text{otherwise} \end{cases}$$

Donde los estados del autómata celular se definen en el **Cuadro 6**.

Color	Estado	Descripción
	0	Campo libre
	1	Nutriente no encontrado
	2	Repelente
	3	Punto inicial
	4	Gel contrayéndose
	5	Gel compuesto
	6	Nutriente encontrado
	7	Expansión de Physarum
	8	Gel no compuesto

Cuadro 6: Estados del autómata celular

En la fuente mencionada, se detalla el algoritmo básico de *Physarum Polycephalum*, diseñado originalmente para con sistemas multiagentes. Sin embargo, en la versión propuesta aquí, optamos por realizar un autómata celular de 2 dimensiones con la vecindad de Moore, facilitando así el acceso a un mayor número de vecinos para comparación y permitiendo obtener una perspectiva más clara de la dirección óptima para el desplazamiento del agente.

Sin embargo, el uso del vecindario de Moore en lugar del vecindario de von Neumann introduce ciertos desafíos no presentes en el algoritmo original. Uno de estos desafíos surge en las esquinas (NW, NE, SW, SE), donde el repelente podría permitir que el agente escape, contrario a lo deseado. Para abordar este inconveniente, se implementó una solución que consiste en colocar

un repelente imaginario en la esquina cuando dos esquinas adyacentes presentan repelentes en un ángulo de 90° entre sí. Este ajuste permite la creación de una gama más amplia de formas, como se ilustra en las Figuras 30, 31 y 32. En estas imágenes, el número total de células es de 50×50 .

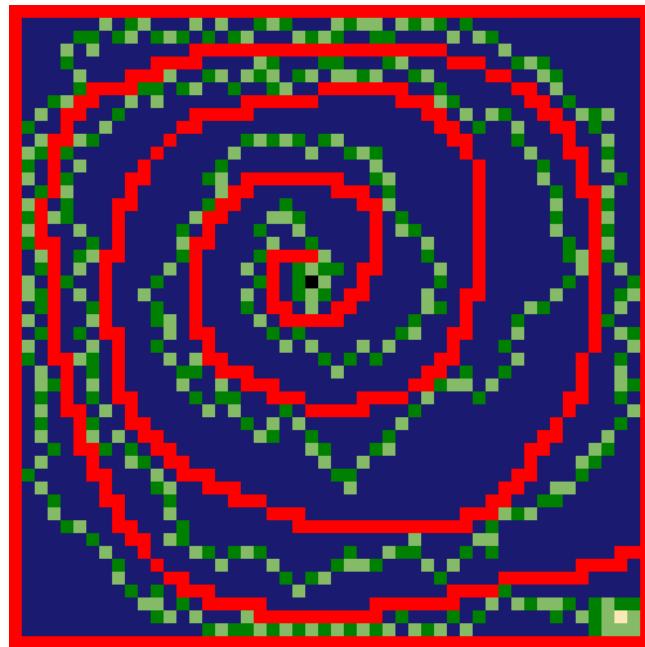


Figura 30: *Physarum Polycephalum* resolviendo un laberinto en espiral.

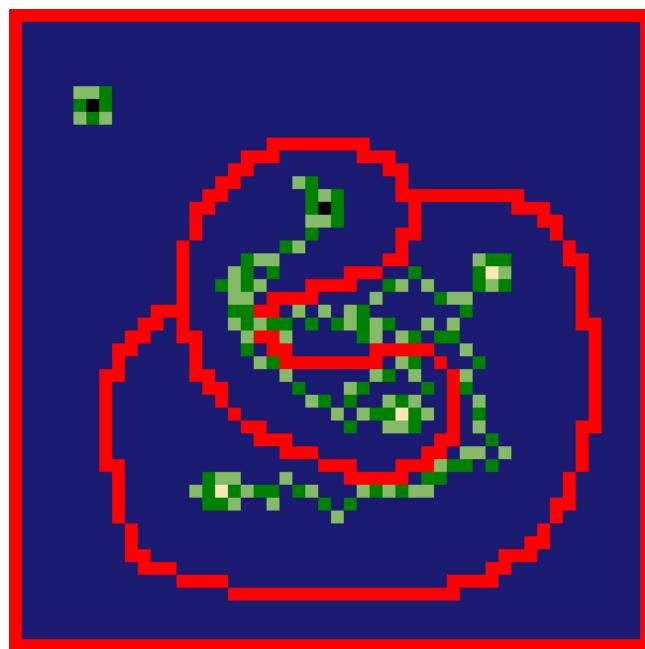


Figura 31: *Physarum Polycephalum* resolviendo un laberinto de tipo circular.

Gracias a la resolución de la fuga en las esquinas por nuestro algoritmo, es posible generar mapeos más diversos de cuevas y catacumbas. Este enfoque mejora significativamente nuestra comprensión de la topografía del área explorada. Además, la diversidad en el mapeo facilita la identificación del número y variedad de caminos disponibles, lo que se ha implementado

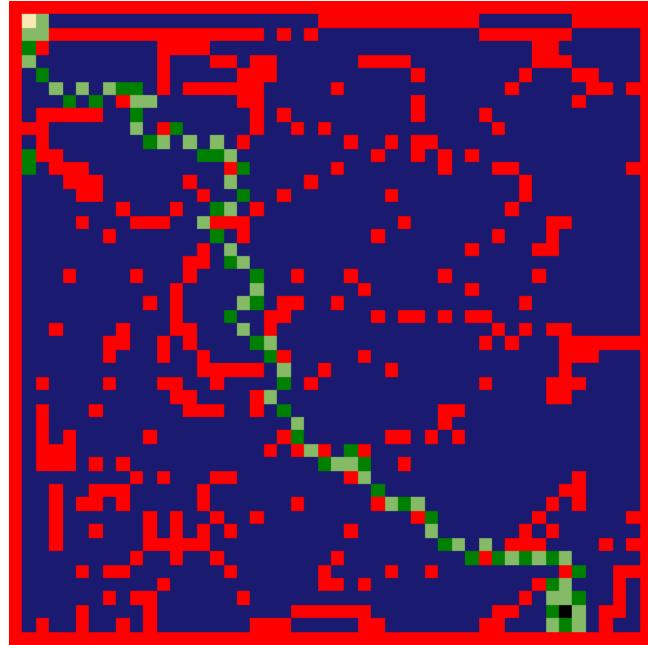


Figura 32: *Physarum Polycephalum* resolviendo un laberinto con obstáculos.

mediante un algoritmo de mapeo de imágenes que ayuda en la representación gráfica de dicha topografía, como se muestra en la Figura 33.

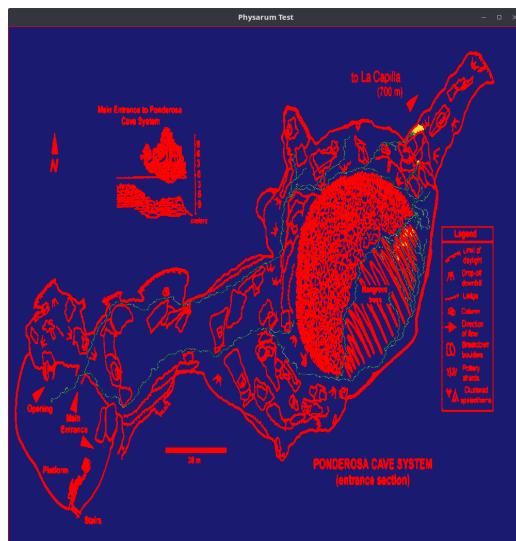


Figura 33: Mapeo del sistema de cuevas usando *Physarum Polycephalum*.

También el algoritmo ha sido probado en un entorno real, donde ha sido capaz de generar rutas óptimas en la Catacumba de París, como se muestra en la Figura 34. El espacio explorado por el algoritmo es de 1000 x 1000 células.

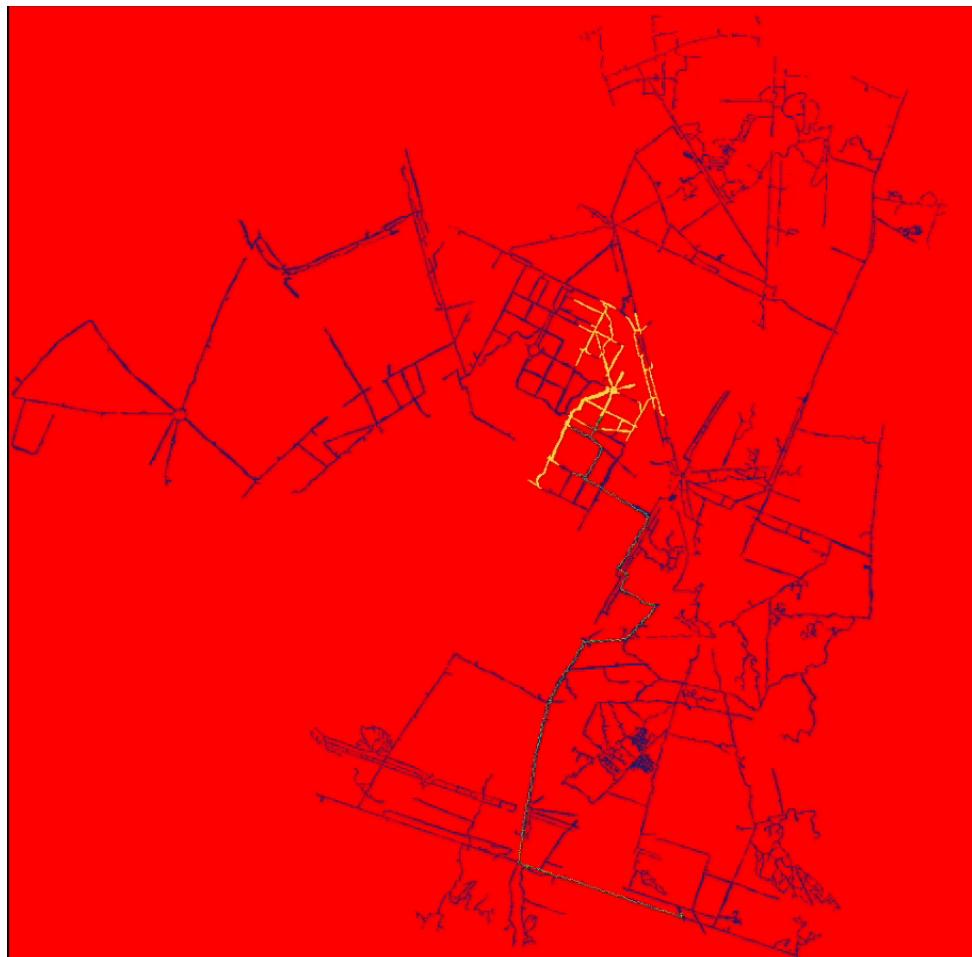


Figura 34: Mapeo de la catacumba usando *Physarum Polycephalum*, utiliza 5264 pasos para obtener la ruta.

Como se puede observar en los Cuadros 7 y 8, el algoritmo propuesto es capaz de resolver laberintos de manera eficiente, generando rutas óptimas en entornos complejos y desconocidos. Además, el algoritmo es capaz de adaptarse a diferentes topografías, lo que lo convierte en una herramienta versátil para la exploración de entornos desconocidos y nos es de ayuda en el monitoreo de poblaciones y sistemas relacionados.

Color	State	Initial State Catacomb
Dark Blue	0	32,891
Light Blue	1	1
Red	2	967,105
Black	3	2
Yellow	4	0
Green	5	0
Light Yellow	6	0
Grey	7	0
Green	8	0

Cuadro 7: Estado inicial de la cataumba

Color	State	Initial State Cave
Dark Blue	0	853,861
Light Blue	1	3
Red	2	146,135
Black	3	1
Yellow	4	0
Green	5	0
Light Yellow	6	0
Grey	7	0
Green	8	0

Cuadro 8: Estado inicial de la cueva

Cabe señalar que dado que el algoritmo está bioinspirado, la función que simula el comportamiento del plasmodio se asigna de manera pseudoaleatoria a un vecino adyacente, con una probabilidad de $1/8$. Esta característica permite que la expansión del algoritmo tome una forma circular en lugar de una expansión cuadrada o lineal. Sin embargo, al modificar la función de probabilidad, es posible lograr una expansión más irregular en lugar de simplemente circular.

En cuanto la implementación del algoritmo, se ha utilizado el lenguaje de programación C++ y la librería OpenCV para la obtención de imágenes. La parte de las esquinas que mencionamos con anterioridad se implementó como se muestra en el Listing 1:

Listing 1: Implementación del problema de las esquinas

```

1      std::vector<int> Physarum::isOnCorner(std::vector<int>
2          neighboursData) {
3      std::vector<int> corners;
4      if (neighboursData[0] == 2 && neighboursData[2] == 2) {
5          corners.push_back(1);
6      }
7      if (neighboursData[0] == 2 && neighboursData[6] == 2) {
8          corners.push_back(7);
9      }
10     if (neighboursData[6] == 2 && neighboursData[4] == 2) {
11         corners.push_back(5);
12     }
13     if (neighboursData[2] == 2 && neighboursData[4] == 2) {
14         corners.push_back(3);
15     }
16     return corners;
}

```

En el Listing 1, se muestra la implementación de la función que detecta si el agente se encuentra en una esquina. La función recibe un vector de enteros que representa los estados de los vecinos adyacentes. Si dos esquinas adyacentes presentan repelentes, la función devuelve un vector con las esquinas en las que se encuentra el agente. En caso contrario, la función devuelve un vector vacío.

Por ello podemos decir que el algoritmo propuesto es capaz de resolver laberintos de manera eficiente, generando rutas óptimas en entornos complejos y desconocidos. Además, el algoritmo es capaz de adaptarse a diferentes topografías, lo que lo convierte en una herramienta versátil para la exploración de entornos desconocidos y nos es de ayuda en el monitoreo de poblaciones y sistemas relacionados.

5.1.1. Generación de rutas de nuestro simulador

El robot de monitoreo deberá de seguir una ruta preestablecida, la cual es calculada por el algoritmo del Physarum Polycephalum en el simulador que ha sido desarrollado.

Primeramente, se tiene un lienzo, el cual es representado por un arreglo de celdas de tamaño $n \times n$, donde n es el tamaño deseado para la representación del espacio en el cual el Physarum calculará la ruta una vez terminada la simulación de este.

En el lienzo, se colocan los estados sobre el lienzo por medio del teclado y el mouse, donde en el teclado son presionadas las teclas de 1 al 9 para poder elegir cada uno de los estados que puede tomar la celda en la cual se haya presionado el botón izquierdo del mouse. Los estados que tienen mayor relevancia y que son los que se deben de colocar para poder realizar la simulación correctamente son los 1 y 4, debido a que representan el nutriente como el punto inicial respectivamente. El punto inicial es de donde se empezará con la expansión del Physarum, mientras que el nutriente es el destino final, puesto que una vez encontrado, cambiara su estado al estado 6, el cual es correspondiente al estado de nutriente encontrado. Y una vez finalizada la simulación, el algoritmo se detendrá automáticamente y quedará plasmada la ruta por la cual

el Physarum encontró el o los nutrientes desde el punto inicial, la cual será enviada al robot para su posterior seguimiento para llegar a su destino en el mundo real.

Al iniciar el programa, se despliega una pantalla la cual muestra el lienzo con el espacio que hemos predefinido anteriormente en el código. En este espacio al inicio se pueden colocar los diferentes estados como se mencionó anteriormente, por lo que una vez se haya colocado la configuración deseada, para iniciar la simulación se presiona en el teclado la tecla ENTER. El algoritmo se empieza a expandir, siendo aplicadas cada una de las reglas en cada una de las celdas del arreglo. El algoritmo termina cuando la ruta es encontrada y el Physarum termina de contraerse, dicha ruta es la que es enviada al robot para su seguimiento. Primeramente, se iniciaron las simulaciones en espacios pequeños. La primera configuración como se puede ver en la Figura 35 fue la de un lienzo de tamaño 10 x 10.

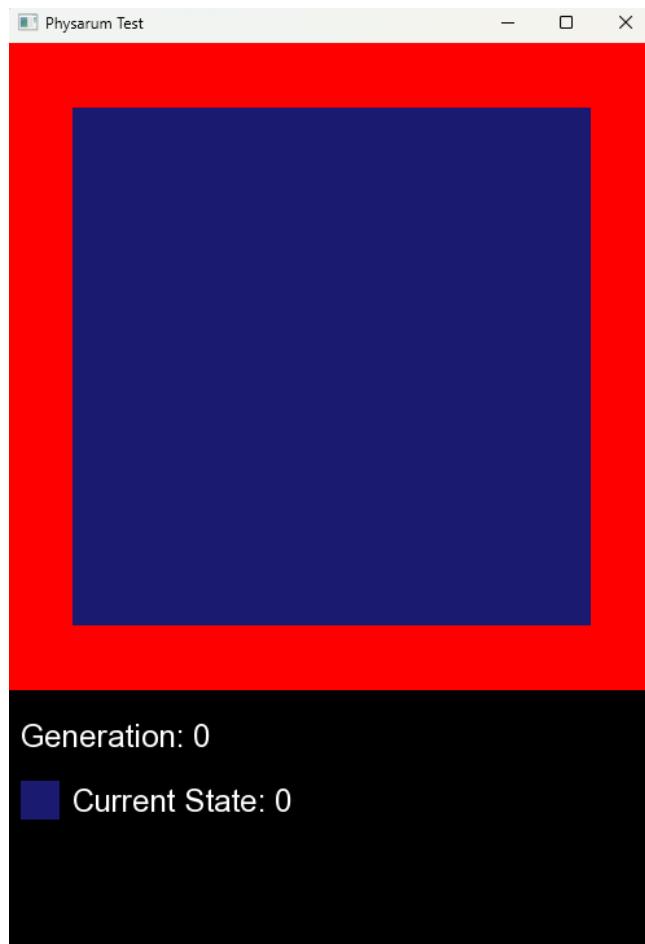


Figura 35: Lienzo de 10 x 10 celdas '1'

Posteriormente se colocaron los estados correspondientes a el estado inicial y al nutriente no encontrado. Como se puede ver en la Figura 36.

Una vez colocada la configuración inicial, entonces se procede a iniciar la simulación, con la cual, al terminar las iteraciones se genera una ruta que va desde el estado inicial al nutriente no encontrado, el cual para este momento ha cambiado su estado a nutriente encontrado, cambiando a su vez el color correspondiente a este estado. Como se puede ver en la Figura 37.

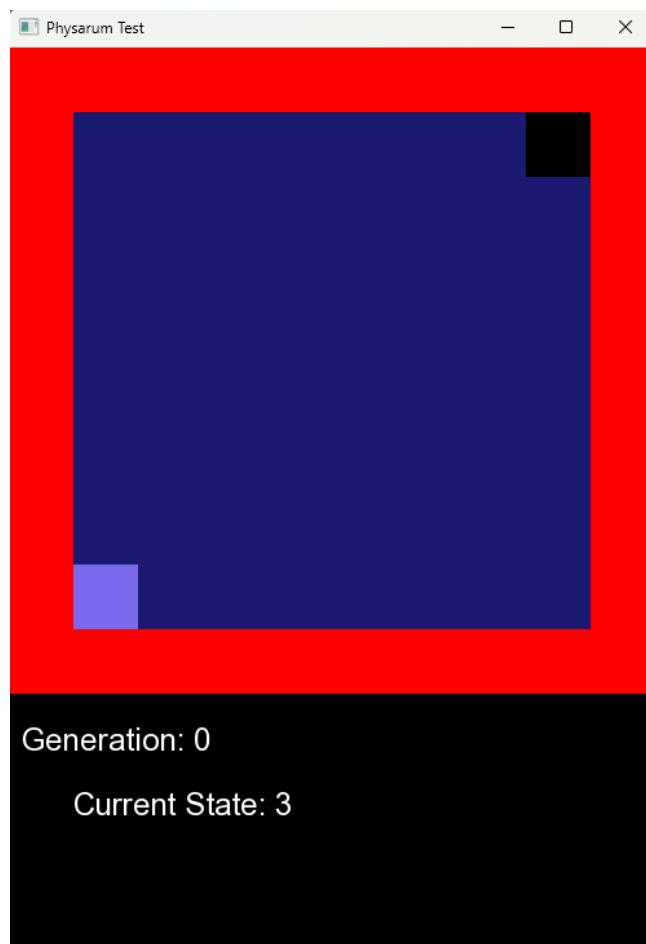


Figura 36: Estados iniciales de la simulación '1'



Figura 37: Ruta encontrada en la simulación '1'

En el programa es posible ver en qué generación es en la que se encuentra el algoritmo, además del estado en el cual en ese momento se tiene elegido en el teclado, esto para ser colocado en el lienzo.

Es posible colocar otra configuración una vez reiniciando el programa, donde se puede colocar los estados en distinta posición, por lo que se muestran a continuación distintas configuraciones una vez finalizadas las simulaciones correspondientes a cada una de ellas. Esto se puede ver en las Figuras 38 - 40.



Figura 38: Resultados de una segunda configuración

Las simulaciones realizadas nos han devuelto resultados satisfactorios, debido a que las rutas son correctamente generadas de acuerdo a las reglas plasmadas en el algoritmo, yendo del punto inicial al final, y finalizando una vez que el Physarum ya no tiene la necesidad de expandirse, debido a que ha encontrado el nutriente que estaba buscando.



Figura 39: Resultados de una tercera configuración



Figura 40: Resultados de una cuarta configuración

5.1.2. Codificación e implementación de algoritmo en el robot en la primera iteración 1

Con las rutas que han sido generadas de acuerdo a cada una de las simulaciones que fueron ejecutadas en el programa, como ya se ha mencionado, se genera la información con la cual el robot realizará el seguimiento de esta ruta para llegar de un punto inicial al final. Esta información corresponde a la ruta que genera el Physarum cuando termina la simulación de llegar desde su punto inicial de expansión a un nutriente con el cual se alimenta.

Para poder hacer una correcta manipulación de la información, la cual pueda ser enviada al robot que este pueda interpretarla y avanzar de acuerdo la ruta generada, es necesario crear un espacio el cual represente la posición del robot en un determinado lugar, junto con el destino al cual se quiere llegar, a partir de ahí pasar esta configuración al modelo del Physarum, ejecutar la simulación y obtener la ruta, finalizando con la recopilación de la información relacionada con la ruta y su almacenamiento, lo cual es representado en el Listing 2.

Listing 2: Pseudo Código de las rutas

```
1      Iniciar programa
2      Colocar la configuraci\'on inicial
3      Coloca punto inicial
4      Coloca punto final
5      Ejecutar Simulador_Physarum
6      Si Simulador_Physarum obtuvo ruta, entonces:
7          Guardar las coordenadas del punto inicial
8          Guardar las coordenadas del punto final
9          Guardar las coordenadas de las c\'elulas por orden de
           aparici\'on
10         Desplegar todas las coordenadas en un archivo
11     Si no
12         Finaliza programa
13     Fin Si
```

A partir de lo anterior, se comprende que al robot se le será enviado las coordenadas correspondientes a la ruta obtenida por el simulador del Physarum, con un punto inicial, el cual marcará su posición actual, su punto final, representando el destino al cual llegará el robot y las células del Physarum ordenadas por orden de aparición, las cuales son las coordenadas por las cuales el robot tendrá que pasar para llegar desde el punto inicial al final.

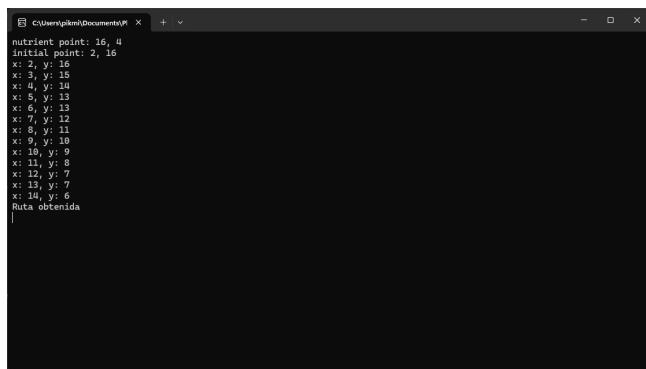
Con lo anterior, es posible la creación de información la cual el robot sea capaz de leer y a partir de sus propias funciones, llegar de un punto inicial al final. Esta información es almacenada en un archivo, el cual contiene cada una de las coordenadas ordenadas, siendo la primera el punto inicial, posteriormente cada una de las demás coordenadas que representan a cada una de las células del Physarum, ordenadas por orden de aparición para que el robot tenga un orden el cual seguir para poder llegar a su destino.

Finalmente, la última coordenada corresponde a la coordenada del punto final, el cual es el destino al cual el robot deberá de llegar, finalizando así su recorrido.

5.1.3. Codificación e implementación de algoritmo en el robot en la primera iteración 2

La implementación en el robot no cambia respecto a cómo se estuvo planeando desde el inicio, debido a que las rutas son obtenidas de acuerdo con la simulación del Physarum, que, una vez finalizada esta, a partir de una serie de condiciones y convirtiendo el camino en coordenadas las cuales el robot pueda interpretar para realizar su recorrido de la mejor forma posible, se pueda llegar de un punto inicial a un punto final de forma correcta.

Como se planteo desde el inicio, el algoritmo debe generar su ruta con ayuda del Physarum, y a partir de esta, generar una serie de coordenadas las cuales el robot pueda seguir para moverse de un punto al otro. En la implementación, el archivo generado a partir de la ruta que se obtuvo con el Physarum, se puede ver en la Figura 41.



```
C:\Users\pkm\Documents\r& nutrient point: 16, 4  
initial point: 2, 16  
x: 2, y: 16  
x: 3, y: 15  
x: 4, y: 14  
x: 5, y: 13  
x: 6, y: 13  
x: 7, y: 12  
x: 8, y: 11  
x: 9, y: 10  
x: 10, y: 9  
x: 11, y: 8  
x: 12, y: 7  
x: 13, y: 7  
x: 14, y: 6  
Ruta obtenida
```

Figura 41: Archivo de rutas

Las coordenadas están dadas de acuerdo con el punto de partida, que está representado a partir del punto inicial en el simulador, hasta el destino, el cual es uno de los nutrientes en el simulador del Physarum y son de vital importancia para poder generar la ruta, ya que, con la ausencia de estos, no se puede generar una ruta. Las coordenadas están ordenadas de forma de aparición, lo que hace que haya una forma de seguirla correctamente y no se deba de perder, o seguir la ruta de manera errónea.

5.1.4. Ajustes del algoritmo basados en pruebas unitarias 1

Recopilando los datos obtenidos en algunas de las pruebas unitarias realizadas hasta el momento de realización de estas mismas, se aplicaron algunos cambios al programa, principalmente enfocados a mejorar cada una de sus funciones y que sea mucho más fácil en cuanto a la calidad del algoritmo.

Los cambios que han sido aplicados hasta el momento corresponden a los siguientes elementos que conforman el programa:

Implementación de la clase LoadMap: A pesar de que se pueda configurar un espacio en el lienzo por medio del dibujo, muchas veces al querer colocar un croquis o un mapa de determinado espacio, dibujar a mano con el ratón resulta poco conveniente, por lo que se implementó la clase LoadMap, la cual hace uso de la librería OpenCV, transformando una imagen en un dibujo en el lienzo del programa al inicializarse este último. Esto se puede ver en la Figura 42.

```

7   <> class LoadMap {
8       public:
9           void convertImageToMap(std::string);
10          void setDataToArray(int**, int, int);
11      private:
12          void grayscaleImage();
13          void setDataToRGBVector(int, int);
14      public:
15          float isImgProcessed = false;
16      private:
17          cv::Mat actualImage;
18          cv::Mat processedImage;
19          std::vector<std::vector<float>> rgbVector;
20          float threshold = 30;
21          int IMG_WIDTH = 500, IMG_HEIGHT = 500;
22      };

```

Figura 42: Implementación del LoadMap

```

8   <> class RandomRange {
9       public:
10          int getRandom(const int, const int);
11      private:
12          public:
13          private:
14      };
15

```

Figura 43: Clase RandomRange

Clase RandomRange: Al realizarse las primeras pruebas unitarias en el software, fue obtenido como resultado que es poco conveniente usar la función rand() debido a que tiende a ser mucho más lenta que otras implementaciones, además de que su aleatoriedad no es la mejor y además, no es multihilo, lo que impide que si en el futuro el algoritmo se quiere paralelizar, esto dificulte la implementación de numeros randoms que sean thread safe. Esto se puede ver en la Figura 43.

5.1.5. Ajustes del algoritmo basados en pruebas unitarias 2

En esta ocasión, no se realizaron cambios en el algoritmo, debido a que se pudo comprobar el correcto funcionamiento de este mismo y la cuestión real de funcionamiento o pruebas es el cambio de configuraciones en el estado inicial para la obtención de rutas, siendo evaluados distintos casos y configuraciones del estado inicial.

Debido a que hasta el momento el algoritmo cumple su función y las pruebas unitarias no revelaron algún resultado en el cual haya algún error o falla con alguna parte del algoritmo, en esta ocasión los ajustes al algoritmo no fueron realizados.

Es importante recalcar que el funcionamiento base del algoritmo es importante para la funcionalidad de la simulación en general, esto debido a que un pequeño cambio a alguna de sus partes puede alterar el resultado general y formar comportamientos o generar resultados no deseados y que no logran cumplir el objetivo general del software el cual es la generación de rutas a partir del organismo Physarum.

5.1.6. Diseño inicial de interfaz para control y monitorear

Para el diseño de la interfaz en la cual se puede controlar y a su vez, hacer uso del sistema de monitoreo del robot, es necesario conocer cada uno de los componentes que son necesarios mostrar y los que serán necesarios para realizar la funcionalidad correspondiente, los cuales corresponden a los siguientes:

Sistema de monitoreo: La interfaz debe mostrar la visión de la cámara que esta incorporada al robot, por lo que debe de haber un espacio dedicado a mostrar a esta misma.

Sistema LiDAR: En la interfaz también se debe mostrar cada uno de los puntos que son obtenidos a través de este sistema, los cuales representan los obstáculos que están siendo detectados por el robot, así como una representación de lo que se encuentra alrededor de éste.

Módulo de navegación: Debido a que el robot puede ser controlado a distancia, se necesita de la colocación de una serie de botones con los cuales el robot pueda realizar la navegación básica. Con los componentes mencionados anteriormente, se realiza el diseño de la interfaz inicial con la cual cada uno de estos es colocado de tal forma que se tenga acceso a él y se puedan realizar cada una de las tareas que fueron explicadas anteriormente.

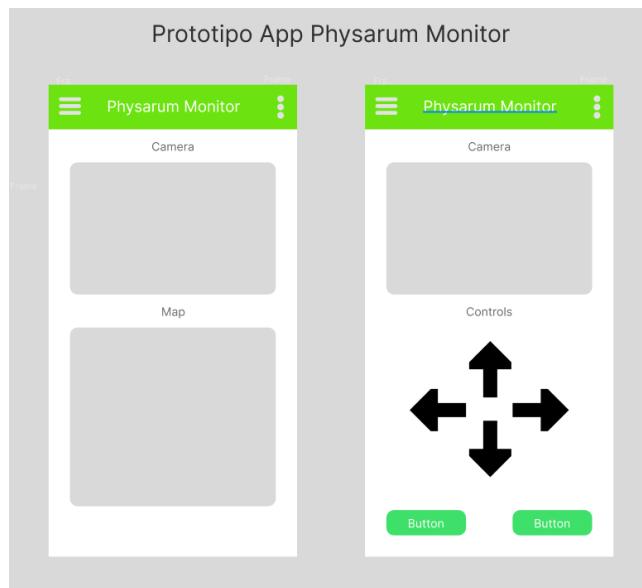


Figura 44: Prototipo App Physarum Monitor

Con la disposición mostrada en la Figura 44, podemos ver que se colocan los elementos que son necesarios para poder cumplir con cada una de las funciones que son necesarias para el control y monitoreo del robot. Se muestran dos configuraciones, una de las cuales contiene tanto la vista de la cámara, así como de la vista de la información que se nos es enviada por parte del robot, la cual es obtenida con el sensor LiDAR. La siguiente configuración contiene tanto la vista de la cámara del robot, como una serie de botones los cuales representan cada uno de los movimientos con los cuales es posible controlar al robot. Esta configuración en particular está diseñada para poder observar por medio de la cámara los movimientos que se están realizando a través de los botones y así, tener una mejor noción de lo que se está haciendo con el robot, así como para poder orientarse y poder tener un mejor control de este robot.

5.1.7. Ajustes de rutas basado en resultado de pruebas de aceptación 1

Con los resultados que fueron obtenidos a partir de la ejecución de las pruebas de aceptación, se toma en cuenta que hay algunas configuraciones o ajustes que se deben de hacer al algoritmo. Una de las primeras es generar una ruta 'mas limpia', es decir, una ruta en la cual existan mucho menos celdas las cuales impidan o dificulten generar una ruta un poco más optima o que causan

algunos impedimentos a la hora de generar el ordenamiento de coordenadas para ser enviadas al robot para la realización de su funcionalidad de llegar a su destino.

Una de las soluciones que se considera para poder obtener las rutas un poco más limpias y ajustarlas es limpiar un poco el camino o ruta generada por el Physarum una vez esta última siendo obtenida es eliminar las células que conforman al estado 8 y 5 respectivamente, las cuales no sean necesarias para el cálculo de la ruta o que no formen parte importante de la ruta que fue obtenida. Para esto se consideran dos casos importantes

- A) Las células alrededor de el punto inicial y el punto final.
- B) Las células que no conectan una célula con otra.

Una de las ideas para poder eliminar estas células las cuales dificultan la obtención correcta de las rutas es eliminar las células que están alrededor del estado inicial y el estado final, dejando únicamente las que tienen conexión con otras células las cuales pertenecen a la ruta general.

Esto se realizó de la siguiente forma creando un nuevo autómata con las siguientes reglas:

- Si el estado es 1 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 1.
- Si el estado es 2 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 2.
- Si el estado es 3, entonces es estado 7.
- Si el estado es 4 y alrededor no hay ningún estado 1, entonces es estado 0, sino, es estado 4.
- Si el estado es 5 y alrededor hay un punto inicial, entonces es estado 2, sino, si alrededor hay un nutriente, entonces es estado 4, sino es estado 1.
- Si es estado 6, entonces es estado 9.
- Si es estado 7, se queda en estado 7.
- Si el estado es 8 y alrededor hay un punto inicial, entonces es estado 2, sino, si alrededor hay un nutriente, entonces es estado 4, sino es estado 1.
- Si es estado 9, entonces se queda en estado 9.

Y queda plasmado con el siguiente Listing 3:

Listing 3: Ruta Ajuste 1

```
1      switch (tab[i][j]) {
2          case 1:
3              if (!aroundState1)
4                  cellsAux[i][j] = 0;
5              else
6                  cellsAux[i][j] = 1;
7          break;
```

```

8      case 2:
9          if (!aroundState1)
10             cellsAux[i][j] = 0;
11         else
12             cellsAux[i][j] = 2;
13     break;
14     case 3:
15         cellsAux[i][j] = 7;
16     break;
17     case 4:
18         if (!aroundState1)
19             cellsAux[i][j] = 0;
20         else
21             cellsAux[i][j] = 4;
22     break;
23     case 5:
24         cellsCounter++;
25         if (isInitialPointAround)
26             cellsAux[i][j] = 2;
27         else if (isNutrientAround)
28             cellsAux[i][j] = 4;
29         else
30             cellsAux[i][j] = 1;
31     break;
32     case 6:
33         cellsAux[i][j] = 9;
34     break;
35     case 7:
36         cellsAux[i][j] = 7;
37     break;
38     case 8:
39         cellsCounter++;
40         if (isInitialPointAround)
41             cellsAux[i][j] = 2;
42         else if (isNutrientAround)
43             cellsAux[i][j] = 4;
44         else
45             cellsAux[i][j] = 1;
46     break;
47     case 9:
48         cellsAux[i][j] = 9;
49     break;
50 }

```

Lo anterior se hace para poder obtener una ruta con una mejor claridad y que dificulte menos el ordenamiento y la manera en la cual se obtienen las coordenadas con las cuales se realiza el recorrido.

5.2. Robot Propuesto

Para construir el robot, se emplearán diversos materiales, cada uno con una función específica para asegurar la operatividad y eficiencia del dispositivo. A continuación, se detallan los materiales y sus descripciones.

El controlador para el motor paso a paso, Nema 23, será fundamental para manejar el movimiento del robot. Este componente incluye un controlador de motor a pasos que se utilizará en cuatro unidades para garantizar un control preciso de los motores. Los motores a pasos Nema

23 son conocidos por su precisión y confiabilidad, y en este caso, se utilizarán cuatro unidades, cada una con una placa frontal de 2.15 x 2.15 pulgadas (57 x 57 mm).

Para la estructura del robot, se utilizará una lámina de aluminio de calibre 14 (1.9 mm) con dimensiones de 20 cm x 40 cm, que proporcionará una base sólida y resistente. Además, se emplearán perfiles de aluminio 2040, específicamente de 20 x 40 mm y 500 mm de longitud, en dos unidades, para construir el marco del robot. También se utilizarán barras redondas sólidas de aluminio de 2 1/2" x 12", en cuatro unidades, para reforzar la estructura y proporcionar soporte adicional.

La movilidad del robot será posible gracias a las ruedas omnidireccionales de 6 pulgadas (152 mm) con rodamientos de silicona y cubos de aleación de aluminio, en cuatro unidades. Estas ruedas permitirán un movimiento fluido en múltiples direcciones. Además, se utilizarán diversos tornillos y tuercas, con un paquete de 60 unidades, para ensamblar todas las partes del robot de manera segura.

Para la energía, se utilizarán baterías de litio de 12V y 20000mAh, recargables, que proporcionarán la energía necesaria para la operación del robot. Se utilizarán dos de estas baterías. Un cargador de baterías de 14V y 20A, específico para baterías de litio de 12V, será empleado para mantener las baterías recargadas y operativas.

La electrónica del robot incluirá una Raspberry Pi 4 B, que actuará como el cerebro del dispositivo, gestionando las operaciones y los datos recibidos. Un sensor de distancia Detección y Rango de Luz (Light Detection and Ranging, LiDAR), modelo DTOF STL27L, permitirá al robot detectar obstáculos y medir distancias con precisión, utilizando un láser LiDAR 360° con bus UART y un rango de 895 - 915 NM (tipo 905).

Para la visión, se empleará una cámara de visión nocturna de luz infrarroja de 5MP, con un ángulo de visión de 130-220 grados, específica para la Raspberry Pi 4B. También se incluirá un convertidor auto Boost Buck de CD-CD, de 5A y rango de 5V-30V, que ayudará a gestionar las diferentes necesidades de voltaje de los componentes electrónicos del robot.

Finalmente, se utilizará una lámina de acrílico transparente de 6 mm, con dimensiones de 60 x 120 cm, para crear cubiertas protectoras y otras partes visibles del robot. Este material es ideal por su durabilidad y resistencia a impactos.

Estos componentes, cuidadosamente seleccionados, se ensamblarán para crear un robot funcional, robusto y versátil, capaz de realizar diversas tareas con eficiencia y precisión.

Ahora claro enseñamos lo que sería la primera versión del sistema de control del robot propuesto.

5.2.1. Desarrollo Inicial del Sistema del control del Robot

En esta subsección nos centramos en desarrollar el código inicial para poder controlar el robot, el cual se puede ver en el Listing 4.

Listing 4: Primera versión del código del sistema de control del robot

```
1      #include "CYdLidar.h"
2      #include <SFML/Graphics.hpp>
```

```

3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>
12 #include <thread>
13 #include <random>
14
15 using namespace std;
16 using namespace ydlidar;
17
18 // Pines y configuraci\'on para los motores
19 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
20     motores
21 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direcci\'on
22     para los motores
23 int frequency = 400; // Frecuencia inicial
24
25 std::atomic<bool> is_running(true);
26 std::atomic<bool> is_manual_mode(true);
27
28 void setMotorSpeed(int motor, int frequency) {
29     if (motor >= 0 && motor < 4) {
30         gpioSetPWMfrequency(PWM_PINS[motor], frequency);
31     }
32 }
33
34 void setMotorDirection(int motor, int direction) {
35     if (motor >= 0 && motor < 4) {
36         gpioWrite(DIR_PINS[motor], direction);
37     }
38 }
39
40 void stopMotors() {
41     for (int i = 0; i < 4; ++i) {
42         gpioPWM(PWM_PINS[i], 0);
43     }
44 }
45
46 void moveForward() {
47     for (int i = 0; i < 4; ++i) {
48         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
49             al 50%
50     }
51     setMotorDirection(0, 0); // Motor 1
52     setMotorDirection(2, 0); // Motor 3
53     setMotorDirection(1, 1); // Motor 2
54     setMotorDirection(3, 1); // Motor 4
55 }
56
57 void moveBackward() {
58     for (int i = 0; i < 4; ++i) {
59         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
60             al 50%
61     }
62     setMotorDirection(0, 1); // Motor 1

```



```

115         if (!is_manual_mode) {
116             LaserScan scan;
117             if (laser.doProcessSimple(scan)) {
118                 bool obstacle_detected = false;
119                 for (const auto &point : scan.points) {
120                     if (point.range < 0.35) { // Detecta un obst\'aculo a 35 cm
121                         obstacle_detected = true;
122                         break;
123                     }
124                 }
125             }
126             if (obstacle_detected) {
127                 // Si detecta un obst\'aculo, retrocede por 4 segundos
128                 moveBackward();
129                 std::this_thread::sleep_for(std::chrono::seconds(4));
130                 stopMotors();
131
132                 // Luego gira aleatoriamente a la izquierda o derecha
133                 int turn = dist(gen) % 2;
134                 if (turn == 0) {
135                     turnLeft();
136                 } else {
137                     turnRight();
138                 }
139                 std::this_thread::sleep_for(std::chrono::seconds(2));
140                 stopMotors();
141             } else {
142                 // Si no hay obst\'aculo, elige un movimiento aleatorio
143                 int move = dist(gen);
144                 switch (move) {
145                     case 0: moveBackward(); break;
146                     case 1: turnRight(); break;
147                     case 2: turnLeft(); break;
148                     case 3: moveForward(); break;
149                 }
150             }
151             std::this_thread::sleep_for(std::chrono::milliseconds(500));
152             stopMotors();
153         }
154         std::this_thread::sleep_for(std::chrono::milliseconds(100));
155     }
156 }
157 }
158
159 int main() {
160     // Inicializar pigpio
161     if (gpioInitialise() < 0) {
162         std::cerr << "Error: No se pudo inicializar pigpio." <<
163             std::endl;
164         return -1;
165     }

```

```

166     // Configurar pines de dirección como salida
167     for (int i = 0; i < 4; ++i) {
168         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
169         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
170         setMotorSpeed(i, frequency); // Inicializar PWM con
171             frecuencia inicial
172     }
173
174     // Asegúrate de establecer XDG_RUNTIME_DIR
175     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
176         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
177     }
178
179     // Ejecuta libcamera-vid en un proceso separado y captura la
180         // salida en YUV, sin previsualización
181     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
182         nopreview -o -", "r");
183     if (!pipe) {
184         std::cerr << "Error: No se pudo ejecutar libcamera-vid."
185             << std::endl;
186         return -1;
187     }
188
189     // Configura la ventana SFML
190     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
191         Visualization with LiDAR");
192     sf::Texture cameraTexture;
193     sf::Sprite cameraSprite;
194
195     // Buffer para leer los datos de video
196     const int width = 640;
197     const int height = 480;
198     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
199         el tamaño del buffer para YUV420
200
201     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
202         data());
203     cv::Mat rgbImage(height, width, CV_8UC3);
204
205     std::string port;
206     ydlidar::os_init();
207
208     // Obtener los puertos disponibles de LiDAR
209     std::map<std::string, std::string> ports = ydlidar::
210         lidarPortList();
211     if (ports.size() > 1) {
212         auto it = ports.begin();
213         std::advance(it, 1); // Selecciona el segundo puerto
214             disponible
215         port = it->second;
216     } else if (ports.size() == 1) {
217         port = ports.begin()->second;
218     } else {
219         std::cerr << "No se detectó ningún LiDAR. Verifica la
220             conexión." << std::endl;
221         return -1;
222     }
223
224     // Configuración del LiDAR
225     int baudrate = 115200;

```

```

216     std::cout << "Baudrate: " << baudrate << std::endl;
217
218     CYdLidar laser;
219     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size()
220                         ());
220     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
221                         int));
221
222     bool isSingleChannel = true;
223     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
224                         sizeof(bool));
224
225     float max_range = 8.0f;
226     float min_range = 0.1f;
227     float max_angle = 180.0f;
228     float min_angle = -180.0f;
229     float frequency = 8.0f;
230
231     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
232                         );
232     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
233                         );
233     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
234                         );
234     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
235                         );
235     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
236                         float));
236
237     // Inicializar LiDAR
238     if (!laser.initialize()) {
239         std::cerr << "Error al inicializar el LiDAR." << std::endl
240                 ;
241         return -1;
242     }
243
244     // Iniciar el escaneo
245     if (!laser.turnOn()) {
246         std::cerr << "Error al encender el LiDAR." << std::endl;
247         return -1;
248     }
249
250     // Thread para movimiento aleatorio
251     std::thread randomMoveThread(randomMovement, std::ref(laser));
252
253     while (window.isOpen()) {
254         sf::Event event;
255         while (window.pollEvent(event)) {
256             if (event.type == sf::Event::Closed)
257                 window.close();
258
259             if (event.type == sf::Event::KeyPressed) {
260                 switch (event.key.code) {
261                     case sf::Keyboard::W:
262                         is_manual_mode = true;
263                         moveForward();
264                         break;
265                     case sf::Keyboard::S:
266                         is_manual_mode = true;
267                         moveBackward();
268                 }
269             }
270         }
271     }
272 }
```

```

267         break;
268     case sf::Keyboard::A:
269         is_manual_mode = true;
270         turnLeft();
271         break;
272     case sf::Keyboard::D:
273         is_manual_mode = true;
274         turnRight();
275         break;
276     case sf::Keyboard::Add:
277         is_manual_mode = true;
278         increaseSpeed();
279         break;
280     case sf::Keyboard::Subtract:
281         is_manual_mode = true;
282         decreaseSpeed();
283         break;
284     case sf::Keyboard::Space:
285         is_manual_mode = true;
286         stopMotors();
287         break;
288     case sf::Keyboard::K:
289         is_manual_mode = false;
290         break;
291     case sf::Keyboard::M:
292         is_manual_mode = true;
293         break;
294     default:
295         break;
296     }
297   }
298 }

// Leer los datos del video desde la tubería
301 size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
302                         pipe);
303 if (bytesRead != buffer.size()) {
304     std::cerr << "Error: No se pudo leer suficientes datos
305     de video." << std::endl;
306     continue;
307 }
308 // Convertir YUV420 a RGB
309 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
310 // Convertir a RGBA añadiendo un canal alfa
311 cv::Mat frame_rgba;
312 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
313 // Actualizar la textura de la cámara con los datos del
314 // frame
315 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows
316 )) {
317     std::cerr << "Error: No se pudo crear la textura." <<
318     std::endl;
319     continue;
320 }
321 cameraTexture.update(frame_rgba.ptr());
322 cameraSprite.setTexture(cameraTexture);

```

```

322     cameraSprite.setScale(
323         window.getSize().x / static_cast<float>(cameraTexture.
324             getSize().x),
325         window.getSize().y / static_cast<float>(cameraTexture.
326             getSize().y)
327     );
328
329     window.clear();
330     window.draw(cameraSprite);
331
332     // Crear un minimapa para el LiDAR
333     sf::RectangleShape minimap(sf::Vector2f(200, 200));
334     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // // Fondo semitransparente
335     minimap.setPosition(10, 10); // Esquina superior izquierda
336
337     window.draw(minimap);
338
339     // Dibujar el centro del LiDAR (color azul)
340     sf::CircleShape lidarCenter(5); // Radio del círculo del
341     lidarCenter.setFillColor(sf::Color::Blue);
342     lidarCenter.setPosition(105, 105); // Posición del
343     centro en el minimapa
344
345     window.draw(lidarCenter);
346
347     // Dibujar la línea hacia el norte
348     sf::Vertex line[] =
349     {
350         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
351         sf::Vertex(sf::Vector2f(110, 60), sf::Color::Black) // Línea hacia arriba (norte)
352     };
353
354     window.draw(line, 2, sf::Lines);
355
356     LaserScan scan;
357     if (laser.doProcessSimple(scan)) {
358         for (const auto& point : scan.points) {
359             // Convertir coordenadas polares a cartesianas
360             float x = point.range * cos(point.angle);
361             float y = point.range * sin(point.angle);
362
363             // Ajustar los puntos al minimapa
364             float scale = 25.0f;
365             float adjustedX = 110 + x * scale;
366             float adjustedY = 110 - y * scale; // Invertir Y
367             // para coordinar con la pantalla
368
369             // Dibujar los puntos en el minimapa, excluyendo
370             // el centro (0,0)
371             if (point.range > 0.05) {
372                 sf::CircleShape lidarPoint(2);
373                 lidarPoint.setPosition(adjustedX, adjustedY);
374                 lidarPoint.setFillColor(getPointColor(point.
375                     range, max_range));
376
377                 window.draw(lidarPoint);
378             }
379         }
380     }

```

```

373         }
374     } else {
375         std::cerr << "No se pudieron obtener los datos del
376             LiDAR." << std::endl;
377     }
378     window.display();
379 }
380
381 // Detener el escaneo del LiDAR
382 laser.turnOff();
383 laser.disconnecting();
384
385 // Cierra la tubería, detiene los motores y apaga el robot
386 pclose(pipe);
387 stopMotors();
388 gpioTerminate();
389
390 is_running = false;
391 randomMoveThread.join();
392
393 return 0;
394 }
```

En el Listing 4 se inicializan los motores y se establecen las funciones para controlar el movimiento del robot. Además, se configura el LiDAR y se inicia el escaneo. El robot se mueve aleatoriamente y evita obstáculos detectados por el LiDAR. La cámara muestra la vista del robot y el minimapa muestra los puntos detectados por el LiDAR. El usuario puede controlar manualmente el robot con las teclas W, A, S y D para moverse hacia adelante, izquierda, atrás y derecha, respectivamente. Las teclas V y B aumentan y disminuyen la velocidad del robot, respectivamente. La tecla Espacio detiene el robot. La tecla K activa el modo automático y la tecla M activa el modo manual. El robot se mueve aleatoriamente y evita obstáculos detectados por el LiDAR. El código se ejecuta en un bucle hasta que se cierra la ventana de la cámara. Al final, se detiene el escaneo del LiDAR y se apaga el robot.

5.2.2. Ajustes de código en función de pruebas unitarias y de aceptación 1

El código del robot funcionó correctamente en las pruebas unitarias y de aceptación, con la excepción del modo aleatorio, que presentó problemas de movimiento. El robot se movía de manera ineficiente en el modo aleatorio, debido a la forma en que se le ordenaba moverse y que en 0,0 detectaba obstáculos donde no los había, en otras palabras daba falsos positivos. Por lo tanto, se realizaron ajustes en el código para mejorar su desempeño y eficiencia, los cuales se pueden ver en el Listing 5.

Listing 5: Primer ajuste de código

```

1 #include "CYdLidar.h"
2 #include <SFML/Graphics.hpp>
3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>
```

```

12 #include <thread>
13 #include <random>
14 #include <sys/socket.h>
15 #include <sys/un.h>
16 #include <unistd.h>
17 #include <cstring>
18 #include <pthread.h>
19 #include <vector>
20 #include <mutex>
21
22 using namespace std;
23 using namespace ydlidar;
24
25 // Pines y configuración para los motores
26 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
27     motores
28 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de dirección para
29     los motores
30 int frequency = 400; // Frecuencia inicial
31
32 std::atomic<bool> is_running(true);
33 std::atomic<bool> is_manual_mode(true);
34
35 void setMotorSpeed(int motor, int frequency) {
36     if (motor >= 0 && motor < 4) {
37         gpioSetPWMfrequency(PWM_PINS[motor], frequency);
38     }
39 }
40
41 void setMotorDirection(int motor, int direction) {
42     if (motor >= 0 && motor < 4) {
43         gpioWrite(DIR_PINS[motor], direction);
44     }
45 }
46
47 void stopMotors() {
48     for (int i = 0; i < 4; ++i) {
49         gpioPWM(PWM_PINS[i], 0);
50     }
51 }
52
53 void moveForward() {
54     for (int i = 0; i < 4; ++i) {
55         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
56             50%
57     }
58     setMotorDirection(0, 0); // Motor 1
59     setMotorDirection(2, 0); // Motor 3
60     setMotorDirection(1, 1); // Motor 2
61     setMotorDirection(3, 1); // Motor 4
62 }
63
64 void moveBackward() {
65     for (int i = 0; i < 4; ++i) {
66         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
67             50%
68     }
69     setMotorDirection(0, 1); // Motor 1
70     setMotorDirection(2, 1); // Motor 3
71     setMotorDirection(1, 0); // Motor 2

```

```

68         setMotorDirection(3, 0); // Motor 4
69     }
70
71     void turnLeft() {
72         for (int i = 0; i < 4; ++i) {
73             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
74             50%
75         }
76         setMotorDirection(0, 1); // Motor 1
77         setMotorDirection(2, 1); // Motor 3
78         setMotorDirection(1, 1); // Motor 2
79         setMotorDirection(3, 1); // Motor 4
80     }
81
82     void turnRight() {
83         for (int i = 0; i < 4; ++i) {
84             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al
85             50%
86         }
87         setMotorDirection(0, 0); // Motor 1
88         setMotorDirection(2, 0); // Motor 3
89         setMotorDirection(1, 0); // Motor 2
90         setMotorDirection(3, 0); // Motor 4
91     }
92
93     void increaseSpeed() {
94         if (frequency < 2000) {
95             frequency += 100;
96             if (frequency > 2000) frequency = 2000;
97             for (int i = 0; i < 4; ++i) {
98                 setMotorSpeed(i, frequency);
99             }
100        }
101    }
102
103    void decreaseSpeed() {
104        if (frequency > 400) {
105            frequency -= 100;
106            if (frequency < 400) frequency = 400;
107            for (int i = 0; i < 4; ++i) {
108                setMotorSpeed(i, frequency);
109            }
110        }
111    }
112
113    sf::Color getPointColor(float distance, float maxRange) {
114        float ratio = distance / maxRange;
115        return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de
116        rojo a verde
117    }
118
119    int contadorObstaculoTotal = 0;
120    int obstaculoHola = 0;
121    int obsRelativo = 0;
122    void randomMovement(CYdLidar &laser) {
123        std::random_device rd;
124        std::mt19937 gen(rd());
125        std::discrete_distribution<> dist({1, 0, 10, 70}); // Distribucion
126        para la probabilidad de movimiento

```

```

124
125     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
126         grados en radaianes
127     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
128         grados en radianes
129     const float DETECTION_RADIUS = 0.25f; // 35 cm
130
131     std::vector<float> previousScanPoints;
132
133     while (is_running) {
134         if (!is_manual_mode) {
135             LaserScan scan;
136             if (laser.doProcessSimple(scan)) {
137                 bool obstacle_detected = false;
138                 std::vector<float> currentScanPoints;
139
140                 for (const auto &point : scan.points) {
141                     // Convertir el ngulo del punto al ngulo relativo
142                     // al "sur" del robot
143                     float adjusted_angle = point.angle + M_PI;
144                     //std::cout << "Er " << point.range << std::endl;
145                     // Verificar si el punto est dentro del rango
146                     // frontal de 30
147                     if (point.range > 0 && point.range < 0.40 && abs(
148                         point.angle) < adjusted_angle) {
149                         obstacle_detected = true;
150                         currentScanPoints.push_back(point.range);
151                         std::cout << "Obstacle distance: " << (float)
152                             point.range << " Y en el angulo " << point
153                             .angle << std::endl;
154                         break;
155                     }
156                 }
157
158                 if (obstacle_detected) {
159                     // Si detecta un obstculo, retrocede por 4
160                     // segundos
161                     contadorObstaculoTotal++;
162                     std::cout << "Obs: " << contadorObstaculoTotal <<
163                         std::endl;
164                     //obsRelativo++;
165                     stopMotors();
166                     //Luego gira aleatoriamente a la izquierda o
167                     //derecha
168                     int turn = dist(gen) % 2;
169                     if (turn == 0) {
170                         turnLeft();
171                         std::this_thread::sleep_for(std::chrono::
172                             seconds(5));
173                     } else {
174                         turnRight();
175                         std::this_thread::sleep_for(std::chrono::
176                             seconds(5));
177                     }
178                     std::this_thread::sleep_for(std::chrono::seconds
179                         (2));
180                     stopMotors();
181
182                     if (!previousScanPoints.empty() &&
183                         previousScanPoints.size() == currentScanPoints

```

```

        .size()){
170            bool mismoObs = true;
171            for(size_t i = 0; i < currentScanPoints.
172                size(); ++i){
173                    if(fabs(currentScanPoints[i] -
174                        previousScanPoints[i]) > 0.05){
175                            mismoObs = false;
176                            break;
177                        }
178                    if(mismoObs){
179                        obsRelativo++;
180                    }else{
181                        obsRelativo = 0;
182                    }
183                }
184
185                previousScanPoints = currentScanPoints;
186
186                if((obsRelativo > 3)){
187                    //obsRelativo = 0;
188                    moveBackward();
189                    std::this_thread::sleep_for(std::chrono::
190                        seconds(3));
191                    stopMotors();
192                    int turn = dist(gen) % 2;
193                    if (turn == 0) {
194                        turnLeft();
195                        std::this_thread::sleep_for(std::chrono::
196                            seconds(5));
197                    } else {
198                        turnRight();
199                        std::this_thread::sleep_for(std::chrono::
200                            seconds(5));
201                    }
202                } else {
203                    moveForward();
204                    // Si no hay obstculo, elige un movimiento
205                    // aleatorio
206                    //
207                    //int move = dist(gen);
208                    //switch (move) {
209                        // case 0: moveBackward(); break;
210                        // case 1: turnRight(); break;
211                        // case 2: turnLeft(); break;
212                        // case 3: moveForward(); break;
213                    //}
214                }
215
216                std::this_thread::sleep_for(std::chrono::milliseconds(500)
217                    );
218                //stopMotors();
219            std::this_thread::sleep_for(std::chrono::milliseconds(100));
220        }

```

```

221     }
222
223     int main() {
224         // Inicializar pigpio
225         if (gpioInitialise() < 0) {
226             std::cerr << "Error: No se pudo inicializar pigpio." << std::
227                 endl;
228             return -1;
229         }
230
231         // Configurar pines de direccin como salida
232         for (int i = 0; i < 4; ++i) {
233             gpioSetMode(DIR_PINS[i], PI_OUTPUT);
234             gpioSetMode(PWM_PINS[i], PI_OUTPUT);
235             setMotorSpeed(i, frequency); // Inicializar PWM con
236                 frecuencia inicial
237         }
238
239         // Asegrate de establecer XDG_RUNTIME_DIR
240         if (getenv("XDG_RUNTIME_DIR") == nullptr) {
241             setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(_id -u)", 1);
242         }
243
244         // Ejecuta libcamera-vid en un proceso separado y captura la
245             salida en YUV, sin previsualizacn
246         FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview
247             -o -", "r");
248         if (!pipe) {
249             std::cerr << "Error: No se pudo ejecutar libcamera-vid." <<
250                 std::endl;
251             return -1;
252         }
253
254         // Configura la ventana SFML
255         sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
256             Visualization with LiDAR");
257         sf::Texture cameraTexture;
258         sf::Sprite cameraSprite;
259
260         // Buffer para leer los datos de video
261         const int width = 640;
262         const int height = 480;
263         std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
264             tamao del buffer para YUV420
265
266         cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data
267             ());
268         cv::Mat rgbImage(height, width, CV_8UC3);
269
270         std::string port;
271         ydlidar::os_init();
272
273         // Obtener los puertos disponibles de LiDAR
274         std::map<std::string, std::string> ports = ydlidar::lidarPortList
275             ();
276         if (ports.size() > 1) {
277             auto it = ports.begin();
278             std::advance(it, 1); // Selecciona el segundo puerto
279                 disponible
280             port = it->second;

```

```

271     } else if (ports.size() == 1) {
272         port = ports.begin()->second;
273     } else {
274         std::cerr << "No se detect ningn LiDAR. Verifica la conexin."
275             << std::endl;
276         return -1;
277     }
278
279     // Configuracin del LiDAR
280     int baudrate = 115200;
281     std::cout << "Baudrate: " << baudrate << std::endl;
282
283     CYdLidar laser;
284     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
285     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int))
286         ;
287
288     bool isSingleChannel = true;
289     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof
290         (bool));
291
292     float max_range = 8.0f;
293     float min_range = 0.1f;
294     float max_angle = 180.0f;
295     float min_angle = -180.0f;
296     float frequency = 8.0f;
297
298     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
299     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
300     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
301     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
302     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float)
303         );
304
305     // Inicializar LiDAR
306     if (!laser.initialize()) {
307         std::cerr << "Error al inicializar el LiDAR." << std::endl;
308         return -1;
309     }
310
311     // Iniciar el escaneo
312     if (!laser.turnOn()) {
313         std::cerr << "Error al encender el LiDAR." << std::endl;
314         return -1;
315     }
316
317     // Thread para movimiento aleatorio
318     std::thread randomMoveThread(randomMovement, std::ref(laser));
319
320     while (window.isOpen()) {
321         sf::Event event;
322         while (window.pollEvent(event)) {
323             if (event.type == sf::Event::Closed)
324                 window.close();
325
326             if (event.type == sf::Event::KeyPressed) {
327                 switch (event.key.code) {
328                     case sf::Keyboard::W:
329                         is_manual_mode = true;
330                         moveForward();

```

```

327         break;
328     case sf::Keyboard::S:
329         is_manual_mode = true;
330         moveBackward();
331         break;
332     case sf::Keyboard::A:
333         is_manual_mode = true;
334         turnLeft();
335         break;
336     case sf::Keyboard::D:
337         is_manual_mode = true;
338         turnRight();
339         break;
340     case sf::Keyboard::V:
341         is_manual_mode = true;
342         increaseSpeed();
343         break;
344     case sf::Keyboard::B:
345         is_manual_mode = true;
346         decreaseSpeed();
347         break;
348     case sf::Keyboard::Space:
349         is_manual_mode = true;
350         stopMotors();
351         break;
352     case sf::Keyboard::K:
353         is_manual_mode = false;
354         break;
355     case sf::Keyboard::M:
356         is_manual_mode = true;
357         break;
358     default:
359         break;
360     }
361   }
362 }
363
364 // Leer los datos del video desde la tubera
365 size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe
366 );
367 if (bytesRead != buffer.size()) {
368   std::cerr << "Error: No se pudo leer suficientes datos de
369   video." << std::endl;
370   continue;
371 }
372
373 // Convertir YUV420 a RGB
374 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
375
376 // Convertir a RGBA añadiendo un canal alfa
377 cv::Mat frame_rgba;
378 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
379
380 // Actualizar la textura de la cámara con los datos del frame
381 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
382   std::cerr << "Error: No se pudo crear la textura." << std
383   ::endl;
384   continue;
385 }
386 cameraTexture.update(frame_rgba.ptr());

```

```

384
385     cameraSprite.setTexture(cameraTexture);
386     cameraSprite.setScale(
387         window.getSize().x / static_cast<float>(cameraTexture.
388             getSize().x),
389         window.getSize().y / static_cast<float>(cameraTexture.
390             getSize().y)
391     );
392
393     window.clear();
394     window.draw(cameraSprite);
395
396     // Crear un minimapa para el LiDAR
397     sf::RectangleShape minimap(sf::Vector2f(200, 200));
398     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
399     minimap.setTransparency(150); // Semicolor transparente
400     minimap.setPosition(10, 10); // Esquina superior izquierda
401
402     window.draw(minimap);
403
404     // Dibujar el centro del LiDAR (color azul)
405     sf::CircleShape lidarCenter(5); // Radio del círculo del LiDAR
406     lidarCenter.setFillColor(sf::Color::Blue);
407     lidarCenter.setPosition(105, 105); // Posición del centro en el
408     // minimapa
409
410     window.draw(lidarCenter);
411
412     // Dibujar la línea hacia el norte
413     sf::Vertex line[] =
414     {
415         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
416         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) // Línea hacia arriba (norte)
417     };
418
419     window.draw(line, 2, sf::Lines);
420
421     LaserScan scan;
422     if (laser.doProcessSimple(scan)) {
423         for (const auto& point : scan.points) {
424             // Convertir coordenadas polares a cartesianas
425             float x = point.range * cos(point.angle);
426             float y = point.range * sin(point.angle);
427
428             // Ajustar los puntos al minimapa
429             float scale = 25.0f;
430             float adjustedX = 110 + x * scale;
431             float adjustedY = 110 - y * scale; // Invertir Y para
432             // coordinar con la pantalla
433
434             // Dibujar los puntos en el minimapa, excluyendo el
435             // centro (0,0)
436             if (point.range > 0.05) {
437                 sf::CircleShape lidarPoint(2);
438                 lidarPoint.setPosition(adjustedX, adjustedY);
439                 lidarPoint.setFillColor(getPointColor(point.range,
440                     max_range));
441
442                 window.draw(lidarPoint);
443             }
444         }
445     }

```

```

436             }
437             //std::cout << "An " << point.range << std :: endl;
438         }
439     } else {
440         std::cerr << "No se pudieron obtener los datos del LiDAR."
441             << std::endl;
442     }
443     window.display();
444 }
445
446 // Detener el escaneo del LiDAR
447 laser.turnOff();
448 laser.disconnecting();
449
450 // Cierra la tubera, detiene los motores y apaga el robot
451 pclose(pipe);
452 stopMotors();
453 gpioTerminate();
454
455 is_running = false;
456 randomMoveThread.join();
457
458 return 0;
459 }
```

5.2.3. Ajustes de código en función de pruebas unitarias y de aceptación 2

En este caso se agregaron modificaciones para el mejor funcionamiento del detectar obstáculos sobre todo los ángulos de giro y la distancia de detección de los obstáculos. Estas modificaciones se realizaron en el código principal del robot, el cual se muestra en el Listing 6.

Listing 6: Segundo ajuste de código

```

1      #include "CYdLidar.h"
2      #include <SFML/Graphics.hpp>
3      #include <opencv2/opencv.hpp>
4      #include <iostream>
5      #include <cstdlib>
6      #include <cstdio>
7      #include <pigpio.h>
8      #include <string>
9      #include <map>
10     #include <vector>
11     #include <atomic>
12     #include <thread>
13     #include <random>
14     #include <sys/socket.h>
15     #include <sys/un.h>
16     #include <unistd.h>
17     #include <cstring>
18     #include <pthread.h>
19     #include <vector>
20     #include <mutex>
21
22     using namespace std;
23     using namespace ydlidar;
24
25 // Pines y configuración para los motores
```

```

26     const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
27     motores
28     const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de direccin
29     para los motores
30     int frequency = 400; // Frecuencia inicial
31
32     std::atomic<bool> is_running(true);
33     std::atomic<bool> is_manual_mode(true);
34
35     void setMotorSpeed(int motor, int frequency) {
36         if (motor >= 0 && motor < 4) {
37             gpioSetPWMrrequency(PWM_PINS[motor], frequency);
38         }
39     }
40
41     void setMotorDirection(int motor, int direction) {
42         if (motor >= 0 && motor < 4) {
43             gpioWrite(DIR_PINS[motor], direction);
44         }
45     }
46
47     void stopMotors() {
48         for (int i = 0; i < 4; ++i) {
49             gpioPWM(PWM_PINS[i], 0);
50         }
51     }
52
53     void moveForward() {
54         for (int i = 0; i < 4; ++i) {
55             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
56             al 50%
57         }
58         setMotorDirection(0, 0); // Motor 1
59         setMotorDirection(2, 0); // Motor 3
60         setMotorDirection(1, 1); // Motor 2
61         setMotorDirection(3, 1); // Motor 4
62     }
63
64     void moveBackward() {
65         for (int i = 0; i < 4; ++i) {
66             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
67             al 50%
68         }
69         setMotorDirection(0, 1); // Motor 1
70         setMotorDirection(2, 1); // Motor 3
71         setMotorDirection(1, 0); // Motor 2
72         setMotorDirection(3, 0); // Motor 4
73     }
74
75     void turnLeft() {
76         for (int i = 0; i < 4; ++i) {
77             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
78             al 50%
79         }
80         setMotorDirection(0, 1); // Motor 1
81         setMotorDirection(2, 1); // Motor 3
82         setMotorDirection(1, 1); // Motor 2
83         setMotorDirection(3, 1); // Motor 4
84     }

```

```

81     void turnRight() {
82         for (int i = 0; i < 4; ++i) {
83             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
84             al 50%
85         }
86         setMotorDirection(0, 0); // Motor 1
87         setMotorDirection(2, 0); // Motor 3
88         setMotorDirection(1, 0); // Motor 2
89         setMotorDirection(3, 0); // Motor 4
90     }
91
92     void increaseSpeed() {
93         if (frequency < 2000) {
94             frequency += 100;
95             if (frequency > 2000) frequency = 2000;
96             for (int i = 0; i < 4; ++i) {
97                 setMotorSpeed(i, frequency);
98             }
99         }
100    }
101
102    void decreaseSpeed() {
103        if (frequency > 400) {
104            frequency -= 100;
105            if (frequency < 400) frequency = 400;
106            for (int i = 0; i < 4; ++i) {
107                setMotorSpeed(i, frequency);
108            }
109        }
110    }
111
112    sf::Color getPointColor(float distance, float maxRange) {
113        float ratio = distance / maxRange;
114        return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
115        de rojo a verde
116    }
117
118    int contadorObstaculoTotal = 0;
119    int obstaculoHola = 0;
120    int obsRelativo = 0;
121    void randomMovement(CYdLidar &laser) {
122        std::random_device rd;
123        std::mt19937 gen(rd());
124        std::discrete_distribution<> dist({1, 0, 10, 70}); // Distribución para la probabilidad de movimiento
125        const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
126        grados en radianes
127        const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
128        grados en radianes
129        const float DETECTION_RADIUS = 0.25f; // 35 cm
130
131        std::vector<float> previousScanPoints;
132
133        while (is_running) {
134            if (!is_manual_mode) {
135                LaserScan scan;
136                if (laser.doProcessSimple(scan)) {
137                    bool obstacle_detected = false;

```

```

136         std::vector<float> currentScanPoints;
137
138     for (const auto &point : scan.points) {
139         // Convertir el ngulo del punto al ngulo
140         // relativo al "sur" del robot
141         float adjusted_angle = point.angle + M_PI;
142         //std::cout << "Er " << point.range << std::endl;
143         // Verificar si el punto est dentro del rango
144         // frontal de 30
145         if (point.range > 0 && point.range < 0.40 &&
146             abs(point.angle) < adjusted_angle) {
147             obstacle_detected = true;
148             currentScanPoints.push_back(point.range);
149             std::cout << "Obstacle distance: " << (
150                 float)point.range << " Y en el angulo
151                 "<<point.angle << std::endl;
152             break;
153         }
154     }
155
156     if (obstacle_detected) {
157         // Si detecta un obstculo, retrocede por 4
158         // segundos
159         contadorObstaculoTotal++;
160         std::cout << "Obs: " << contadorObstaculoTotal
161             << std::endl;
162         //obsRelativo++;
163         stopMotors();
164         //Luego gira aleatoriamente a la izquierda o
165         //derecha
166         int turn = dist(gen) % 2;
167         if (turn == 0) {
168             turnLeft();
169             std::this_thread::sleep_for(std::chrono::
170                 seconds(5));
171         } else {
172             turnRight();
173             std::this_thread::sleep_for(std::chrono::
174                 seconds(5));
175         }
176         std::this_thread::sleep_for(std::chrono::
177             seconds(2));
178         stopMotors();
179
180         if (!previousScanPoints.empty() &&
181             previousScanPoints.size() ==
182             currentScanPoints.size()){
183             bool mismoObs = true;
184             for(size_t i = 0; i <
185                 currentScanPoints.size(); ++i){
186                 if(fabs(currentScanPoints[i] -
187                     previousScanPoints[i]) > 0.05)
188                 {
189                     mismoObs = false;
190                     break;
191                 }
192             }
193             if(mismoObs){
194                 obsRelativo++;
195             }
196         }
197     }
198 }
```

```

179                     } else{
180                         obsRelativo = 0;
181                     }
182                 }
183
184             previousScanPoints = currentScanPoints;
185
186             if((obsRelativo > 3)){
187                 //obsRelativo = 0;
188                 moveBackward();
189                 std::this_thread::sleep_for(std::chrono::
190                     seconds(3));
191                 stopMotors();
192                 int turn = dist(gen) % 2;
193                 if (turn == 0) {
194                     turnLeft();
195                     std::this_thread::sleep_for(std::
196                         chrono::seconds(5));
197                 } else {
198                     turnRight();
199                     std::this_thread::sleep_for(std::
200                         chrono::seconds(5));
201                 }
202             }
203             std::this_thread::sleep_for(std::
204                 chrono::seconds(2));
205             stopMotors();
206         }
207     }
208     moveForward();
209     // Si no hay obstculo, elige un movimiento
210     // aleatorio
211     //int move = dist(gen);
212     //switch (move) {
213     //    case 0: moveBackward(); break;
214     //    case 1: turnRight(); break;
215     //    case 2: turnLeft(); break;
216     //    case 3: moveForward(); break;
217     //}
218 }
219 std::this_thread::sleep_for(std::chrono::milliseconds
220     (500));
221 //stopMotors();
222 }
223 std::this_thread::sleep_for(std::chrono::milliseconds(100)
224 );
225
226 int main() {
227     // Inicializar pigpio
228     if (gpioInitialise() < 0) {
229         std::cerr << "Error: No se pudo inicializar pigpio." <<
230             std::endl;
231         return -1;
232     }
233
234     // Configurar pines de direccin como salida

```

```

231     for (int i = 0; i < 4; ++i) {
232         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
233         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
234         setMotorSpeed(i, frequency); // Inicializar PWM con
235             frecuencia inicial
236     }
237
238     // Asegurate de establecer XDG_RUNTIME_DIR
239     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
240         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(_id -u)", 1);
241     }
242
243     // Ejecuta libcamera-vid en un proceso separado y captura la
244     // salida en YUV, sin previsualizacn
245     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
246         nopreview -o -", "r");
247     if (!pipe) {
248         std::cerr << "Error: No se pudo ejecutar libcamera-vid."
249             << std::endl;
250         return -1;
251     }
252
253     // Configura la ventana SFML
254     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
255         Visualization with LiDAR");
256     sf::Texture cameraTexture;
257     sf::Sprite cameraSprite;
258
259     // Buffer para leer los datos de video
260     const int width = 640;
261     const int height = 480;
262     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
263         el tamao del buffer para YUV420
264
265     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
266         data());
267     cv::Mat rgbImage(height, width, CV_8UC3);
268
269     std::string port;
270     ydlidar::os_init();
271
272     // Obtener los puertos disponibles de LiDAR
273     std::map<std::string, std::string> ports = ydlidar::
274         lidarPortList();
275     if (ports.size() > 1) {
276         auto it = ports.begin();
277         std::advance(it, 1); // Selecciona el segundo puerto
278             disponible
279         port = it->second;
280     } else if (ports.size() == 1) {
281         port = ports.begin()->second;
282     } else {
283         std::cerr << "No se detect ningn LiDAR. Verifica la
284             conexin." << std::endl;
285         return -1;
286     }
287
288     // Configuracin del LiDAR
289     int baudrate = 115200;
290     std::cout << "Baudrate: " << baudrate << std::endl;

```

```

281
282     CYdLidar laser;
283     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
284         ());
285     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
286         int));
287
288     bool isSingleChannel = true;
289     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
290         sizeof(bool));
291
292     float max_range = 8.0f;
293     float min_range = 0.1f;
294     float max_angle = 180.0f;
295     float min_angle = -180.0f;
296     float frequency = 8.0f;
297
298     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
299         );
300     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
301         );
302     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
303         );
304     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
305         );
306     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
307         float));
308
309     // Inicializar LiDAR
310     if (!laser.initialize()) {
311         std::cerr << "Error al inicializar el LiDAR." << std::endl
312             ;
313         return -1;
314     }
315
316     // Iniciar el escaneo
317     if (!laser.turnOn()) {
318         std::cerr << "Error al encender el LiDAR." << std::endl;
319         return -1;
320     }
321
322     // Thread para movimiento aleatorio
323     std::thread randomMoveThread(randomMovement, std::ref(laser));
324
325     while (window.isOpen()) {
326         sf::Event event;
327         while (window.pollEvent(event)) {
328             if (event.type == sf::Event::Closed)
329                 window.close();
330
331             if (event.type == sf::Event::KeyPressed) {
332                 switch (event.key.code) {
333                     case sf::Keyboard::W:
334                         is_manual_mode = true;
335                         moveForward();
336                         break;
337                     case sf::Keyboard::S:
338                         is_manual_mode = true;
339                         moveBackward();
340                         break;

```

```

332             case sf::Keyboard::A:
333                 is_manual_mode = true;
334                 turnLeft();
335                 break;
336             case sf::Keyboard::D:
337                 is_manual_mode = true;
338                 turnRight();
339                 break;
340             case sf::Keyboard::V:
341                 is_manual_mode = true;
342                 increaseSpeed();
343                 break;
344             case sf::Keyboard::B:
345                 is_manual_mode = true;
346                 decreaseSpeed();
347                 break;
348             case sf::Keyboard::Space:
349                 is_manual_mode = true;
350                 stopMotors();
351                 break;
352             case sf::Keyboard::K:
353                 is_manual_mode = false;
354                 break;
355             case sf::Keyboard::M:
356                 is_manual_mode = true;
357                 break;
358         default:
359             break;
360     }
361 }
362 }

363 // Leer los datos del video desde la tubera
364 size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
365                         pipe);
366 if (bytesRead != buffer.size()) {
367     std::cerr << "Error: No se pudo leer suficientes datos
368 de video." << std::endl;
369     continue;
370 }
371 // Convertir YUV420 a RGB
372 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
373 // Convertir a RGBA añadiendo un canal alfa
374 cv::Mat frame_rgba;
375 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
376
377 // Actualizar la textura de la cámara con los datos del
378 // frame
379 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows))
380 {
381     std::cerr << "Error: No se pudo crear la textura." <<
382     std::endl;
383     continue;
384 }
385 cameraTexture.update(frame_rgba.ptr());
386
387 cameraSprite.setTexture(cameraTexture);
388 cameraSprite.setScale(

```

```

387         window.getSize().x / static_cast<float>(cameraTexture.
388             getSize().x),
389         window.getSize().y / static_cast<float>(cameraTexture.
390             getSize().y)
391     );
392
393     window.clear();
394     window.draw(cameraSprite);
395
396     // Crear un minimapa para el LiDAR
397     sf::RectangleShape minimap(sf::Vector2f(200, 200));
398     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo semitransparente
399     minimap.setPosition(10, 10); // Esquina superior izquierda
400
401     window.draw(minimap);
402
403     // Dibujar el centro del LiDAR (color azul)
404     sf::CircleShape lidarCenter(5); // Radio del círculo del
405     lidarCenter.setFillColor(sf::Color::Blue);
406     lidarCenter.setPosition(105, 105); // Posición del centro
407     en el minimapa
408
409     window.draw(lidarCenter);
410
411     // Dibujar la línea hacia el norte
412     sf::Vertex line[] =
413     {
414         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
415         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black)
416             // Línea hacia arriba (norte)
417     };
418
419     window.draw(line, 2, sf::Lines);
420
421     LaserScan scan;
422     if (laser.doProcessSimple(scan)) {
423         for (const auto& point : scan.points) {
424             // Convertir coordenadas polares a cartesianas
425             float x = point.range * cos(point.angle);
426             float y = point.range * sin(point.angle);
427
428             // Ajustar los puntos al minimapa
429             float scale = 25.0f;
430             float adjustedX = 110 + x * scale;
431             float adjustedY = 110 - y * scale; // Invertir Y
432             para coordinar con la pantalla
433
434             // Dibujar los puntos en el minimapa, excluyendo
435             el centro (0,0)
436             if (point.range > 0.05) {
437                 sf::CircleShape lidarPoint(2);
438                 lidarPoint.setPosition(adjustedX, adjustedY);
439                 lidarPoint.setFillColor(getPointColor(point.
440                     range, max_range));
441
442                 window.draw(lidarPoint);
443             }
444             //std::cout << "An " << point.range << std :: endl;

```

```

438         }
439     } else {
440         std::cerr << "No se pudieron obtener los datos del
441             LiDAR." << std::endl;
442     }
443     window.display();
444 }
445
446 // Detener el escaneo del LiDAR
447 laser.turnOff();
448 laser.disconnecting();
449
450 // Cierra la tubera, detiene los motores y apaga el robot
451 pclose(pipe);
452 stopMotors();
453 gpioTerminate();
454
455 is_running = false;
456 randomMoveThread.join();
457
458 return 0;
459 }
```

5.2.4. Ajustes de código en función de las pruebas de carga 1

Las pruebas de carga realizadas al robot propuesto permitieron identificar ciertos problemas en el código de detección de obstáculos, que afectaban el desempeño del robot en situaciones de alta demanda. Para solucionar estos problemas, se realizaron ajustes en el código, con el objetivo de mejorar la precisión y eficiencia del sistema. Estos ajustes incluyeron la optimización de los algoritmos de detección de obstáculos, la reducción del tiempo de respuesta del robot y la mejora de la coordinación entre los motores y los sensores.

Los ajustes realizados permitieron al robot moverse de manera más fluida y precisa, evitando los obstáculos de manera oportuna y manteniendo una distancia segura en todo momento. Además, se mejoró la capacidad de respuesta del robot, permitiéndole ajustar su trayectoria de manera rápida y eficiente. Estos ajustes fueron fundamentales para garantizar el correcto funcionamiento del robot en situaciones de alta demanda y para mejorar su desempeño en entornos complejos. Esto se puede ver en el Listing 7.

Listing 7: Tercer ajuste de código

```

1 #include "CYdLidar.h"
2 #include <SFML/Graphics.hpp>
3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>
12 #include <thread>
13 #include <random>
14 #include <sys/socket.h>
15 #include <sys/un.h>
```

```

16 #include <unistd.h>
17 #include <cstring>
18 #include <pthread.h>
19 #include <vector>
20 #include <mutex>
21 #define PI 180.f
22
23 using namespace std;
24 using namespace ydlidar;
25
26 // Pines y configuración para los motores
27 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los motores
28 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de dirección para los
     motores
29 const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
30 int frequency = 400; // Frecuencia inicial
31
32 std::atomic<bool> is_running(true);
33 std::atomic<bool> is_manual_mode(true);
34
35 void setMotorSpeed(int motor, int frequency) {
36     if (motor >= 0 && motor < 4) {
37         gpioSetPWMfrequency(PWM_PINS[motor], frequency);
38     }
39 }
40
41 void setMotorDirection(int motor, int direction) {
42     if (motor >= 0 && motor < 4) {
43         gpioWrite(DIR_PINS[motor], direction);
44     }
45 }
46
47 void stopMotors() {
48     for (int i = 0; i < 4; ++i) {
49         gpioPWM(PWM_PINS[i], 0);
50     }
51 }
52
53 void moveForward() {
54     for (int i = 0; i < 4; ++i) {
55         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
56     }
57     setMotorDirection(0, 0); // Motor 1
58     setMotorDirection(2, 0); // Motor 3
59     setMotorDirection(1, 1); // Motor 2
60     setMotorDirection(3, 1); // Motor 4
61 }
62
63 void moveBackward() {
64     for (int i = 0; i < 4; ++i) {
65         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
66     }
67     setMotorDirection(0, 1); // Motor 1
68     setMotorDirection(2, 1); // Motor 3
69     setMotorDirection(1, 0); // Motor 2
70     setMotorDirection(3, 0); // Motor 4
71 }
72
73 void turnLeft() {
74     for (int i = 0; i < 4; ++i) {

```

```

75         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
76     }
77     setMotorDirection(0, 1); // Motor 1
78     setMotorDirection(2, 1); // Motor 3
79     setMotorDirection(1, 1); // Motor 2
80     setMotorDirection(3, 1); // Motor 4
81 }
82
83 void turnRight() {
84     for (int i = 0; i < 4; ++i) {
85         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
86     }
87     setMotorDirection(0, 0); // Motor 1
88     setMotorDirection(2, 0); // Motor 3
89     setMotorDirection(1, 0); // Motor 2
90     setMotorDirection(3, 0); // Motor 4
91 }
92
93 void increaseSpeed() {
94     if (frequency < 2000) {
95         frequency += 100;
96         if (frequency > 2000) frequency = 2000;
97         for (int i = 0; i < 4; ++i) {
98             setMotorSpeed(i, frequency);
99         }
100    }
101 }
102
103 void decreaseSpeed() {
104     if (frequency > 400) {
105         frequency -= 100;
106         if (frequency < 400) frequency = 400;
107         for (int i = 0; i < 4; ++i) {
108             setMotorSpeed(i, frequency);
109         }
110    }
111 }
112
113
114 sf::Color getPointColor(float distance, float maxRange) {
115     float ratio = distance / maxRange;
116     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de rojo
117     a verde
118 }
119 int contadorObstaculoTotal = 0;
120 int obstaculoHola = 0;
121 int obsRelativo = 0;
122 int idTipoObstaculo = 0;
123 void randomMovement(CYdLidar &laser) {
124     std::random_device rd;
125     std::mt19937 gen(rd());
126     std::discrete_distribution<> dist({0, 5}); // Distribucion para la
127     probabilidad de movimiento
128     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15 grados
129     en radianes
130     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15 grados
131     en radianes
132     const float DETECTION_RADIUS = 0.25f; // 35 cm

```

```

131
132     std::vector<float> previousScanPoints;
133
134     while (is_running) {
135         if (!is_manual_mode) {
136             LaserScan scan;
137             if (laser.doProcessSimple(scan)) {
138                 bool obstacle_detected = false;
139                 std::vector<float> currentScanPoints;
140
141                 for (const auto &point : scan.points) {
142                     // Convertir el ngulo del punto al ngulo relativo al "sur" del robot
143                     float adjusted_angle = point.angle + PI;
144                     //std::cout << "Er " << point.range << std::endl;
145                     // Verificar si el punto est dentro del rango frontal de 30
146                     //std::cout << point.angle << std::endl;
147                     if (point.range > 0 && point.range < 0.30 && ( point.angle <= -0.5235f && point.angle >= -2.617f )) {
148                         // Norte
149                         obstacle_detected = true;
150                         currentScanPoints.push_back(point.range);
151                         idTipoObstaculo = 1;
152                         std::cout << "Obstacle distance N: " << (float)
153                             point.range << " Y en el angulo " << point.angle << std::endl;
154                         break;
155                     } else if (point.range > 0 && point.range < 0.25 && ( point.angle <= 0.872f && point.angle >= -0.5235f )) {
156                         // Este
157                         obstacle_detected = true;
158                         currentScanPoints.push_back(point.range);
159                         idTipoObstaculo = 2;
160                         std::cout << "Obstacle distance E: " << (float)
161                             point.range << " Y en el angulo " << point.angle << std::endl;
162                         break;
163                     } else if (point.range > 0 && point.range < 0.45 && ( point.angle <= 2.26f && point.angle >= 0.872f )) {
164                         // Sur
165                         obstacle_detected = true;
166                         currentScanPoints.push_back(point.range);
167                         idTipoObstaculo = 3;
168                         std::cout << "Obstacle distance S: " << (float)
169                             point.range << " Y en el angulo " << point.angle << std::endl;
170                         break;
171                     } else if (point.range > 0 && point.range < 0.25 && ( point.angle <= -2.61f || point.angle >= 2.27f )) {
172                         // Oeste
173                         obstacle_detected = true;
174                         currentScanPoints.push_back(point.range);
175                         idTipoObstaculo = 4;
176                         std::cout << "Obstacle distance O: " << (float)
177                             point.range << " Y en el angulo " << point.angle << std::endl;
178                         break;
179                 }
180             }
181         }
182     }

```

```

173
174    }
175
176    if (obstacle_detected) {
177        // Si detecta un obstáculo, retrocede por 4 segundos
178        contadorObstaculoTotal++;
179        std::cout << "Obs: " << contadorObstaculoTotal << std
180            ::endl;
181        //obsRelativo++;
182        stopMotors();
183        //Luego gira aleatoriamente a la izquierda o derecha
184        switch(idTipoObstaculo) {
185            case 1: {
186                int turn = dist(gen) % 2;
187                moveBackward();
188                std::this_thread::sleep_for(std::chrono::
189                    seconds(5));
190                if (turn == 0) {
191                    turnLeft();
192                    std::this_thread::sleep_for(std::chrono::
193                        seconds(5));
194                } else {
195                    turnRight();
196                    std::this_thread::sleep_for(std::chrono::
197                        seconds(5));
198                }
199                std::this_thread::sleep_for(std::chrono::
200                    seconds(2));
201                stopMotors();
202            }
203            break;
204
205            case 2:
206                turnLeft();
207                std::this_thread::sleep_for(std::chrono::
208                    seconds(5));
209                std::this_thread::sleep_for(std::chrono::
210                    seconds(2));
211                stopMotors();
212            break;
213
214            case 3:// este es el 4 y el de abajo el 3
215                moveForward();
216                std::this_thread::sleep_for(std::chrono::
217                    seconds(2));
218                stopMotors();
219            break;
220
221            default: {
222                int turn = dist(gen) % 2;
223                if (turn == 0) {

```

```

224                         std::this_thread::sleep_for(std::chrono::
225                                         seconds(5));
226                     } else {
227                         turnRight();
228                         std::this_thread::sleep_for(std::chrono::
229                                         seconds(5));
230                     }
231                     stopMotors();
232                 }
233             break;
234         }
235
236         if(!previousScanPoints.empty() && previousScanPoints.
237             size() == currentScanPoints.size()){
238             bool mismoObs = true;
239             for(size_t i = 0; i < currentScanPoints.size()
240                 ; ++i){
241                 if(fabs(currentScanPoints[i] -
242                     previousScanPoints[i]) > 0.05){
243                     mismoObs = false;
244                     break;
245                 }
246             }
247             if(mismoObs){
248                 obsRelativo++;
249             }else{
250                 obsRelativo = 0;
251             }
252         }
253         previousScanPoints = currentScanPoints;
254
255         if((obsRelativo > 3)){
256             //obsRelativo = 0;
257             moveBackward();
258             std::this_thread::sleep_for(std::chrono::seconds
259                                         (3));
260             stopMotors();
261
262             switch(idTipoObstaculo) {
263                 case 1: {
264                     int turn = dist(gen) % 2;
265                     moveBackward();
266                     std::this_thread::sleep_for(std::chrono::
267                                         seconds(5));
268                     if (turn == 0) {
269                         turnLeft();
270                         std::this_thread::sleep_for(std::
271                                         chrono::seconds(5));
272                     } else {
273                         turnRight();
274                         std::this_thread::sleep_for(std::
275                                         chrono::seconds(5));
276                     }
277                     std::this_thread::sleep_for(std::chrono::
278                                         seconds(2));
279                 }
280             }
281         }
282     }
283 }
```

```

273                     stopMotors();
274                 }
275             break;
276
277         case 2:
278             turnLeft();
279             std::this_thread::sleep_for(std::chrono::
280                 seconds(5));
281             std::this_thread::sleep_for(std::chrono::
282                 seconds(2));
283             stopMotors();
284             break;
285
286         case 3:
287             turnRight();
288             std::this_thread::sleep_for(std::chrono::
289                 seconds(7));
290             stopMotors();
291             break;
292
293         case 4:
294             moveForward();
295             std::this_thread::sleep_for(std::chrono::
296                 seconds(2));
297             stopMotors();
298             break;
299
300         default: {
301             int turn = dist(gen) % 2;
302             if (turn == 0) {
303                 turnLeft();
304                 std::this_thread::sleep_for(std::chrono::
305                     seconds(5));
306             } else {
307                 turnRight();
308                 std::this_thread::sleep_for(std::chrono::
309                     seconds(5));
310             }
311             std::this_thread::sleep_for(std::chrono::
312                     seconds(2));
313             stopMotors();
314         }
315     }
316     moveForward();
317     // Si no hay obstculo, elige un movimiento aleatorio
318     //
319     //int move = dist(gen);
320     //switch (move) {
321     //    case 0: moveBackward(); break;
322     //    case 1: turnRight(); break;
323     //    case 2: turnLeft(); break;
324     //    case 3: moveForward(); break;
325     //}
326 }
```

```

326     }
327
328     std::this_thread::sleep_for(std::chrono::milliseconds(500));
329     //stopMotors();
330   }
331   std::this_thread::sleep_for(std::chrono::milliseconds(100));
332 }
333 }
334
335 int main() {
336   // Inicializar pigpio
337   if (gpioInitialise() < 0) {
338     std::cerr << "Error: No se pudo inicializar pigpio." << std::endl;
339     return -1;
340   }
341
342   // Configurar pines de direccion como salida
343   for (int i = 0; i < 4; ++i) {
344     gpioSetMode(DIR_PINS[i], PI_OUTPUT);
345     gpioSetMode(PWM_PINS[i], PI_OUTPUT);
346     setMotorSpeed(i, frequency); // Inicializar PWM con frecuencia
347     inicial
348   }
349
350   // Asegurate de establecer XDG_RUNTIME_DIR
351   if (getenv("XDG_RUNTIME_DIR") == nullptr) {
352     setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
353   }
354
355   // Ejecuta libcamera-vid en un proceso separado y capture la salida en
356   // YUV, sin previsualizacn
357   FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview -o -
358   ", "r");
359   if (!pipe) {
360     std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::endl;
361     return -1;
362   }
363
364   // Configura la ventana SFML
365   sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
366   Visualization with LiDAR");
367   sf::Texture cameraTexture;
368   sf::Sprite cameraSprite;
369
370   // Buffer para leer los datos de video
371   const int width = 640;
372   const int height = 480;
373   std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
374   tamao del buffer para YUV420
375
376   cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data());
377   cv::Mat rgbImage(height, width, CV_8UC3);
378
379   std::string port;
380   ydlidar::os_init();
381
382   // Obtener los puertos disponibles de LiDAR
383   std::map<std::string, std::string> ports = ydlidar::lidarPortList();
384   if (ports.size() > 1) {

```

```

380         auto it = ports.begin();
381         std::advance(it, 1); // Selecciona el segundo puerto disponible
382         port = it->second;
383     } else if (ports.size() == 1) {
384         port = ports.begin()->second;
385     } else {
386         std::cerr << "No se detect ningn LiDAR. Verifica la conexin." <<
387             std::endl;
388         return -1;
389     }
390
391     // Configuracin del LiDAR
392     int baudrate = 115200;
393     std::cout << "Baudrate: " << baudrate << std::endl;
394
395     CYdLidar laser;
396     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
397     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int));
398
399     bool isSingleChannel = true;
400     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
401         bool));
402
403     float max_range = 8.0f;
404     float min_range = 0.1f;
405     float max_angle = 180.0f;
406     float min_angle = -180.0f;
407     float frequency = 8.0f;
408
409     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
410     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
411     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
412     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
413     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float));
414
415     // Inicializar LiDAR
416     if (!laser.initialize()) {
417         std::cerr << "Error al inicializar el LiDAR." << std::endl;
418         return -1;
419     }
420
421     // Iniciar el escaneo
422     if (!laser.turnOn()) {
423         std::cerr << "Error al encender el LiDAR." << std::endl;
424         return -1;
425     }
426
427     // Thread para movimiento aleatorio
428     std::thread randomMoveThread(randomMovement, std::ref(laser));
429
430     while (window.isOpen()) {
431         sf::Event event;
432         while (window.pollEvent(event)) {
433             if (event.type == sf::Event::Closed)
434                 window.close();
435
436             if (event.type == sf::Event::KeyPressed) {
437                 switch (event.key.code) {
438                     case sf::Keyboard::W:
439                         is_manual_mode = true;

```

```

438                     moveForward();
439                     break;
440                 case sf::Keyboard::S:
441                     is_manual_mode = true;
442                     moveBackward();
443                     break;
444                 case sf::Keyboard::A:
445                     is_manual_mode = true;
446                     turnLeft();
447                     break;
448                 case sf::Keyboard::D:
449                     is_manual_mode = true;
450                     turnRight();
451                     break;
452                 case sf::Keyboard::V:
453                     is_manual_mode = true;
454                     increaseSpeed();
455                     break;
456                 case sf::Keyboard::B:
457                     is_manual_mode = true;
458                     decreaseSpeed();
459                     break;
460                 case sf::Keyboard::Space:
461                     is_manual_mode = true;
462                     stopMotors();
463                     break;
464                 case sf::Keyboard::K:
465                     is_manual_mode = false;
466                     break;
467                 case sf::Keyboard::M:
468                     is_manual_mode = true;
469                     break;
470             default:
471                 break;
472         }
473     }
474 }
475
476 // Leer los datos del video desde la tubera
477 size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe);
478 if (bytesRead != buffer.size()) {
479     std::cerr << "Error: No se pudo leer suficientes datos de
480         video." << std::endl;
481     continue;
482 }
483
484 // Convertir YUV420 a RGB
485 cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
486
487 // Convertir a RGBA añadiendo un canal alfa
488 cv::Mat frame_rgba;
489 cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
490
491 // Actualizar la textura de la cámara con los datos del frame
492 if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
493     std::cerr << "Error: No se pudo crear la textura." << std::
494         endl;
495     continue;
496 }
497 cameraTexture.update(frame_rgba.ptr());

```

```

496
497     cameraSprite.setTexture(cameraTexture);
498     cameraSprite.setScale(
499         window.getSize().x / static_cast<float>(cameraTexture.getSize()
500             () .x),
501         window.getSize().y / static_cast<float>(cameraTexture.getSize()
502             () .y)
503     );
504
505     window.clear();
506     window.draw(cameraSprite);
507
508     // Crear un minimapa para el LiDAR
509     sf::RectangleShape minimap(sf::Vector2f(200, 200));
510     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
511     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // semitransparente
512     minimap.setPosition(10, 10); // Esquina superior izquierda
513
514     window.draw(minimap);
515
516     // Dibujar el centro del LiDAR (color azul)
517     sf::CircleShape lidarCenter(5); // Radio del círculo del LiDAR
518     lidarCenter.setFillColor(sf::Color::Blue);
519     lidarCenter.setPosition(105, 105); // Posición del centro en el
520     minimapa
521
522     window.draw(lidarCenter);
523
524     // Dibujar la línea hacia el norte
525     sf::Vertex line[] =
526     {
527         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
528         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) // Línea
529         hacia arriba (norte)
530     };
531
532     window.draw(line, 2, sf::Lines);
533
534     LaserScan scan;
535     if (laser.doProcessSimple(scan)) {
536         for (const auto& point : scan.points) {
537             // Convertir coordenadas polares a cartesianas
538             float x = point.range * cos(point.angle);
539             float y = point.range * sin(point.angle);
540
541             // Ajustar los puntos al minimapa
542             float scale = 25.0f;
543             float adjustedX = 110 + x * scale;
544             float adjustedY = 110 - y * scale; // Invertir Y para
545             coordinar con la pantalla
546
547             // Dibujar los puntos en el minimapa, excluyendo el centro
548             (0,0)
549             if (point.range > 0.05) {
550                 sf::CircleShape lidarPoint(2);
551                 lidarPoint.setPosition(adjustedX, adjustedY);
552                 lidarPoint.setFillColor(getPointColor(point.range,
553                     max_range));
554
555                 window.draw(lidarPoint);
556             }
557         }
558     }
559
560     window.display();
561
562     std::this_thread::sleep(std::chrono::milliseconds(10));
563
564     if (sf::Keyboard::isKeyPressed(sf::Keyboard::Escape))
565         break;
566 }

```

```

548         }
549         //std::cout << "An " << point.range << std :: endl;
550     }
551 } else {
552     std::cerr << "No se pudieron obtener los datos del LiDAR." <<
553     std::endl;
554 }
555 window.display();
556 }
557
558 // Detener el escaneo del LiDAR
559 laser.turnOff();
560 laser.disconnecting();
561
562 // Cierra la tubera, detiene los motores y apaga el robot
563 pclose(pipe);
564 stopMotors();
565 gpioTerminate();
566
567 is_running = false;
568 randomMoveThread.join();
569
570 return 0;
571 }

```

5.2.5. Ajustes de código en función de pruebas unitarias y de aceptación 3

Durante el desarrollo del sistema, se realizaron pruebas unitarias y de aceptación para verificar el correcto funcionamiento de los módulos y componentes del sistema. Estas pruebas permitieron identificar errores y fallas en el código, así como realizar ajustes y mejoras en el sistema. A continuación, se describen los ajustes realizados en función de las pruebas unitarias y de aceptación.

En el módulo de control de movimiento, se realizaron ajustes para mejorar la precisión y eficiencia del movimiento del robot. Se implementó una función de seguimiento de rutas por línea, que permite al robot seguir una línea dibujada en el minimapa con el mouse. Esta función utiliza los sensores de distancia LIDAR para detectar la línea y ajustar la trayectoria del robot en tiempo real. Además, se optimizó el algoritmo de control de movimiento para reducir el tiempo de respuesta y mejorar la precisión del movimiento.

En el módulo de control de visión, se realizaron ajustes para mejorar la precisión y eficiencia de la detección de obstáculos. Se implementó un algoritmo de detección de obstáculos basado en visión por computadora, que utiliza la cámara de visión nocturna para identificar obstáculos en el entorno del robot. Este algoritmo utiliza técnicas de procesamiento de imágenes para detectar objetos y calcular su distancia y posición relativa al robot. Además, se optimizó el algoritmo de detección de obstáculos para reducir el tiempo de procesamiento y mejorar la precisión de la detección.

Estos ajustes se pueden observar en el código fuente del sistema, que se presenta a continuación en el Listing 8.

Listing 8: Cuarto ajuste de código

```
1 #include "CYdLidar.h"
```

```

2 #include <SFML/Graphics.hpp>
3 #include <opencv2/opencv.hpp>
4 #include <iostream>
5 #include <cstdlib>
6 #include <cstdio>
7 #include <pigpio.h>
8 #include <string>
9 #include <map>
10 #include <vector>
11 #include <atomic>
12 #include <thread>
13 #include <random>
14 #include <sys/socket.h>
15 #include <sys/un.h>
16 #include <unistd.h>
17 #include <cstring>
18 #include <pthread.h>
19 #include <vector>
20 #include <mutex>
21 #define PI 180.f
22
23 using namespace std;
24 using namespace ydlidar;
25
26 // Pines y configuración para los motores
27 const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los motores
28 const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de dirección para los
     motores
29 const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
30 int frequency = 400; // Frecuencia inicial
31 const sf::Vector2f robotFixedPosition(110, 110);
32 std::atomic<bool> is_running(true);
33 std::atomic<bool> is_manual_mode(true);
34
35 void setMotorSpeed(int motor, int frequency) {
36     if (motor >= 0 && motor < 4) {
37         gpioSetPWMfrequency(PWM_PINS[motor], frequency);
38     }
39 }
40
41 void setMotorDirection(int motor, int direction) {
42     if (motor >= 0 && motor < 4) {
43         gpioWrite(DIR_PINS[motor], direction);
44     }
45 }
46
47 void stopMotors() {
48     for (int i = 0; i < 4; ++i) {
49         gpioPWM(PWM_PINS[i], 0);
50     }
51 }
52
53 void moveForward() {
54     for (int i = 0; i < 4; ++i) {
55         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
56     }
57     setMotorDirection(0, 0); // Motor 1
58     setMotorDirection(2, 0); // Motor 3
59     setMotorDirection(1, 1); // Motor 2
60     setMotorDirection(3, 1); // Motor 4

```

```

61 }
62
63 void moveBackward() {
64     for (int i = 0; i < 4; ++i) {
65         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
66     }
67     setMotorDirection(0, 1); // Motor 1
68     setMotorDirection(2, 1); // Motor 3
69     setMotorDirection(1, 0); // Motor 2
70     setMotorDirection(3, 0); // Motor 4
71 }
72
73 void turnLeft() {
74     for (int i = 0; i < 4; ++i) {
75         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
76     }
77     setMotorDirection(0, 1); // Motor 1
78     setMotorDirection(2, 1); // Motor 3
79     setMotorDirection(1, 1); // Motor 2
80     setMotorDirection(3, 1); // Motor 4
81 }
82
83 void turnRight() {
84     for (int i = 0; i < 4; ++i) {
85         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo al 50%
86     }
87     setMotorDirection(0, 0); // Motor 1
88     setMotorDirection(2, 0); // Motor 3
89     setMotorDirection(1, 0); // Motor 2
90     setMotorDirection(3, 0); // Motor 4
91 }
92
93 void increaseSpeed() {
94     if (frequency < 2000) {
95         frequency += 100;
96         if (frequency > 2000) frequency = 2000;
97         for (int i = 0; i < 4; ++i) {
98             setMotorSpeed(i, frequency);
99         }
100    }
101 }
102
103 void decreaseSpeed() {
104     if (frequency > 400) {
105         frequency -= 100;
106         if (frequency < 400) frequency = 400;
107         for (int i = 0; i < 4; ++i) {
108             setMotorSpeed(i, frequency);
109         }
110    }
111 }
112
113
114 sf::Color getPointColor(float distance, float maxRange) {
115     float ratio = distance / maxRange;
116     return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color de rojo
117     a verde
118 }
119

```

```

120 std::vector<sf::Vector2f> routePoints;
121 bool isDrawing = false; // Bandera para rastrear si el raton esta
122 presionado
123
124
125
126 void drawLiDARPoints(sf::RenderWindow &window, const LaserScan &
127 lidarPoints, const sf::Vector2f &robotPosition, float scale, float
128 max_range) {
129     for (const auto& point : lidarPoints.points) {
130         float x = point.range * cos(point.angle);
131         float y = point.range * sin(point.angle);
132
133         float adjustedX = robotFixedPosition.x + (x * scale) -
134             robotPosition.x;
135         float adjustedY = robotFixedPosition.y - (y * scale) -
136             robotPosition.y; // Invertir Y para la pantalla
137
138         if (point.range > 0.05) { // Excluir puntos cercanos al centro
139             sf::CircleShape lidarPoint(2); // Tama\~no del punto
140             lidarPoint.setPosition(adjustedX, adjustedY);
141             lidarPoint.setFillColor(getPointColor(point.range, max_range))
142                 ; // Color basado en la distancia
143
144             window.draw(lidarPoint);
145     }
146 }
147
148
149 void handleMouseClick(const sf::Event &event, const sf::RectangleShape &
150 minimap) {
151     // Verificar si el raton esta presionado
152     if (event.type == sf::Event::MouseButtonPressed && event.mouseButton.
153         button == sf::Mouse::Left) {
154         isDrawing = true; // Marcar que estamos dibujando
155
156         // Verificar si el raton fue soltado
157     } else if (event.type == sf::Event::MouseButtonReleased && event.
158 mouseButton.button == sf::Mouse::Left) {
159         isDrawing = false; // Dejar de dibujar
160     }
161 }
162
163 void updateRoute(const sf::RectangleShape &minimap, sf::RenderWindow &
164 window) {
165     // Si el raton esta presionado, agregar puntos a la ruta
166     if (isDrawing) {
167         // Obtener la posicion actual del raton
168         sf::Vector2i mousePosition = sf::Mouse::getPosition(window);
169         sf::Vector2f clickPosition(static_cast<float>(mousePosition.x),
170             static_cast<float>(mousePosition.y));
171
172         // Verificar si la posicion del raton esta dentro del minimapa
173         if (minimap.getGlobalBounds().contains(clickPosition)) {

```

```

169         sf::Vector2f relativePosition = clickPosition - minimap.
170             getPosition();
171         routePoints.push_back(relativePosition);
172         std::cout << "Dibujando punto: (" << relativePosition.x << ","
173             " << relativePosition.y << ")" << std::endl;
174     }
175
176
177
178 void drawRoute(sf::RenderWindow &window, const std::vector<sf::Vector2f> &
179   routePoints, const sf::RectangleShape &minimap) {
180     if (routePoints.size() > 1) {
181       for (size_t i = 0; i < routePoints.size() - 1; ++i) {
182         // Dibujar las l\'ineas de la ruta, ajustando las coordenadas
183         // en funci\'on de la posici\'on del robot
184         sf::Vertex line[] = {
185           sf::Vertex(routePoints[i] + minimap.getPosition(), sf::
186             Color::Red), // Ajuste de posici\'on para el minimapa
187           sf::Vertex(routePoints[i + 1] + minimap.getPosition(), sf
188             ::Color::Red)
189         };
190         window.draw(line, 2, sf::Lines);
191     }
192   }
193
194
195 // Ajustar la referencia para que el norte sea -1.57 rad (sur en
196 // coordenadas cartesianas es 0 rad)
197 float adjustAngleForNorth(float angle) {
198   float adjustedAngle = angle - (-M_PI / 2); // Ajuste de 90 (norte en
199   // -90)
200
201   // Asegurarse de que el angulo este dentro de los limites de -PI a PI
202   if (adjustedAngle > M_PI) {
203     adjustedAngle -= 2 * M_PI;
204   } else if (adjustedAngle < -M_PI) {
205     adjustedAngle += 2 * M_PI;
206   }
207
208
209
210
211
212
213
214
215 void rotateTowardsPoint(float currentAngle, float targetAngle) {
216   float angleDifference = targetAngle - currentAngle;
217
218   // Asegurate de que el angulo este entre -PI y PI
219   if (angleDifference > M_PI) {
220     angleDifference -= 2 * M_PI;

```

```

221 } else if (angleDifference < -M_PI) {
222     angleDifference += 2 * M_PI;
223 }
224
225 // Gira en la direccion correcta
226 if (angleDifference > 0) {
227     turnRight(); // Girar hacia la derecha
228 } else {
229     turnLeft(); // Girar hacia la izquierda
230 }
231
232 // Esperar hasta que el robot este alineado
233 std::this_thread::sleep_for(std::chrono::milliseconds(static_cast<int>(std::fabs(angleDifference) * 1000)));
234
235 stopMotors();
236 }
237
238
239
240
241
242
243 void moveTowardsPoint(float distance) {
244     moveForward(); // Mover el robot hacia adelante
245
246     // Esperar un tiempo proporcional a la distancia
247     std::this_thread::sleep_for(std::chrono::milliseconds(static_cast<int>(distance * 1000)));
248
249     stopMotors(); // Detener el robot
250 }
251
252
253
254
255 void moveTo(sf::Vector2f point, sf::Vector2f &robotPosition, float &currentAngle) {
256     // Calcular la diferencia en X e Y
257     float deltaX = point.x - robotPosition.x;
258     float deltaY = robotPosition.y - point.y; // Invertir la Y para la pantalla
259
260     // Calcular la distancia al punto
261     float distanceToPoint = sqrt(deltaX * deltaX + deltaY * deltaY);
262
263     // Si la distancia es mayor a un umbral, mover el robot
264     const float threshold = 5.0f; // Tolerancia para considerar que el punto ha sido alcanzado
265     if (distanceToPoint > threshold) {
266         // Calcular el angulo hacia el punto de destino
267         float targetAngle = atan2(deltaY, deltaX);
268
269         // Ajustar el angulo para que el norte sea -90°
270         targetAngle = adjustAngleForNorth(targetAngle);
271
272         // Girar hacia el angulo correcto
273         rotateTowardsPoint(currentAngle, targetAngle); // Funcion para girar hacia el \'angulo
274

```

```

275     // Mover hacia el punto
276     moveTowardsPoint(distanceToPoint);
277
278     // Actualizar la posicion y el \'angulo del robot
279     robotPosition = point;
280     currentAngle = targetAngle;
281 }
282 }
283
284
285
286 void followRoute(sf::RenderWindow &window, std::vector<sf::Vector2f> &
287     routePoints, const LaserScan &lidarPoints, const sf::Vector2f &
288     minimapPosition, float max_range) {
289     sf::Vector2f robotPosition(110, 110); // Posici\on inicial del robot
290         en el minimapa
291     float currentAngle = -M_PI / 2; // \'angulo inicial del robot (norte
292         en -90\circ)
293
294     while (!routePoints.empty()) {
295         sf::Vector2f targetPoint = routePoints.front();
296
297         moveTo(targetPoint, robotPosition, currentAngle);
298
299         routePoints.erase(routePoints.begin());
300
301
302
303
304
305
306
307
308
309
310
311     int contadorObstaculoTotal = 0;
312     int obstaculoHola = 0;
313     int obsRelativo = 0;
314     int idTipoObstaculo = 0;
315     void randomMovement(CYdLidar &laser) {
316         std::random_device rd;
317         std::mt19937 gen(rd());
318         std::discrete_distribution<> dist({0, 5}); // Distribucion para la
319             probabilidad de movimiento
320
321         const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15 grados
322             en radianes
323         const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15 grados
324             en radianes
325         const float DETECTION_RADIUS = 0.25f; // 35 cm
326
327         std::vector<float> previousScanPoints;
328
329         while (is_running) {

```

```

327     if (!is_manual_mode) {
328         LaserScan scan;
329         if (laser.doProcessSimple(scan)) {
330             bool obstacle_detected = false;
331             std::vector<float> currentScanPoints;
332
333             for (const auto &point : scan.points) {
334                 // Convertir el ngulo del punto al ngulo relativo al "sur" del robot
335                 float adjusted_angle = point.angle + PI;
336                 //std::cout << "Er " << point.range << std::endl;
337                 // Verificar si el punto est dentro del rango frontal de 30
338                 //std::cout << point.angle << std::endl;
339                 if (point.range > 0 && point.range < 0.30 && ( point.
340                     angle <= -0.5235f && point.angle >= -2.617f)) {
341                     // Norte
342                     obstacle_detected = true;
343                     currentScanPoints.push_back(point.range);
344                     idTipoObstaculo = 1;
345                     std::cout << "Obstacle distance N: " << (float)
346                         point.range << " Y en el angulo " << point.
347                             angle << std::endl;
348                     break;
349                 } else if (point.range > 0 && point.range < 0.25 && (
350                     point.angle <= 0.872f && point.angle >= -0.5235f))
351                 { // Este
352                     obstacle_detected = true;
353                     currentScanPoints.push_back(point.range);
354                     idTipoObstaculo = 2;
355                     std::cout << "Obstacle distance E: " << (float)
356                         point.range << " Y en el angulo " << point.
357                             angle << std::endl;
358                     break;
359                 } else if (point.range > 0 && point.range < 0.45 && (
360                     point.angle <= 2.26f && point.angle >= 0.872f)){
361                     // Sur
362                     obstacle_detected = true;
363                     currentScanPoints.push_back(point.range);
364                     idTipoObstaculo = 3;
365                     std::cout << "Obstacle distance S: " << (float)
366                         point.range << " Y en el angulo " << point.
367                             angle << std::endl;
368                     break;
369                 } else if (point.range > 0 && point.range < 0.25 && (
370                     point.angle <= -2.61f || point.angle >= 2.27f)){
371                     // Oeste
372                     obstacle_detected = true;
373                     currentScanPoints.push_back(point.range);
374                     idTipoObstaculo = 4;
375                     std::cout << "Obstacle distance O: " << (float)
376                         point.range << " Y en el angulo " << point.
377                             angle << std::endl;
378                     break;
379                 }
380             }
381
382             if (obstacle_detected) {

```

```

369 // Si detecta un obstaculo, retrocede por 4 segundos
370 contadorObstaculoTotal++;
371 std::cout << "Obs: " << contadorObstaculoTotal << std
372 ::endl;
373 //obsRelativo++;
374 stopMotors();
375 //Luego gira aleatoriamente a la izquierda o derecha
376 switch(idTipoObstaculo) {
377     case 1: {
378         int turn = dist(gen) % 2;
379         moveBackward();
380         std::this_thread::sleep_for(std::chrono::
381             seconds(5));
382         if (turn == 0) {
383             turnLeft();
384             std::this_thread::sleep_for(std::chrono::
385                 seconds(5));
386         } else {
387             turnRight();
388             std::this_thread::sleep_for(std::chrono::
389                 seconds(5));
390         }
391         stopMotors();
392     }
393     break;
394
395     case 2:
396         turnLeft();
397         std::this_thread::sleep_for(std::chrono::
398             seconds(5));
399         std::this_thread::sleep_for(std::chrono::
400             seconds(2));
401         stopMotors();
402     break;
403
404     case 3:// este es el 4 y el de abajo el 3
405         moveForward();
406         std::this_thread::sleep_for(std::chrono::
407             seconds(2));
408         stopMotors();
409
410     break;
411
412     default: {
413         int turn = dist(gen) % 2;
414         if (turn == 0) {
415             turnLeft();
416             std::this_thread::sleep_for(std::chrono::
417                 seconds(5));
418         } else {
419             turnRight();

```

```

419                         std::this_thread::sleep_for(std::chrono::
420                                         seconds(5));
421                     }
422                     std::this_thread::sleep_for(std::chrono::
423                                         seconds(2));
424                     stopMotors();
425                 }
426             break;
427         }
428
429         if(!previousScanPoints.empty() && previousScanPoints.
430             size() == currentScanPoints.size()){
431             bool mismoObs = true;
432             for(size_t i = 0; i < currentScanPoints.size()
433                 ; ++i){
434                 if(fabs(currentScanPoints[i] -
435                     previousScanPoints[i]) > 0.05){
436                     mismoObs = false;
437                     break;
438                 }
439             }
440             if(mismoObs){
441                 obsRelativo++;
442             }else{
443                 obsRelativo = 0;
444             }
445         }
446         previousScanPoints = currentScanPoints;
447
448         if((obsRelativo > 3)){
449             //obsRelativo = 0;
450             moveBackward();
451             std::this_thread::sleep_for(std::chrono::seconds
452                                         (3));
453             stopMotors();
454
455             switch(idTipoObstaculo) {
456                 case 1: {
457                     int turn = dist(gen) % 2;
458                     moveBackward();
459                     std::this_thread::sleep_for(std::chrono::
460                                         seconds(5));
461                     if (turn == 0) {
462                         turnLeft();
463                         std::this_thread::sleep_for(std::chrono::
464                                         seconds(5));
465                     } else {
466                         turnRight();
467                         std::this_thread::sleep_for(std::chrono::
468                                         seconds(5));
469                     }
470                     std::this_thread::sleep_for(std::chrono::
471                                         seconds(2));
472                     stopMotors();
473                 }
474             break;
475         }

```

```

469             case 2:
470                 turnLeft();
471                 std::this_thread::sleep_for(std::chrono::
472                     seconds(5));
473                 std::this_thread::sleep_for(std::chrono::
474                     seconds(2));
475                 stopMotors();
476             break;
477
478             case 3:
479                 turnRight();
480                 std::this_thread::sleep_for(std::chrono::
481                     seconds(7));
482                 stopMotors();
483             break;
484
485             case 4:
486                 moveForward();
487                 std::this_thread::sleep_for(std::chrono::
488                     seconds(2));
489                 stopMotors();
490             break;
491
492         default: {
493             int turn = dist(gen) % 2;
494             if (turn == 0) {
495                 turnLeft();
496                 std::this_thread::sleep_for(std::chrono::
497                     seconds(5));
498             } else {
499                 turnRight();
500                 std::this_thread::sleep_for(std::chrono::
501                     seconds(5));
502             }
503             stopMotors();
504         }
505     }
506 } else {
507     moveForward();
508     // Si no hay obstculo, elige un movimiento aleatorio
509     //
510     //int move = dist(gen);
511     //switch (move) {
512     //    case 0: moveBackward(); break;
513     //    case 1: turnRight(); break;
514     //    case 2: turnLeft(); break;
515     //    case 3: moveForward(); break;
516     //}
517 }
518
519 std::this_thread::sleep_for(std::chrono::milliseconds(500));
520 //stopMotors();

```

```

522         }
523         std::this_thread::sleep_for(std::chrono::milliseconds(100));
524     }
525 }
526
527
528
529
530
531
532 int main() {
533     // Inicializar pigpio
534     if (gpioInitialise() < 0) {
535         std::cerr << "Error: No se pudo inicializar pigpio." << std::endl;
536         return -1;
537     }
538
539     // Configurar pines de direccin como salida
540     for (int i = 0; i < 4; ++i) {
541         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
542         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
543         setMotorSpeed(i, frequency); // Inicializar PWM con frecuencia
544             inicial
545     }
546
547     // Asegrate de establecer XDG_RUNTIME_DIR
548     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
549         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
550     }
551
552     // Ejecuta libcamera-vid en un proceso separado y captura la salida en
553         YUV, sin previsualizacn
554     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview -o -
555         ", "r");
556     if (!pipe) {
557         std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::endl;
558         return -1;
559     }
560
561     // Configura la ventana SFML
562     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
563         Visualization with LiDAR");
564     sf::Texture cameraTexture;
565     sf::Sprite cameraSprite;
566
567     // Buffer para leer los datos de video
568     const int width = 640;
569     const int height = 480;
570     std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta el
571         tamao del buffer para YUV420
572
573     cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.data());
574     cv::Mat rgbImage(height, width, CV_8UC3);
575
576     std::string port;
577     ydlidar::os_init();
578
579     // Obtener los puertos disponibles de LiDAR
580     std::map<std::string, std::string> ports = ydlidar::lidarPortList();

```

```

576 if (ports.size() > 1) {
577     auto it = ports.begin();
578     std::advance(it, 1); // Selecciona el segundo puerto disponible
579     port = it->second;
580 } else if (ports.size() == 1) {
581     port = ports.begin()->second;
582 } else {
583     std::cerr << "No se detect ningn LiDAR. Verifica la conexin." <<
584         std::endl;
585     return -1;
586 }
587
588 // Configuracin del LiDAR
589 int baudrate = 115200;
590 std::cout << "Baudrate: " << baudrate << std::endl;
591
592 CYdLidar laser;
593 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
594 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int));
595
596 bool isSingleChannel = true;
597 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
598     bool));
599
600 float max_range = 8.0f;
601 float min_range = 0.1f;
602 float max_angle = 180.0f;
603 float min_angle = -180.0f;
604 float frequency = 8.0f;
605
606 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float));
607 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float));
608 laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float));
609 laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float));
610 laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float));
611
612 // Inicializar LiDAR
613 if (!laser.initialize()) {
614     std::cerr << "Error al inicializar el LiDAR." << std::endl;
615     return -1;
616 }
617
618 // Iniciar el escaneo
619 if (!laser.turnOn()) {
620     std::cerr << "Error al encender el LiDAR." << std::endl;
621     return -1;
622 }
623
624 // Thread para movimiento aleatorio
625 std::thread randomMoveThread(randomMovement, std::ref(laser));
626
627 while (window.isOpen()) {
628     // Crear un minimapa para el LiDAR
629     sf::RectangleShape minimap(sf::Vector2f(200, 200));
630     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo
631         semitransparente
632     minimap.setPosition(10, 10); // Esquina superior izquierda
633
634     // Leer los datos del video desde la tubera

```

```

633     size_t bytesRead = fread(buffer.data(), 1, buffer.size(), pipe);
634     if (bytesRead != buffer.size()) {
635         std::cerr << "Error: No se pudo leer suficientes datos de
636             video." << std::endl;
637         continue;
638     }
639
640     // Convertir YUV420 a RGB
641     cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
642
643     // Convertir a RGBA añadiendo un canal alfa
644     cv::Mat frame_rgba;
645     cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
646
647     // Actualizar la textura de la cámara con los datos del frame
648     if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows)) {
649         std::cerr << "Error: No se pudo crear la textura." << std::
650             endl;
651         continue;
652     }
653     cameraTexture.update(frame_rgba.ptr());
654
655     cameraSprite.setTexture(cameraTexture);
656     cameraSprite.setScale(
657         window.getSize().x / static_cast<float>(cameraTexture.getSize()
658             (.x),
659             window.getSize().y / static_cast<float>(cameraTexture.getSize()
660                 (.y)
661             );
662
663     // Actualizar la ruta en el minimapa
664
665     window.clear();
666     window.draw(cameraSprite);
667
668     window.draw(minimap);
669     updateRoute(minimap, window);
670     // Dibujar la ruta en el minimapa
671     drawRoute(window, routePoints, minimap);
672     // Dibujar el centro del LiDAR (color azul)
673     sf::CircleShape lidarCenter(5); // Radio del círculo del LiDAR
674     lidarCenter.setFillColor(sf::Color::Blue);
675     lidarCenter.setPosition(105, 105); // Posición del centro en el
676         minimapa
677
678     window.draw(lidarCenter);
679
680     // Dibujar la línea hacia el norte
681     sf::Vertex line[] =
682     {
683         sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
684         sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black) // Línea
685             hacia arriba (norte)
686     };
687
688     window.draw(line, 2, sf::Lines);
689
690     LaserScan scan;

```

```

687     if (laser.doProcessSimple(scan)) {
688         drawLiDARPoints(window, scan, minimap.getPosition(), 25.0f,
689                         max_range);
690     }else {
691         std::cerr << "No se pudieron obtener los datos del LiDAR." <<
692                         std::endl;
693     }
694
695     sf::Event event;
696     while (window.pollEvent(event)) {
697         if (event.type == sf::Event::Closed){
698             window.close();
699         }
700
701         if(event.type == sf::Event::MouseButtonPressed){
702             std::cout << "Mouse pressed" << std::endl;
703             handleMouseClicked(event, minimap);
704         }else if(event.type == sf::Event::MouseButtonReleased){
705             std::cout << "Mouse released" << std::endl;
706             handleMouseClicked(event, minimap);
707             isDrawing = false;
708         }
709
710         if (event.type == sf::Event::KeyPressed) {
711             switch (event.key.code) {
712                 case sf::Keyboard::W:
713                     is_manual_mode = true;
714                     std::cout << "W" << std::endl;
715                     moveForward();
716                     break;
717                 case sf::Keyboard::S:
718                     is_manual_mode = true;
719                     std::cout << "S" << std::endl;
720                     moveBackward();
721                     break;
722                 case sf::Keyboard::A:
723                     is_manual_mode = true;
724                     std::cout << "A" << std::endl;
725                     turnLeft();
726                     break;
727                 case sf::Keyboard::D:
728                     is_manual_mode = true;
729                     std::cout << "D" << std::endl;
730                     turnRight();
731                     break;
732                 case sf::Keyboard::V:
733                     is_manual_mode = true;
734                     std::cout << "V" << std::endl;
735                     increaseSpeed();
736                     break;
737                 case sf::Keyboard::B:
738                     is_manual_mode = true;
739                     std::cout << "B" << std::endl;
740                     decreaseSpeed();
741                     break;
742                 case sf::Keyboard::Space:
743                     is_manual_mode = true;

```

```

745             std::cout << "Space" << std::endl;
746             stopMotors();
747             break;
748         case sf::Keyboard::R: {
749             is_manual_mode = false;
750             std::thread routeThread(followRoute, std::ref(
751                 window), std::ref(routePoints), std::ref(scan)
752                 , minimap.getPosition(), max_range);
753             routeThread.detach();
754             break;
755         }
756         case sf::Keyboard::K:
757             is_manual_mode = false;
758             std::cout << "K" << std::endl;
759             break;
760         case sf::Keyboard::M:
761             is_manual_mode = true;
762             std::cout << "M" << std::endl;
763             break;
764         default:
765             break;
766         }
767     }
768
769     window.display();
770 }
771
772 // Detener el escaneo del LiDAR
773 laser.turnOff();
774 laser.disconnecting();
775
776 // Cierra la tubera, detiene los motores y apaga el robot
777 pclose(pipe);
778 stopMotors();
779 gpioTerminate();
780
781 is_running = false;
782 randomMoveThread.join();
783
784 return 0;
785 }
```

Después de eso, se mejoró o se cambió más bien la función de seguimiento de rutas por medio de un valor numérico, que se obtiene del algoritmo del Physarum, se puede ver la función en el Listing 9.

Listing 9: Sexto ajuste de código

```

1      void moveAlongRoute(const std::vector<int> &route) {
2          // Las celdas están numeradas de 1 a N como en la imagen
3          // proporcionada.
4          std::map<int, std::pair<int, int>> cellCoordinates = {
5              {1, {0, 0}}, {2, {0, 1}}, {3, {0, 2}},
6              {4, {1, 0}}, {5, {1, 1}}, {6, {1, 2}},
7              {7, {2, 0}}, {8, {2, 1}}, {9, {2, 2}},
8              // Continúa con el resto de las celdas de la cuadrícula...
9              {10, {3, 0}}, {11, {3, 1}}, {12, {3, 2}},
{13, {4, 0}}, {14, {4, 1}}, {15, {4, 2}},
```

```

10          {16,{5, 0}}, {17,{5, 1}}, {18,{5, 2}},
11          {19,{6, 0}}, {20,{6, 1}}, {21,{6, 2}},
12          {22,{7, 0}}, {23,{7, 1}}, {24,{7, 2}}
13      };
14
15      std::cout << "Moviendo a lo largo de la ruta..." << std::endl;
16      std::cout << "La ruta es: " << std::endl;
17      for (int cell : route) {
18          std::cout << cell << " ";
19      }
20      std::cout << std::endl;
21      for (size_t i = 0; i < route.size() - 1; ++i) {
22          int currentCell = route[i];
23          int nextCell = route[i + 1];
24          std::cout << "Current cell: " << currentCell << " Next
25          cell: " << nextCell << endl;
26          auto currentCoord = cellCoordinates[currentCell];
27          auto nextCoord = cellCoordinates[nextCell];
28
29          // Calculamos la direcci\on en la que el robot debe
30          // moverse.
31          int deltaX = nextCoord.first - currentCoord.first;
32          int deltaY = nextCoord.second - currentCoord.second;
33          is_manual_mode = false;
34          std::cout << "Delta X: " << deltaX << " Delta Y: " <<
35          deltaY << endl;
36          if (deltaX == 0 && deltaY == 1) {
37              // Movimiento hacia la derecha.
38              stopMotors();
39              std::cout << "Girando hacia la derecha y avanzando..." <<
40              std::endl;
41              turnRight();
42              std::this_thread::sleep_for(std::chrono::milliseconds
43              (4500)); // Girar por 4.35 segundos.
44              stopMotors();
45              std::this_thread::sleep_for(std::chrono::milliseconds
46              (500)); // Pausa antes de avanzar.
47              moveForward();
48              std::this_thread::sleep_for(std::chrono::milliseconds
49              (5000)); // Avanzar por 5 segundos.
50              stopMotors();
51          } else if (deltaX == 0 && deltaY == -1) {
52              // Movimiento hacia la izquierda.
53              stopMotors();
54              std::cout << "Girando hacia la izquierda y avanzando
55              ..." << std::endl;
56              turnLeft();
57              std::this_thread::sleep_for(std::chrono::milliseconds
58              (4350)); // Girar por 4.35 segundos.

```

```

        endl;
    moveForward();
    std::this_thread::sleep_for(std::chrono::milliseconds
        (5000)); // Avanzar por 5 segundos.
    stopMotors();
} else if (deltaX == -1 && deltaY == 0) {
    // Movimiento hacia atrás (eje X negativo).
    std::cout << "Retrocediendo..." << std::endl;
    moveBackward();
    std::this_thread::sleep_for(std::chrono::milliseconds
        (5000)); // Retroceder por 5 segundos.
    stopMotors();
} else {
    std::cerr << "Movimiento no soportado entre las celdas
    : " << currentCell << " y " << nextCell << std::
    endl;
}
std::cout << "Movimiento completado." << std::endl;
stopMotors();
// Pausa entre movimientos para evitar sobrecargar los
// motores.
std::this_thread::sleep_for(std::chrono::milliseconds
    (1000));
}
}

```

5.2.6. Ajustes de código en función de las pruebas de carga 2

Luego de realizar las pruebas de carga 2, se identificaron algunos problemas en el código del robot, específicamente en el algoritmo de seguimiento de ruta. Se observó que el robot tenía dificultades para seguir la ruta de manera precisa, lo que resultaba en movimientos erráticos y desviaciones significativas de la trayectoria deseada. Para solucionar este problema, se realizaron los siguientes ajustes en el código del robot:

- Se mejoró el algoritmo de seguimiento de ruta para que el robot pudiera seguir la trayectoria de manera más precisa. Se ajustaron los parámetros de control de velocidad y dirección para minimizar las desviaciones y los errores en el seguimiento de la ruta.
- Se añadió un nuevo apartado gráfico en la interfaz de usuario para mostrar las dimensiones del robot en 2D, lo que permitió al operador visualizar la posición y orientación del robot en relación con la ruta planificada. Esto facilitó la supervisión y el control del robot durante la operación, y ayudó a prevenir colisiones y desviaciones no deseadas.

Estos ajustes permitieron mejorar el desempeño del robot y garantizar un seguimiento preciso de la ruta, lo que resultó en un funcionamiento más eficiente y fiable del sistema en general. Estos cambios fueron fundamentales para optimizar el rendimiento del robot y garantizar su correcto funcionamiento durante las pruebas de carga y las operaciones en entornos reales, y se pueden observar en el Listing 10.

Listing 10: Quinto ajuste de código

```

1      #include "CYdLidar.h"
2      #include <SFML/Graphics.hpp>
3      #include <opencv2/opencv.hpp>
4      #include <iostream>

```

```

5      #include <cstdlib>
6      #include <cstdio>
7      #include <pigpio.h>
8      #include <iostream>
9      #include <map>
10     #include <vector>
11     #include <atomic>
12     #include <thread>
13     #include <random>
14     #include <sys/socket.h>
15     #include <sys/un.h>
16     #include <unistd.h>
17     #include <cstring>
18     #include <pthread.h>
19     #include <vector>
20     #include <mutex>
21     #define PI 180.f
22
23     using namespace std;
24     using namespace ydlidar;
25
26     // Pines y configuración para los motores
27     const int PWM_PINS[] = {13, 19, 18, 12}; // Pines PWM para los
28     motores
29     const int DIR_PINS[] = {5, 6, 23, 24}; // Pines de dirección
30     para los motores
31     const int SENSOR_PROFUNDIDAD_PINS[] = {2, 3};
32     int frequency = 400; // Frecuencia inicial
33     const sf::Vector2f robotFixedPosition(110, 110);
34     std::atomic<bool> is_running(true);
35     std::atomic<bool> is_manual_mode(true);
36
37     void setMotorSpeed(int motor, int frequency) {
38         if (motor >= 0 && motor < 4) {
39             gpioSetPWMfrequency(PWM_PINS[motor], frequency);
40         }
41     }
42
43     void setMotorDirection(int motor, int direction) {
44         if (motor >= 0 && motor < 4) {
45             gpioWrite(DIR_PINS[motor], direction);
46         }
47     }
48
49     void stopMotors() {
50         for (int i = 0; i < 4; ++i) {
51             gpioPWM(PWM_PINS[i], 0);
52         }
53     }
54
55     void moveForward() {
56         for (int i = 0; i < 4; ++i) {
57             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
58             al 50%
59         }
60         setMotorDirection(0, 0); // Motor 1
61         setMotorDirection(2, 0); // Motor 3
62         setMotorDirection(1, 1); // Motor 2
63         setMotorDirection(3, 1); // Motor 4
64     }

```

```

62
63     void moveBackward() {
64         for (int i = 0; i < 4; ++i) {
65             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
66                                     al 50%
67         }
68         setMotorDirection(0, 1); // Motor 1
69         setMotorDirection(2, 1); // Motor 3
70         setMotorDirection(1, 0); // Motor 2
71         setMotorDirection(3, 0); // Motor 4
72     }
73
74     void turnLeft() {
75         std::cout << "Turning left" << std::endl;
76         for (int i = 0; i < 4; ++i) {
77             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
78                                     al 50%
79         }
80         setMotorDirection(0, 1); // Motor 1
81         setMotorDirection(2, 1); // Motor 3
82         setMotorDirection(1, 1); // Motor 2
83         setMotorDirection(3, 1); // Motor 4
84     }
85
86     void turnRight() {
87         std::cout << "Turning right" << std::endl;
88         for (int i = 0; i < 4; ++i) {
89             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
90                                     al 50%
91         }
92         setMotorDirection(0, 0); // Motor 1
93         setMotorDirection(2, 0); // Motor 3
94         setMotorDirection(1, 0); // Motor 2
95         setMotorDirection(3, 0); // Motor 4
96     }
97
98     void increaseSpeed() {
99         if (frequency < 2000) {
100             frequency += 100;
101             if (frequency > 2000) frequency = 2000;
102             for (int i = 0; i < 4; ++i) {
103                 setMotorSpeed(i, frequency);
104             }
105         }
106     }
107
108     void decreaseSpeed() {
109         if (frequency > 400) {
110             frequency -= 100;
111             if (frequency < 400) frequency = 400;
112             for (int i = 0; i < 4; ++i) {
113                 setMotorSpeed(i, frequency);
114             }
115         }
116     }
117
118     sf::Color getPointColor(float distance, float maxRange) {
119         float ratio = distance / maxRange;
120         return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color

```

```

        de rojo a verde
    }

120
121
122 std::vector<sf::Vector2f> routePoints;
123 bool isDrawing = false; // Bandera para rastrear si el raton esta
124 presionado
125
126
127
128 void drawLiDARPoints(sf::RenderWindow &window, const LaserScan &
129 lidarPoints, const sf::Vector2f &robotPosition, float scale,
130 float max_range) {
131     for (const auto& point : lidarPoints.points) {
132         float x = point.range * cos(point.angle);
133         float y = point.range * sin(point.angle);

134         float adjustedX = robotFixedPosition.x + (x * scale) -
135             robotPosition.x;
136         float adjustedY = robotFixedPosition.y - (y * scale) -
137             robotPosition.y; // Invertir Y para la pantalla
138
139         if (point.range > 0.05) { // Excluir puntos cercanos al
140             centro
141             sf::CircleShape lidarPoint(2); // Tamaño del punto
142             lidarPoint.setPosition(adjustedX, adjustedY);
143             lidarPoint.setFillColor(getPointColor(point.range,
144                                         max_range)); // Color basado en la distancia
145
146             window.draw(lidarPoint);
147         }
148     }
149 }
150
151 void handleMouseClick(const sf::Event &event, const sf:::
152 RectangleShape &minimap) {
153     // Verificar si el raton esta presionado
154     if (event.type == sf::Event::MouseButtonPressed && event.
155         mouseButton.button == sf::Mouse::Left) {
156         isDrawing = true; // Marcar que estamos dibujando
157
158         // Verificar si el raton fue soltado
159     } else if (event.type == sf::Event::MouseButtonReleased &&
160                 event.mouseButton.button == sf::Mouse::Left) {
161         isDrawing = false; // Dejar de dibujar
162     }
163
164     void updateRoute(const sf::RectangleShape &minimap, sf:::
165 RenderWindow &window) {
166         // Si el raton esta presionado, agregar puntos a la ruta
167         if (isDrawing) {
168             // Obtener la posicion actual del raton
169             sf::Vector2i mousePosition = sf::Mouse::getPosition(window

```

```

        );
167     sf::Vector2f clickPosition(static_cast<float>(
168         mousePosition.x), static_cast<float>(mousePosition.y))
169         ;
170
171         // Verificar si la posicion del raton esta dentro del
172         // minimapa
173         if (minimap.getGlobalBounds().contains(clickPosition)) {
174             sf::Vector2f relativePosition = clickPosition -
175                 minimap.getPosition();
176             routePoints.push_back(relativePosition);
177             std::cout << "Dibujando punto: (" << relativePosition.
178                 x << ", " << relativePosition.y << ")" << std::
179                 endl;
180         }
181     }
182 }
183
184
185 void drawRoute(sf::RenderWindow &window, const std::vector<sf::
186     Vector2f> &routePoints, const sf::Vector2f &robotPosition) {
187     if (routePoints.size() > 1) {
188         for (size_t i = 0; i < routePoints.size() - 1; ++i) {
189             // Dibujar las lneas de la ruta, ajustando las
190             // coordenadas en funcin de la posicin del robot
191             sf::Vertex line[] = {
192                 sf::Vertex(robotFixedPosition + (routePoints[i] -
193                     robotPosition), sf::Color::Red), // Ajustar
194                 posicin en funcin del robot
195                 sf::Vertex(robotFixedPosition + (routePoints[i] +
196                     1] - robotPosition), sf::Color::Red)
197             };
198             window.draw(line, 2, sf::Lines);
199         }
200     }
201 }
202
203
204 // Ajustar la referencia para que el norte sea -1.57 rad (sur en
205 // coordenadas cartesianas es 0 rad)
206 float adjustAngleForNorth(float angle) {
207     float adjustedAngle = angle - (-M_PI / 2); // Ajuste de 90 (
208     norte en -90)
209
210     // Asegurarse de que el angulo este dentro de los limites de -
211     PI a PI
212     if (adjustedAngle > M_PI) {
213         adjustedAngle -= 2 * M_PI;
214     } else if (adjustedAngle < -M_PI) {
215         adjustedAngle += 2 * M_PI;
216     }
217
218     return adjustedAngle;
219 }
220
221

```

```

212
213
214
215
216
217     void rotateTowardsPoint(float currentAngle, float targetAngle) {
218         float angleDifference = targetAngle - currentAngle;
219
220         // Asegurate de que el angulo este entre -PI y PI
221         if (angleDifference > M_PI) {
222             angleDifference -= 2 * M_PI;
223         } else if (angleDifference < -M_PI) {
224             angleDifference += 2 * M_PI;
225         }
226
227         // Gira en la direccion correcta
228         if (angleDifference > 0) {
229             turnRight(); // Girar hacia la derecha
230         } else {
231             turnLeft(); // Girar hacia la izquierda
232         }
233
234         // Esperar hasta que el robot este alineado
235         std::this_thread::sleep_for(std::chrono::milliseconds(
236             static_cast<int>(std::fabs(angleDifference) * 1000)));
237
238         stopMotors();
239
240
241
242
243
244
245     void moveTowardsPoint(float distance) {
246         moveForward(); // Mover el robot hacia adelante
247
248         // Esperar un tiempo proporcional a la distancia
249         std::this_thread::sleep_for(std::chrono::milliseconds(
250             static_cast<int>(distance * 1000)));
251
252         stopMotors(); // Detener el robot
253
254
255
256
257     void moveTo(sf::Vector2f point, sf::Vector2f &robotPosition, float
258                 &currentAngle) {
259         // Calcular la diferencia en X e Y
260         float deltaX = point.x - robotPosition.x;
261         float deltaY = robotPosition.y - point.y; // Invertir la Y
262             para la pantalla
263
264         // Calcular la distancia al punto
265         float distanceToPoint = sqrt(deltaX * deltaX + deltaY * deltaY
266             );
267
268         // Si la distancia es mayor a un umbral, mover el robot
269         const float threshold = 5.0f; // Tolerancia para considerar

```

```

        que el punto ha sido alcanzado
267     if (distanceToPoint > threshold) {
268         // Calcular el angulo hacia el punto de destino
269         float targetAngle = atan2(deltaY, deltaX);
270         // Ajustar el angulo para que el norte sea -90
271         targetAngle = adjustAngleForNorth(targetAngle);
272
273         // Girar hacia el angulo correcto
274         rotateTowardsPoint(currentAngle, targetAngle); // Funcin
275             para girar hacia el ngulo
276
277         // Mover hacia el punto
278         moveTowardsPoint(distanceToPoint);
279
280         // Actualizar la posicion y el ngulo del robot
281         robotPosition = point;
282         currentAngle = targetAngle;
283     }
284
285
286
287     void followRoute(sf::RenderWindow &window, std::vector<sf::
288     Vector2f> &routePoints, const LaserScan &lidarPoints, const sf
289     ::Vector2f &minimapPosition, float max_range) {
290         sf::Vector2f robotPosition(110, 110); // Posicin inicial del
291             robot en el minimapa
292         float currentAngle = -M_PI / 2; // ngulo inicial del robot (
293             norte en -90)
294
295         while (!routePoints.empty()) {
296             sf::Vector2f targetPoint = routePoints.front();
297
298             moveTo(targetPoint, robotPosition, currentAngle);
299
300             routePoints.erase(routePoints.begin());
301
302             drawRoute(window, routePoints, minimapPosition);
303             drawLiDARPoints(window, lidarPoints, robotPosition, 25.0f,
304                 max_range);
305         }
306     }
307
308
309
310
311
312     int contadorObstaculoTotal = 0;
313     int obstaculoHola = 0;
314     int obsRelativo = 0;
315     int idTipoObstaculo = 0;
316     void randomMovement(CYdLidar &laser) {
317         std::random_device rd;
318         std::mt19937 gen(rd());
319         std::discrete_distribution<> dist({0, 5}); // Distribucion para

```

```

la probabilidad de movimiento

320
321     const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
322             grados en radaianes
323     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
324             grados en radianes
325     const float DETECTION_RADIUS = 0.25f; // 35 cm
326
327     std::vector<float> previousScanPoints;
328
329     while (is_running) {
330         if (!is_manual_mode) {
331             LaserScan scan;
332             if (laser.doProcessSimple(scan)) {
333                 bool obstacle_detected = false;
334                 std::vector<float> currentScanPoints;
335
336                 for (const auto &point : scan.points) {
337                     // Convertir el ngulo del punto al ngulo
338                     // relativo al "sur" del robot
339                     float adjusted_angle = point.angle + PI;
340                     //std::cout << "Er " << point.range << std::endl;
341
342                     // Verificar si el punto est dentro del rango
343                     // frontal de 30
344                     //std::cout << point.angle << std::endl;
345                     if (point.range > 0 && point.range < 0.30 && (
346                         point.angle <= -0.5235f && point.angle
347                         >= -2.617f)) { // Norte
348                         obstacle_detected = true;
349                         currentScanPoints.push_back(point.range);
350                         idTipoObstaculo = 1;
351                         std::cout << "Obstacle distance N: " << (
352                             float)point.range << " Y en el angulo
353                             "<<point.angle << std::endl;
354                         break;
355                     } else if (point.range > 0 && point.range <
356                         0.25 && (point.angle <= 0.872f && point.
357                         angle >= -0.5235f)){ // Este
358                         obstacle_detected = true;
359                         currentScanPoints.push_back(point.range);

```

```

360                     currentScanPoints.push_back(point.range);
361                     idTipoObstaculo = 4;
362                     std::cout << "Obstacle distance 0:" << (
363                         float)point.range << " Y en el angulo
364                         "<<point.angle << std::endl;
365                         break;
366                     }
367                 }
368
369             if (obstacle_detected) {
370                 // Si detecta un obstaculo, retrocede por 4
371                 // segundos
372                 contadorObstaculoTotal++;
373                 std::cout << "Obs: " << contadorObstaculoTotal
374                     << std::endl;
375                 //obsRelativo++;
376                 stopMotors();
377                 //Luego gira aleatoriamente a la izquierda o
378                 //derecha
379                 switch(idTipoObstaculo) {
380                     case 1: {
381                         int turn = dist(gen) % 2;
382                         moveBackward();
383                         std::this_thread::sleep_for(std::
384                             chrono::seconds(5));
385                         if (turn == 0) {
386                             turnLeft();
387                             std::this_thread::sleep_for(std::
388                                 chrono::seconds(5));
389                         } else {
390                             turnRight();
391                             std::this_thread::sleep_for(std::
392                                 chrono::seconds(5));
393                         }
394                         std::this_thread::sleep_for(std::
395                             chrono::seconds(2));
396                         stopMotors();
397                     }
398                     break;
399
400                     case 2:
401                         turnLeft();
402                         std::this_thread::sleep_for(std::
403                             chrono::seconds(5));
404                         std::this_thread::sleep_for(std::
405                             chrono::seconds(2));
406                         stopMotors();
407                     break;
408
409                     case 3:// este es el 4 y el de abajo el 3
410                         moveForward();
411                         std::this_thread::sleep_for(std::
412                             chrono::seconds(2));
413                         stopMotors();
414
415                     break;
416
417                     case 4:

```

```

408                     turnRight();
409                     std::this_thread::sleep_for(std::
410                         chrono::seconds(7));
411                     stopMotors();
412                     break;
413
414             default: {
415                 int turn = dist(gen) % 2;
416                 if (turn == 0) {
417                     turnLeft();
418                     std::this_thread::sleep_for(std::
419                         chrono::seconds(5));
420                 } else {
421                     turnRight();
422                     std::this_thread::sleep_for(std::
423                         chrono::seconds(5));
424                 }
425             }
426         }
427
428
429     if(!previousScanPoints.empty() &&
430         previousScanPoints.size() ==
431         currentScanPoints.size()){
432         bool mismoObs = true;
433         for(size_t i = 0; i <
434             currentScanPoints.size(); ++i){
435             if(fabs(currentScanPoints[i] -
436                 previousScanPoints[i]) > 0.05)
437             {
438                 mismoObs = false;
439                 break;
440             }
441         }
442     }
443
444     previousScanPoints = currentScanPoints;
445
446     if((obsRelativo > 3)){
447         //obsRelativo = 0;
448         moveBackward();
449         std::this_thread::sleep_for(std::chrono::
450             seconds(3));
451         stopMotors();
452
453         switch(idTipoObstaculo) {
454             case 1: {
455                 int turn = dist(gen) % 2;
456                 moveBackward();
457                 std::this_thread::sleep_for(std::

```

```

        chrono::seconds(5));
458     if (turn == 0) {
459         turnLeft();
460         std::this_thread::sleep_for(
461             std::chrono::seconds(5));
462     } else {
463         turnRight();
464         std::this_thread::sleep_for(
465             std::chrono::seconds(5));
466     }
467     std::this_thread::sleep_for(std::
468         chrono::seconds(2));
469     stopMotors();
470 }
471 break;

case 2:
472     turnLeft();
473     std::this_thread::sleep_for(std::
474         chrono::seconds(5));
475     std::this_thread::sleep_for(std::
476         chrono::seconds(2));
477     stopMotors();
478 break;

case 3:
479     turnRight();
480     std::this_thread::sleep_for(std::
481         chrono::seconds(7));
482     stopMotors();
483 break;

case 4:
484     moveForward();
485     std::this_thread::sleep_for(std::
486         chrono::seconds(2));
487     stopMotors();
488 break;

default: {
489     int turn = dist(gen) % 2;
490     if (turn == 0) {
491         turnLeft();
492         std::this_thread::sleep_for(
493             std::chrono::seconds(5));
494     } else {
495         turnRight();
496         std::this_thread::sleep_for(
497             std::chrono::seconds(5));
498     }
499     std::this_thread::sleep_for(std::
500         chrono::seconds(2));
501     stopMotors();
502 }
503 break;
504 }
505 }

```

```

507     } else {
508         moveForward();
509         // Si no hay obstculo, elige un movimiento
510         // aleatorio
511         //
512         //int move = dist(gen);
513         //switch (move) {
514             // case 0: moveBackward(); break;
515             // case 1: turnRight(); break;
516             // case 2: turnLeft(); break;
517             // case 3: moveForward(); break;
518         //}
519     }
520
521     std::this_thread::sleep_for(std::chrono::milliseconds
522         (500));
523     //stopMotors();
524     std::this_thread::sleep_for(std::chrono::milliseconds(100)
525         );
526 }
527
528
529 // Definir las dimensiones del robot en el minimapa
530 const float frontDistance = 15.0f;    // cm hacia adelante
531 const float backDistance = 35.0f;    // cm hacia atrs
532 const float rightDistance = 21.0f;   // cm hacia la derecha
533 const float leftDistance = 21.0f;    // cm hacia la izquierda
534
535 // Escalar las distancias al minimapa (considera ajustar el
536 // factor de escala segn lo necesites)
537 float scale = 0.4f; // Cambia este valor si necesitas ajustar
538 // el tamao del robot en el minimapa
539
540
541 void drawRobot(sf::RenderWindow &window, const sf::Vector2f &
542     robotPosition, const sf::RectangleShape &minimap) {
543
544     float width = (leftDistance + rightDistance) * scale;
545     float height = (frontDistance + backDistance) * scale;
546
547     // Dibujar el rectngulo representando al robot
548     sf::RectangleShape robotShape(sf::Vector2f(width, height));
549     robotShape.setFillColor(sf::Color(100, 100, 255, 150)); // // Color azul claro con algo de transparencia
550
551     // Ajustar el origen del rectngulo al punto donde est el LiDAR
552     // , es decir, al centro del robot
553     robotShape.setOrigin(leftDistance * scale, backDistance *
554         scale);
555
556     // Establecer la posicin del rectngulo del robot en el
557     // minimapa, centrado en la posicin del LiDAR
558     robotShape.setPosition(105, 105); // Usamos la misma posicin
559     // del LiDAR para centrar el robot
560
561     window.draw(robotShape);

```

```

556     }
557
558
559
560
561     void moveAlongRoute(const std::vector<int> &route) {
562         // Las celdas est\'an numeradas de 1 a N como en la imagen
563         // proporcionada.
564         std::map<int, std::pair<int, int>> cellCoordinates = {
565             {1, {0, 0}}, {2, {0, 1}}, {3, {0, 2}},
566             {4, {1, 0}}, {5, {1, 1}}, {6, {1, 2}},
567             {7, {2, 0}}, {8, {2, 1}}, {9, {2, 2}},
568             // Contin\'ua con el resto de las celdas de la cuadr\'atica...
569             {10,{3, 0}}, {11,{3, 1}}, {12,{3, 2}},
570             {13,{4, 0}}, {14,{4, 1}}, {15,{4, 2}},
571             {16,{5, 0}}, {17,{5, 1}}, {18,{5, 2}},
572             {19,{6, 0}}, {20,{6, 1}}, {21,{6, 2}},
573             {22,{7, 0}}, {23,{7, 1}}, {24,{7, 2}}
574         };
575
576         std::cout << "Moviendo a lo largo de la ruta..." << std::endl;
577         std::cout << "La ruta es: " << std::endl;
578         for (int cell : route) {
579             std::cout << cell << " ";
580         }
581         std::cout << std::endl;
582         for (size_t i = 0; i < route.size() - 1; ++i) {
583             int currentCell = route[i];
584             int nextCell = route[i + 1];
585             std::cout << "Current cell: " << currentCell << " Next
586             cell: " << nextCell << endl;
587             auto currentCoord = cellCoordinates[currentCell];
588             auto nextCoord = cellCoordinates[nextCell];
589
590             // Calculamos la direcci\'on en la que el robot debe
591             // moverse.
592             int deltaX = nextCoord.first - currentCoord.first;
593             int deltaY = nextCoord.second - currentCoord.second;
594             is_manual_mode = false;
595             std::cout << "Delta X: " << deltaX << " Delta Y: " <<
596             deltaY << endl;
597             if (deltaX == 0 && deltaY == 1) {
598                 // Movimiento hacia la derecha.
599                 stopMotors();
600                 std::cout << "Girando hacia la derecha y avanzando..." <<
601                 std::endl;
602                 turnRight();
603                 std::this_thread::sleep_for(std::chrono::milliseconds
604                     (4500)); // Girar por 4.35 segundos.
605                 stopMotors();
606                 std::this_thread::sleep_for(std::chrono::milliseconds
607                     (500)); // Pausa antes de avanzar.
608                 moveForward();
609                 std::this_thread::sleep_for(std::chrono::milliseconds
610                     (5000)); // Avanzar por 5 segundos.
611                 stopMotors();
612             } else if (deltaX == 0 && deltaY == -1) {
613                 // Movimiento hacia la izquierda.
614                 stopMotors();
615             }
616         }
617     }

```

```

607         std::cout << "Girando hacia la izquierda y avanzando
608             ..." << std::endl;
609         turnLeft();
610         std::this_thread::sleep_for(std::chrono::milliseconds
611             (4350)); // Girar por 4.35 segundos.
612         stopMotors();
613         std::this_thread::sleep_for(std::chrono::milliseconds
614             (500)); // Pausa antes de avanzar.
615         moveForward();
616         std::this_thread::sleep_for(std::chrono::milliseconds
617             (5000)); // Avanzar por 5 segundos.
618         stopMotors();
619     } else if (deltaX == 1 && deltaY == 0) {
620         // Movimiento hacia adelante (eje X positivo).
621         stopMotors();
622         std::cout << "Avanzando hacia adelante..." << std::
623             endl;
624         moveForward();
625         std::this_thread::sleep_for(std::chrono::milliseconds
626             (5000)); // Avanzar por 5 segundos.
627         stopMotors();
628     } else if (deltaX == -1 && deltaY == 0) {
629         // Movimiento hacia atr\as (eje X negativo).
630         std::cout << "Retrocediendo..." << std::endl;
631         moveBackward();
632         std::this_thread::sleep_for(std::chrono::milliseconds
633             (5000)); // Retroceder por 5 segundos.
634         stopMotors();
635     } else {
636         std::cerr << "Movimiento no soportado entre las celdas
637             : " << currentCell << " y " << nextCell << std::
638             endl;
639     }
640     std::cout << "Movimiento completado." << std::endl;
641     stopMotors();
642     // Pausa entre movimientos para evitar sobrecargar los
643     // motores.
644     std::this_thread::sleep_for(std::chrono::milliseconds
645             (1000));
646 }
647
648
649
650
651 void drawGrid(sf::RenderWindow &window, const sf::RectangleShape &
minimap) {
652     const float cellSizeCm = 50.0f; // Tama\~no de cada celda en
653         cent\`imetros
654     const float scaleFactor = 10.0f; // Relaci\on 1 cm = 10 p\
655         ixelos en el minimapa
656     const float cellSizePx = cellSizeCm * scaleFactor; // Tama\~no de
657         cada celda en p\`ixelos
658
659     // Obtener la posici\on y tama\~no del minimapa

```

```

652     sf::Vector2f minimapPosition = minimap.getPosition();
653     sf::Vector2f minimapSize = minimap.getSize();
654
655     // Calcular el origen del grid para mantenerlo fijo respecto
656     // al robot centrado en (105, 105)
657     float originX = minimapPosition.x - fmod(minimapPosition.x +
658         105.0f, cellSizePx);
659     float originY = minimapPosition.y - fmod(minimapPosition.y +
660         105.0f, cellSizePx);
661
662     // Dibujar las l\'ineas verticales de la cuadr\'icula
663     for (float x = originX; x <= minimapPosition.x + minimapSize.x
664         ; x += cellSizePx) {
665         sf::Vertex verticalLine[] = {
666             sf::Vertex(sf::Vector2f(x, minimapPosition.y), sf::
667                 Color(150, 150, 150, 200)),
668             sf::Vertex(sf::Vector2f(x, minimapPosition.y +
669                 minimapSize.y), sf::Color(150, 150, 150, 200))
670         };
671         window.draw(verticalLine, 2, sf::Lines);
672     }
673
674     // Dibujar las l\'ineas horizontales de la cuadr\'icula
675     for (float y = originY; y <= minimapPosition.y + minimapSize.y
676         ; y += cellSizePx) {
677         sf::Vertex horizontalLine[] = {
678             sf::Vertex(sf::Vector2f(minimapPosition.x, y), sf::
679                 Color(150, 150, 150, 200)),
680             sf::Vertex(sf::Vector2f(minimapPosition.x +
681                 minimapSize.x, y), sf::Color(150, 150, 150, 200))
682         };
683         window.draw(horizontalLine, 2, sf::Lines);
684     }
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
int main() {
    // Inicializar pigpio
    if (gpioInitialise() < 0) {
        std::cerr << "Error: No se pudo inicializar pigpio." <-
            std::endl;
        return -1;
    }

    // Configurar pines de direcci\'on como salida
    for (int i = 0; i < 4; ++i) {
        gpioSetMode(DIR_PINS[i], PI_OUTPUT);
        gpioSetMode(PWM_PINS[i], PI_OUTPUT);
        setMotorSpeed(i, frequency); // Inicializar PWM con
            frecuencia inicial
    }

    // Asegurate de establecer XDG_RUNTIME_DIR
    if (getenv("XDG_RUNTIME_DIR") == nullptr) {
        setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
}

```

```

701 }
702
703 // Ejecuta libcamera-vid en un proceso separado y captura la
704 // salida en YUV, sin previsualizacn
705 FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
706     nopreview -o -", "r");
707 if (!pipe) {
708     std::cerr << "Error: No se pudo ejecutar libcamera-vid."
709     << std::endl;
710     return -1;
711 }
712
713 // Configura la ventana SFML
714 sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
715     Visualization with LiDAR");
716 sf::Texture cameraTexture;
717 sf::Sprite cameraSprite;
718
719 // Buffer para leer los datos de video
720 const int width = 640;
721 const int height = 480;
722 std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
723     el tamao del buffer para YUV420
724
725 cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
726     data());
727 cv::Mat rgbImage(height, width, CV_8UC3);
728
729 std::string port;
730 ydlidar::os_init();
731
732 // Obtener los puertos disponibles de LiDAR
733 std::map<std::string, std::string> ports = ydlidar::
734     lidarPortList();
735 if (ports.size() > 1) {
736     auto it = ports.begin();
737     std::advance(it, 1); // Selecciona el segundo puerto
738     disponible
739     port = it->second;
740 } else if (ports.size() == 1) {
741     port = ports.begin()->second;
742 } else {
743     std::cerr << "No se detect ningn LiDAR. Verifica la
744     conexin." << std::endl;
745     return -1;
746 }
747
748 // Configuracin del LiDAR
749 int baudrate = 115200;
750 std::cout << "Baudrate: " << baudrate << std::endl;
751
752 CYdLidar laser;
753 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
754     ());
755 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
756     int));
757
758 bool isSingleChannel = true;
759 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
760     sizeof(bool));

```

```

749
750     float max_range = 8.0f;
751     float min_range = 0.1f;
752     float max_angle = 180.0f;
753     float min_angle = -180.0f;
754     float frequency = 8.0f;
755
756     laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float))
757         );
758     laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float))
759         );
760     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float))
761         );
762     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float))
763         );
764     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
765         float));
766
767     // Inicializar LiDAR
768     if (!laser.initialize()) {
769         std::cerr << "Error al inicializar el LiDAR." << std::endl
770             ;
771         return -1;
772     }
773
774     // Iniciar el escaneo
775     if (!laser.turnOn()) {
776         std::cerr << "Error al encender el LiDAR." << std::endl;
777         return -1;
778     }
779
780     // Thread para movimiento aleatorio
781     std::thread randomMoveThread(randomMovement, std::ref(laser));
782
783
784     while (window.isOpen()) {
785         // Crear un minimapa para el LiDAR
786         sf::RectangleShape minimap(sf::Vector2f(200, 200));
787         minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo semitransparente
788         minimap.setPosition(10, 10); // Esquina superior izquierda
789
790
791         // Leer los datos del video desde la tubera
792         size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
793             pipe);
794         if (bytesRead != buffer.size()) {
795             std::cerr << "Error: No se pudo leer suficientes datos
796                 de video." << std::endl;
797             continue;
798         }
799
800         // Convertir YUV420 a RGB
801         cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
802
803         // Convertir a RGBA añadiendo un canal alfa
804         cv::Mat frame_rgba;
805         cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
806
807         // Actualizar la textura de la cámara con los datos del
808         frame

```

```

799     if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows
800         )) {
801         std::cerr << "Error: No se pudo crear la textura." <<
802             std::endl;
803         continue;
804     }
805     cameraTexture.update(frame_rgba.ptr());
806
807     cameraSprite.setTexture(cameraTexture);
808     cameraSprite.setScale(
809         window.getSize().x / static_cast<float>(cameraTexture.
810             getSize().x),
811         window.getSize().y / static_cast<float>(cameraTexture.
812             getSize().y)
813     );
814
815     // Actualizar la ruta en el minimapa
816
817     window.clear();
818     window.draw(cameraSprite);
819
820
821     window.draw(minimap);
822     drawGrid(window, minimap);
823     updateRoute(minimap, window);
824     // Dibujar la ruta en el minimapa
825     drawRoute(window, routePoints, minimap.getPosition());
826     // Dibujar el centro del LiDAR (color azul)
827     drawRobot(window, robotFixedPosition, minimap);
828
829
830     // Calcular el centro del rectngulo del robot para dibujar
831     // la lnea que indica el norte
832     sf::Vector2f robotCenter = minimap.getPosition() +
833         robotFixedPosition;
834
835     // Dibujar la lnea hacia el norte (ajustada al centro del
836     // rectngulo)
837     sf::Vertex line[] =
838     {
839         sf::Vertex(sf::Vector2f(105, 105), sf::Color::Black),
840             // Posicin inicial de la lnea
841         sf::Vertex(sf::Vector2f(105, 160), sf::Color::Black)
842             // Posicin final hacia arriba
843     };
844
845     // Dibujar la lnea que representa el norte
846     window.draw(line, 2, sf::Lines);
847
848
849     LaserScan scan;
850     if (laser.doProcessSimple(scan)) {
851         drawLiDARPoints(window, scan, minimap.getPosition(),
852             25.0f, max_range);
853     } else {
854         std::cerr << "No se pudieron obtener los datos del
855             LiDAR." << std::endl;
856     }

```

```

848
849     sf::Event event;
850     while (window.pollEvent(event)) {
851         if (event.type == sf::Event::Closed) {
852             window.close();
853         }
854
855         if(event.type == sf::Event::MouseButtonPressed){
856             std::cout << "Mouse pressed" << std::endl;
857             handleMouseClick(event, minimap);
858         }else if(event.type == sf::Event::MouseButtonReleased)
859         {
860             std::cout << "Mouse released" << std::endl;
861             handleMouseClick(event, minimap);
862             isDrawing = false;
863         }
864
865
866
867         if (event.type == sf::Event::KeyPressed) {
868             switch (event.key.code) {
869                 case sf::Keyboard::W:
870                     is_manual_mode = true;
871                     std::cout << "W" << std::endl;
872                     moveForward();
873                     break;
874                 case sf::Keyboard::S:
875                     is_manual_mode = true;
876                     std::cout << "S" << std::endl;
877                     moveBackward();
878                     break;
879                 case sf::Keyboard::A:
880                     is_manual_mode = true;
881                     std::cout << "A" << std::endl;
882                     turnLeft();
883                     break;
884                 case sf::Keyboard::D:
885                     is_manual_mode = true;
886                     std::cout << "D" << std::endl;
887                     turnRight();
888                     break;
889                 case sf::Keyboard::V:
890                     is_manual_mode = true;
891                     std::cout << "V" << std::endl;
892                     increaseSpeed();
893                     break;
894                 case sf::Keyboard::B:
895                     is_manual_mode = true;
896                     std::cout << "B" << std::endl;
897                     decreaseSpeed();
898                     break;
899                 case sf::Keyboard::Space:
900                     is_manual_mode = true;
901                     std::cout << "Space" << std::endl;
902                     stopMotors();
903                     break;
904                 case sf::Keyboard::R: {
905                     is_manual_mode = false;
906                     std::vector<int> route = {1, 2, 5, 8, 11,

```

```

14, 13, 16};
907 moveAlongRoute(route);
908 //std::thread routeThread(followRoute, std
909 //::ref(window), std::ref(routePoints),
910 //std::ref(scan), minimap.getPosition(),
911 //max_range);
912 //routeThread.detach();
913 break;
914 }
915 case sf::Keyboard::K:
916 is_manual_mode = false;
917 std::cout << "K" << std::endl;
918 break;
919 case sf::Keyboard::M:
920 is_manual_mode = true;
921 std::cout << "M" << std::endl;
922 break;
923 default:
924 break;
925 }
926 }
927 window.display();
928 }
929
930 // Detener el escaneo del LiDAR
931 laser.turnOff();
932 laser.disconnecting();
933
934 // Cierra la tubera, detiene los motores y apaga el robot
935 pclose(pipe);
936 stopMotors();
937 gpioTerminate();
938
939 is_running = false;
940 randomMoveThread.join();
941
942 return 0;
943 }

```

6. Pruebas del sistema

En esta sección se describen y documentan las diferentes pruebas realizadas al sistema desarrollado. Estas pruebas se realizaron con el objetivo de verificar el correcto funcionamiento del sistema y de identificar posibles errores o fallas en el código. Las pruebas se realizaron en diferentes etapas del desarrollo del sistema, desde la implementación de los módulos hasta la integración de los mismos. Primero para cada uno de los productos desarrollados, el simulador del Physarum Polycephalum y el robot, se describen como son las pruebas que se realizaron y los resultados obtenidos. Posteriormente se describen las pruebas correspondientes para cada uno de los productos.

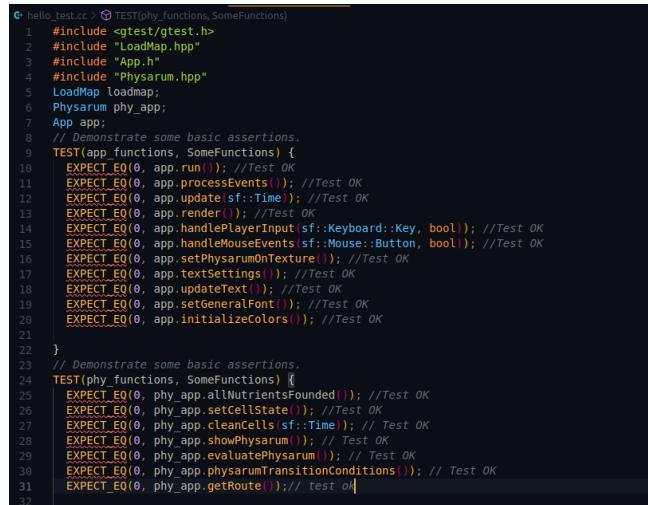
6.1. Pruebas unitarias del algoritmo en simulación de mapas 1

Para la realización de un programa, es necesario realizar las pruebas unitarias correspondientes para comprobar el funcionamiento de cada una de las partes involucradas con el programa.

En la realización de las pruebas unitarias, se utilizó Google Test, el cual es un framework de Google que funciona para el lenguaje de programación C++, que es el lenguaje utilizado para el desarrollo del simulador del Physarum. Es importante recordar que las pruebas unitarias son las que se encargan de verificar que cada una de las partes que componen a un software funcionen bien una a una, por lo que el mayor uso que se le dió al framework es la realización de las pruebas unitarias al código del simulador del Physarum, verificando cada una de las funciones que lo componen y demás componentes del software.

Principalmente, fue probado la implementación del algoritmo, el cual contiene algunas funciones importantes y algunas otras que devuelven valores importantes para la correcta implementación del algoritmo.

Algunas de las pruebas unitarias que fueron aplicadas se pueden ver en la Figura 45.



```
hello_test.c:1 #include <gtest/gtest.h>
hello_test.c:2 #include "LoadMap.hpp"
hello_test.c:3 #include "App.h"
hello_test.c:4 #include "Physarum.hpp"
hello_test.c:5 LoadMap loadmap;
hello_test.c:6 Physarum phy_app;
hello_test.c:7 App app;
hello_test.c:8 // Demonstrate some basic assertions.
hello_test.c:9 TEST(app_functions, Somefunctions) {
hello_test.c:10    EXPECT_EQ(0, app.run()); //Test OK
hello_test.c:11    EXPECT_EQ(0, app.processEvents()); //Test OK
hello_test.c:12    EXPECT_EQ(0, app.update(sf::Time)); //Test OK
hello_test.c:13    EXPECT_EQ(0, app.render()); //Test OK
hello_test.c:14    EXPECT_EQ(0, app.handlePlayerInput(sf::Keyboard::Key, bool)); //Test OK
hello_test.c:15    EXPECT_EQ(0, app.handleMouseEvents(sf::Mouse::Button, bool)); //Test OK
hello_test.c:16    EXPECT_EQ(0, app.setPhysarumOnTexture()); //Test OK
hello_test.c:17    EXPECT_EQ(0, app.textSettings()); //Test OK
hello_test.c:18    EXPECT_EQ(0, app.updateText()); //Test OK
hello_test.c:19    EXPECT_EQ(0, app.setGeneralFont()); //Test OK
hello_test.c:20    EXPECT_EQ(0, app.initializeColors()); //Test OK
hello_test.c:21
hello_test.c:22 }
hello_test.c:23 // Demonstrate some basic assertions.
hello_test.c:24 TEST(phy_functions, Somefunctions) {
hello_test.c:25    EXPECT_EQ(0, phy_app.allNutrientsFounded()); //Test OK
hello_test.c:26    EXPECT_EQ(0, phy_app.setCellState()); //Test OK
hello_test.c:27    EXPECT_EQ(0, phy_app.cleanCells(sf::Time)); // Test OK
hello_test.c:28    EXPECT_EQ(0, phy_app.showPhysarum()); // Test OK
hello_test.c:29    EXPECT_EQ(0, phy_app.evaluatePhysarum()); // Test OK
hello_test.c:30    EXPECT_EQ(0, phy_app.physarumTransitionConditions()); // Test OK
hello_test.c:31    EXPECT_EQ(0, phy_app.getRoute()); // test ok
hello_test.c:32 }
```

Figura 45: Prueba unitaria 1

Como se puede ver, gran parte de las pruebas realizadas al algoritmo fueron satisfactorias a este punto, por lo que podemos ver que a pesar de que la implementación fue en su mayoría correcta, hay una pequeña ventana de oportunidad para la mejora y corrección de la implementación del algoritmo en determinadas circunstancias, donde a pesar de que casi toda la implementación no está relacionada a una entrada del usuario, sino a partes involucradas en la realización de la simulación, entonces hay muy pocos casos en los que una parte de este mismo falle, ya que, debido a la implementación de las reglas, la manera en la que funciona un autómata celular y del propio funcionamiento del algoritmo en sí, no es permitido que algo falle ya que esto alteraría completamente el resultado final del algoritmo, realizando cálculos erróneos en la simulación y errando el objetivo del algoritmo.

6.2. Evaluación de desempeño del software en pruebas iniciales 1

El apartado de desempeño del software es complicado de medir, debido a que el único caso en el que se puede observar si hay una mejora o no es con el tiempo en el cual se realiza la

simulación.

Este tiempo puede variar en cada una de las ejecuciones del programa sin importar si las condiciones iniciales son las mismas, debido a que existe un grado de aleatoriedad inherente al algoritmo, por lo que cada una de las rutas que pueden llegar a ser obtenidas por el simulador varían en cada una de las ejecuciones, como se puede comprobar al observar las Figuras 46 - 50.

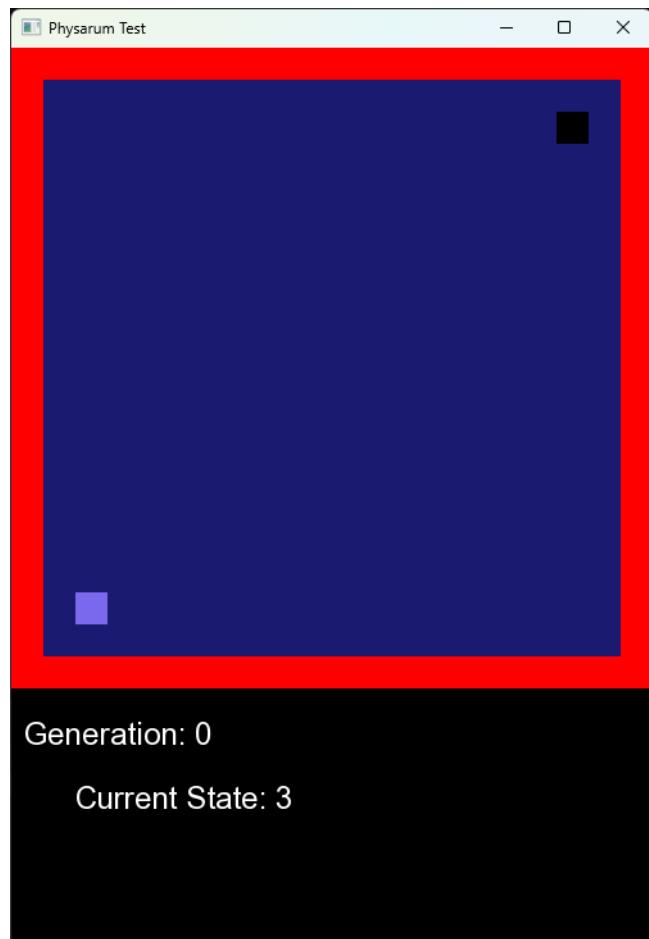


Figura 46: Configuración Base

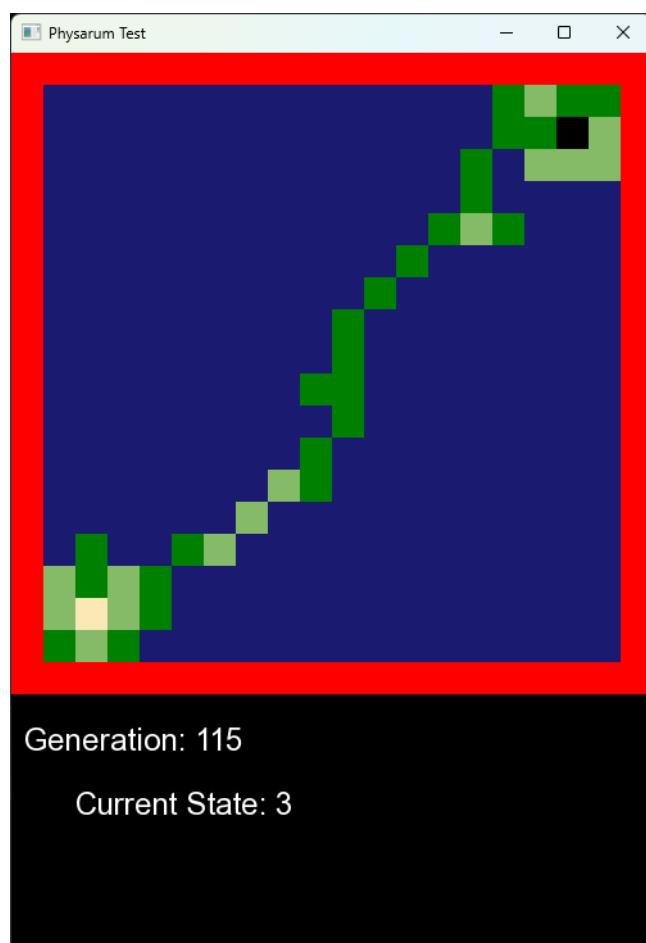


Figura 47: Primera Ruta Obtenida

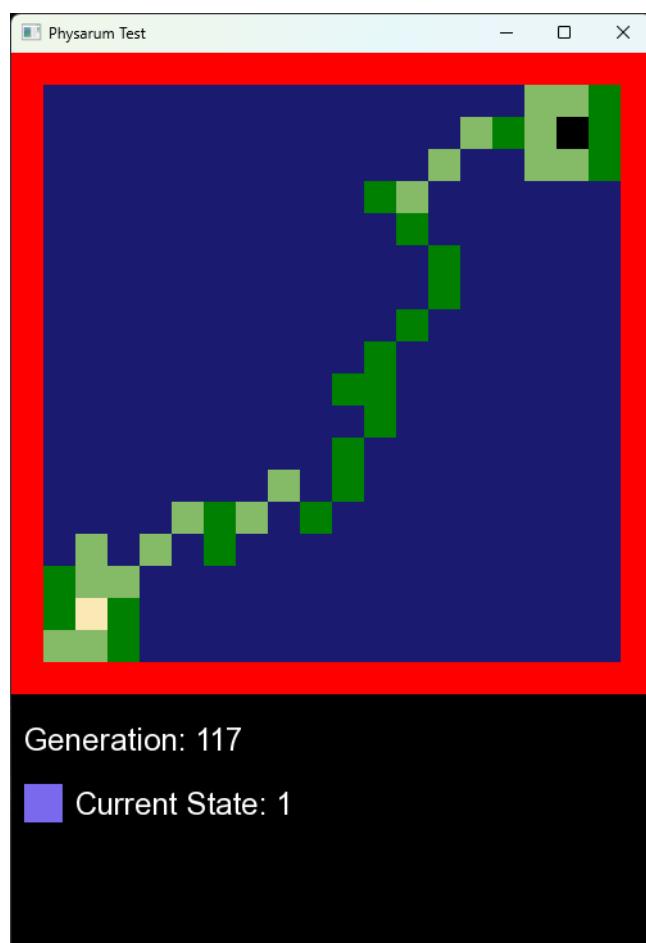


Figura 48: Segunda Ruta Obtenida

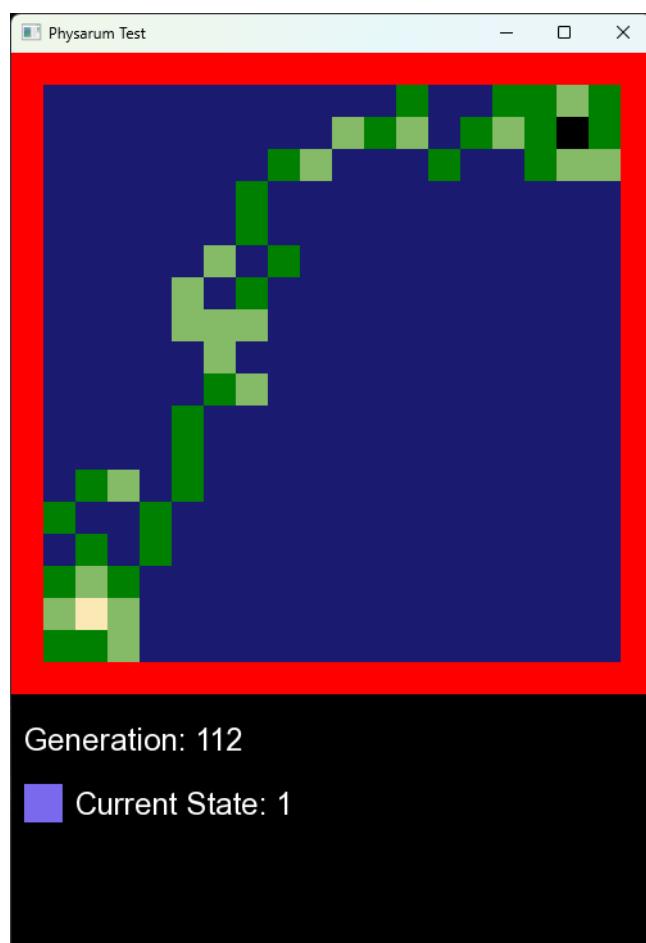


Figura 49: Tercera Ruta Obtenida

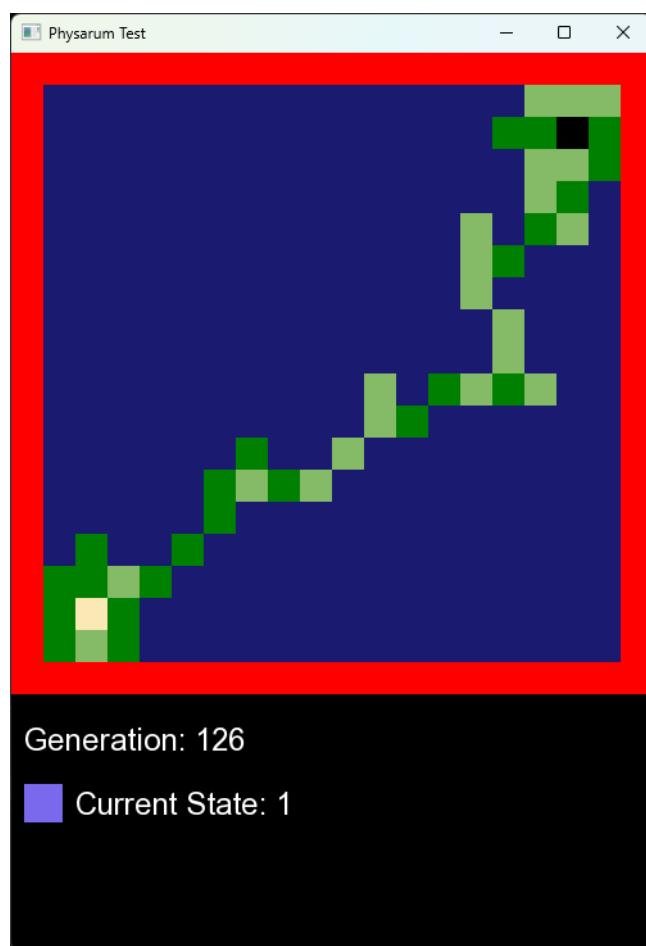


Figura 50: Cuarta Ruta Obtenida

Como se puede comprobar, a pesar de que el estado inicial sea exactamente el mismo, la ruta obtenida a partir del simulador cambia en cada uno de los casos, por lo que medir el tiempo en el que se es obtenida la ruta se vuelve inviable, ya que cada una de las rutas que puede ser generada puede llegar a tardar mayor o menor tiempo de acuerdo a las generaciones que sean necesarias para terminar por obtenerla.

Debido a lo anterior, se consideró otra forma de medir el tiempo de ejecución del programa para analizar su desempeño, es evaluar el tiempo de ejecución entre una generación y la siguiente. Al ser evaluada una matriz de tamaño $n \times n$, donde en cada una se evalúan una serie de condiciones para determinar el valor que contiene dicha celda, hay una diferencia de tiempo entre cada evaluación de la matriz mientras pasa cada una de las generaciones, por lo que se vuelve conveniente usar esta diferencia de tiempo como un parámetro de evaluación del desempeño para el algoritmo.

Durante las pruebas de funcionamiento iniciales del programa, también se encontró que existe una diferencia de desempeño en su ejecución entre Windows y Linux, por lo que para la recopilación de información y datos recabados se debe de tomar en cuenta las ejecuciones en ambos sistemas operativos, debido a que pueden existir diferencias significativas a pesar de que las condiciones iniciales sean similares entre sí.

6.3. Recopilación de datos del software y desempeño 1

La recopilación de datos sobre el software y desempeño fueron basadas en distintas configuraciones iniciales, las cuales fueron introducidas en el simulador como la condición inicial de la simulación. Estas pruebas fueron, además, recopiladas tanto en el sistema operativo de Windows como el de Linux, que son las dos plataformas en las cuales el programa puede funcionar. Los datos recopilados se muestran de laS Figuras 51 - 61.

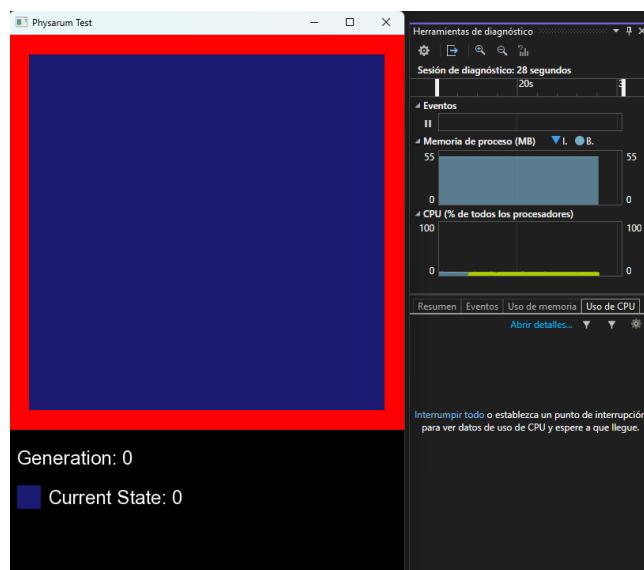


Figura 51: Rendimiento de la aplicación sin estados adicionales en Windows

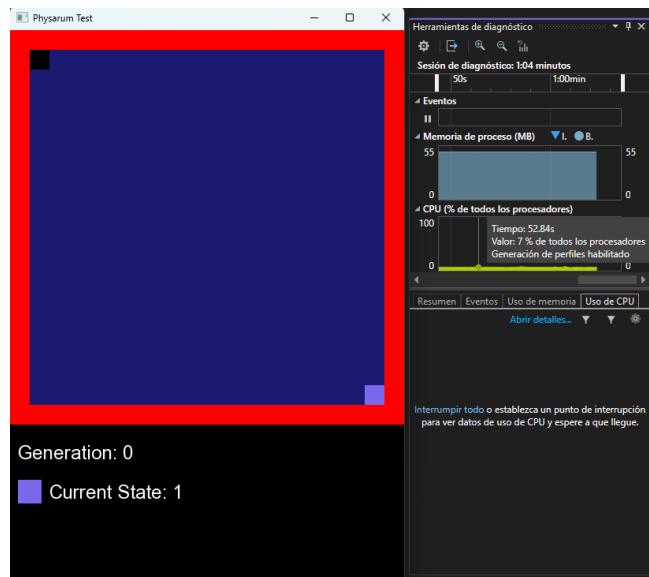


Figura 52: Rendimiento de la aplicación con una configuración inicial en Windows

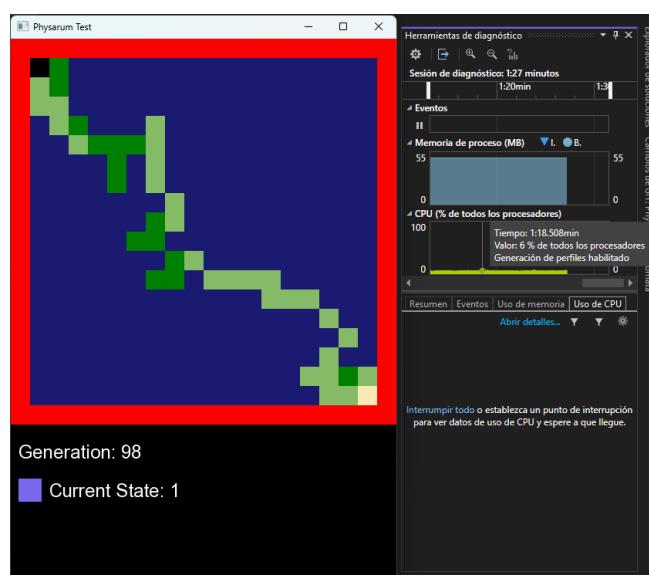


Figura 53: Rendimiento al obtener la ruta en Windows

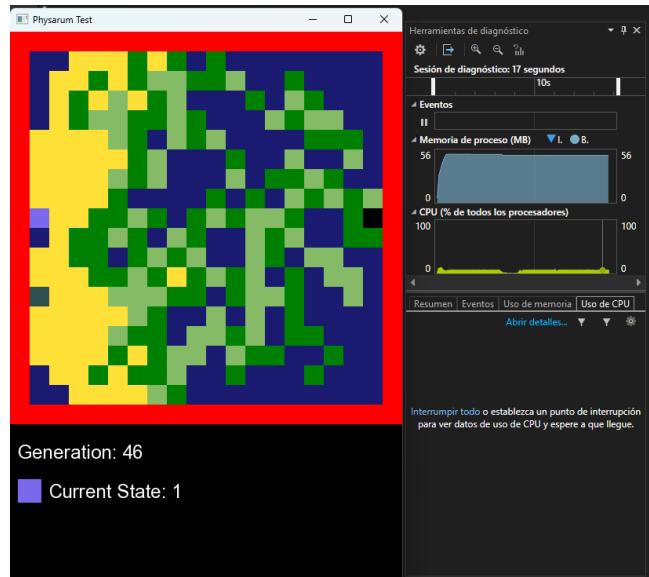


Figura 54: Rendimiento con configuración distinta a la anterior mientras se expande en Windows

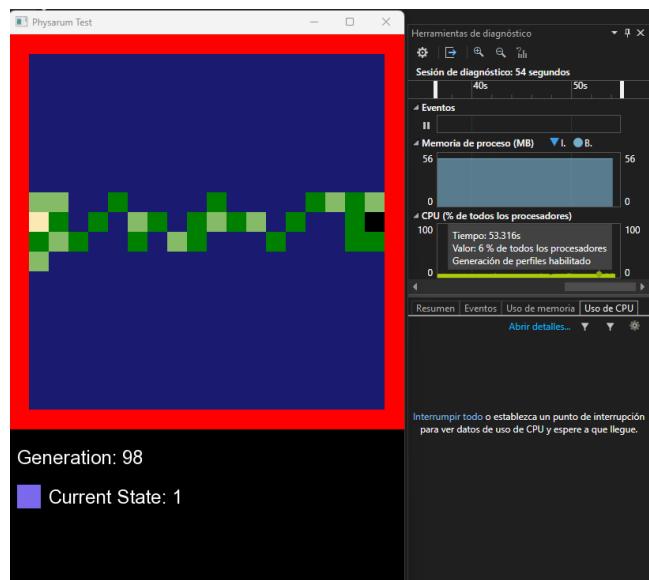


Figura 55: Rendimiento al obtener una ruta en otra configuración

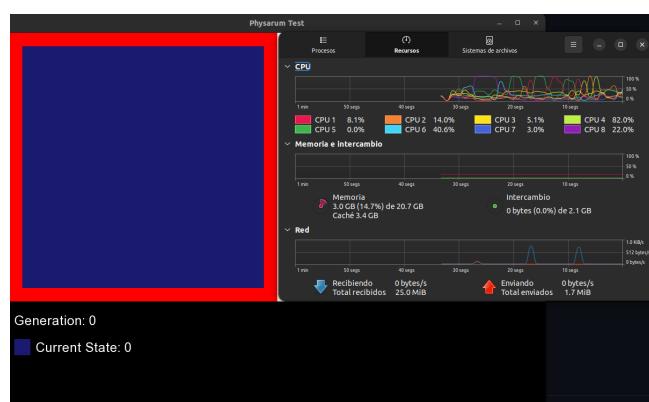


Figura 56: Rendimiento de la aplicación al iniciar el programa en Linux

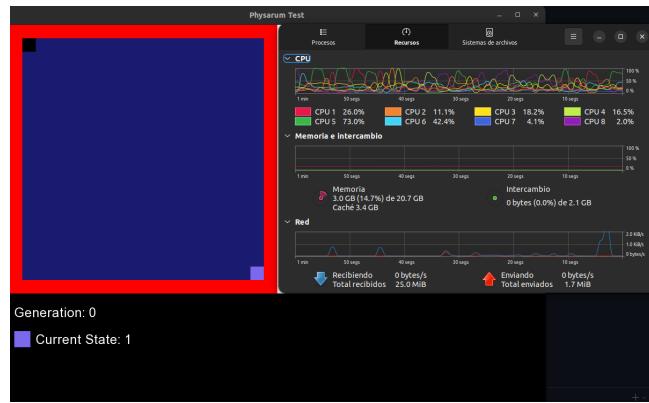


Figura 57: Rendimiento de la aplicación con una configuración inicial en Linux

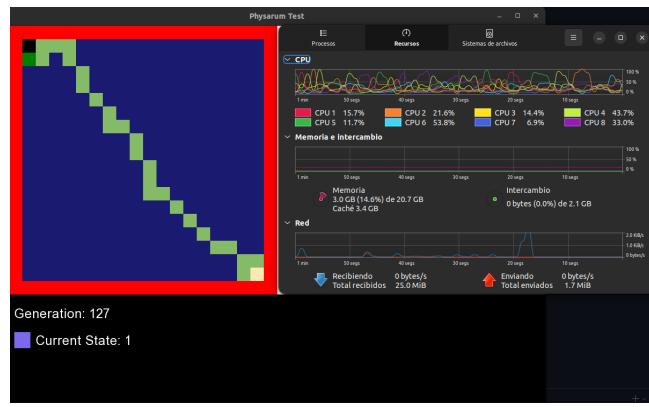


Figura 58: Rendimiento al obtener la ruta en Linux



Figura 59: Rendimiento de la aplicación con un estado inicial diferente en Linux

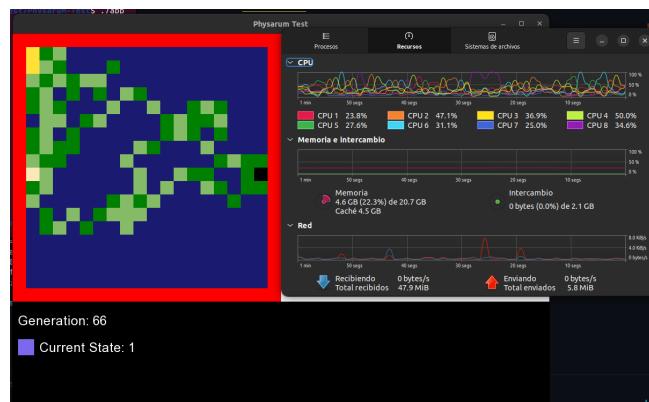


Figura 60: Rendimiento de la aplicación durante expansión de Physarum en Linux

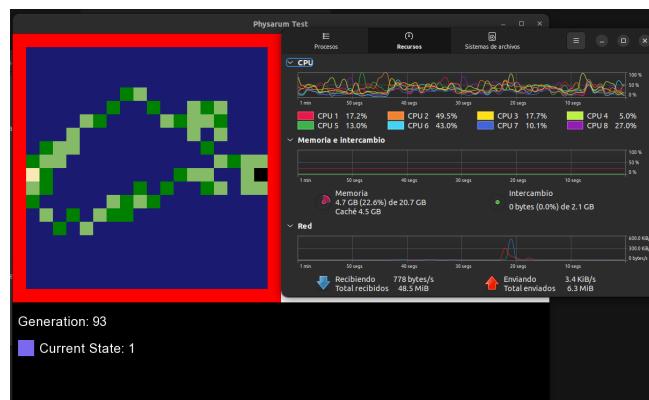


Figura 61: Rendimiento al obtener una ruta en otra configuración en Linux

6.4. Redacción del informe técnico inicial basado en la Iteración 1

Con la información que hasta el momento se ha recabado, se comienza con la redacción del reporte técnico, en la cual quedan asentados varios aspectos que fueron siendo encontrados durante las pruebas de funcionamiento, así como los resultados arrojados por las pruebas unitarias aplicadas al programa.

Además, se tiene en consideración la recopilación de información obtenida a partir de la evaluación del desempeño con algunas de las pruebas iniciales de funcionamiento aplicadas en el programa.

A demás en esta iteración se pudo comprobar el funcionamiento en general del Physarum, obteniendo así el funcionamiento principal de este, el cual es obtener una ruta a partir de un punto inicial y de un punto final. Es importante ver que debido a que es la primera iteración, han surgido muchas observaciones las cuales son de vital importancia para el análisis y mejora del programa, por lo que eso significa que mientras más se avance, el programa estará en un proceso de mejora continua.

Principalmente lo que se pudo notar es el funcionamiento en el caso base, que es el de generar la ruta a través de distintas configuraciones del estado inicial, siendo obtenidas rutas las cuales, son las que el robot interpretaría para poder realizar su funcionalidad de moverse de un lugar a otro.

En los distintos escenarios y pruebas que fueron realizadas en este proceso, se obtuvieron distintos resultados los cuales nos pueden dar una idea sobre el rendimiento y ventanas de oportunidad para la mejora del algoritmo, esto es debido a que se plantearon escenarios y pruebas cuyos resultados son reveladores y que ayudarían a obtener mejores resultados para el objetivo principal del programa, el cual es generar rutas.

6.5. Pruebas unitarias del algoritmo en simulación de mapas 2

Las pruebas realizadas, como ya se había mencionado anteriormente, son realizadas por medio del framework de Google Test hecho para programas que estén escritos en el lenguaje de programación C++.

En estas pruebas, se realizaron nuevas pruebas unitarias a algunas funciones que cambiaron un poco debido al desarrollo y los resultados de algunas pruebas que fueron realizadas en etapas anteriores. Se muestran algunas de las pruebas unitarias realizadas en la Figura 62.

Las funciones mostradas son las que fueron agregadas al archivo de pruebas unitarias que ya se habían realizado junto con las nuevas aplicadas debido a la introducción de nuevas funcionalidades dentro del programa. Como podemos ver, todas las pruebas resultaron exitosas, lo que es esperable al menos en su mayoría, ya que como se mencionó anteriormente en las primeras pruebas unitarias, el programa funciona a partir de un modelo matemático y debido a su naturaleza, el funcionamiento depende de que la implementación de ese modelo matemático a través del algoritmo sea correcta.

Las nuevas funciones corresponden a la implementación de la lectura de imágenes a transformar en mapas al inicio del programa, lo que facilita la entrada de entornos en los cuales se quiera simular al Physarum.

```

24 // Demonstrate some basic assertions.
25 TEST(phy_functions, SomeFunctions) {
26     EXPECT_EQ(0, phy_app.allNutrientsFounded()); //Test OK
27     EXPECT_EQ(0, phy_app.setCellState()); // Test OK
28     EXPECT_EQ(0, phy_app.cleanCells(sf::Time)); // Test OK
29     EXPECT_EQ(0, phy_app.showPhysarum()); // Test OK
30     EXPECT_EQ(0, phy_app.evaluatePhysarum()); // Test OK
31     EXPECT_EQ(0, phy_app.physarumTransitionConditions()); // Test OK
32     EXPECT_EQ(0, phy_app.getRoute()); // test ok
33 }
34 // Demonstrate some basic assertions.
35 TEST(loadmap, SomeFunctions) {
36     EXPECT_EQ(0, loadmap.convertImageToMap()); //Test OK
37     EXPECT_EQ(0, loadmap.setDataToArray()); //Test OK
38     EXPECT_EQ(0, loadmap.grayscaleImage(sf::Time)); //Test OK
39     EXPECT_EQ(0, loadmap.setDataToArray()); //Test OK
40 }
41 // Demonstrate some basic assertions.
42 TEST(autotest, SomeFunctions) {
43     EXPECT_EQ(0, loadmap.getRandom(const int, const int)); //Test OK
44 }
45
46 }
47

```

Figura 62: Prueba unitaria 2

6.6. Pruebas de aceptación en escenarios pequeños

6.6.1. 1ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado.

Flujo: El programa es cargado.

Se le presenta el lienzo en pantalla.

Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección.

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.6.2. 2da Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.6.3. 3ra Prueba de aceptación: Inicialización de la simulación.

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.7. Evaluación de desempeño del software en pruebas iniciales 2

En las pruebas iniciales ya en entornos controlados y de un tamaño bastante reducido, se hizo del conocimiento de algunos detalles que surgen de la implementación de algunas funcionalidades, las cuales fueron revisadas a detalle.

Como se mencionó anteriormente, como medida de desempeño se utilizará el tiempo que tarda el software entre el cambio de generación entre una evaluación de las reglas y otra. A diferencia de las anteriores evaluaciones, en este caso al ya implementar un mapa que está basado ya en una locación en específico, se tiene que realizar una configuración la cual sea lo más aproximado a su locación en el mundo real.

Las siguientes pruebas corresponden a un mapa el cual se tiene por defecto y representa en el área en la cual el robot fue colocado en la vida real para poder comprobar el funcionamiento del seguimiento de la ruta. Es importante recalcar que las pruebas se realizan en ambos sistemas operativos, los cuales son, como es asumido desde el inicio, el sistema operativo de Windows y Linux, usando como principal distribución la de Ubuntu en el caso de Linux. Esto se puede ver en las Figuras 63 - 68.

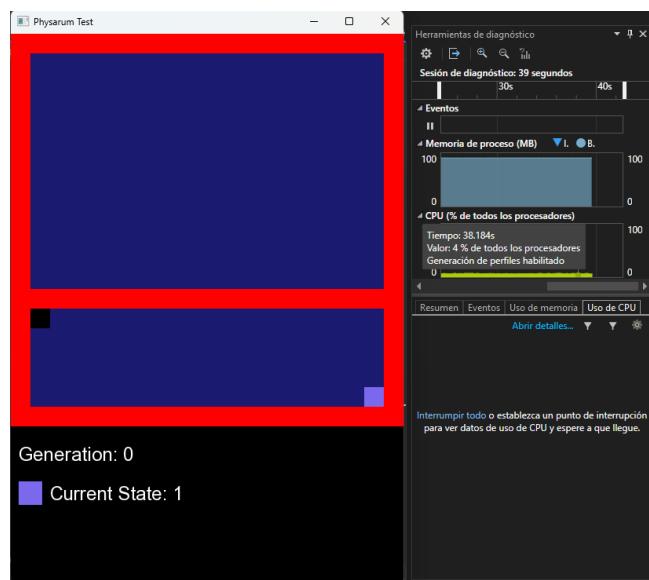


Figura 63: Estado inicial con una barrera simulando un cuadro cerrado en Windows

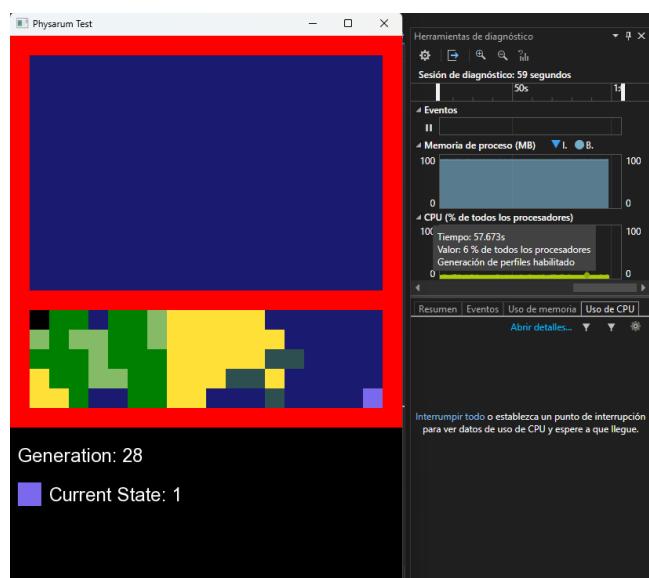


Figura 64: Rendimiento de la aplicación en un entorno cerrado y durante la expansión del Physarum en Windows

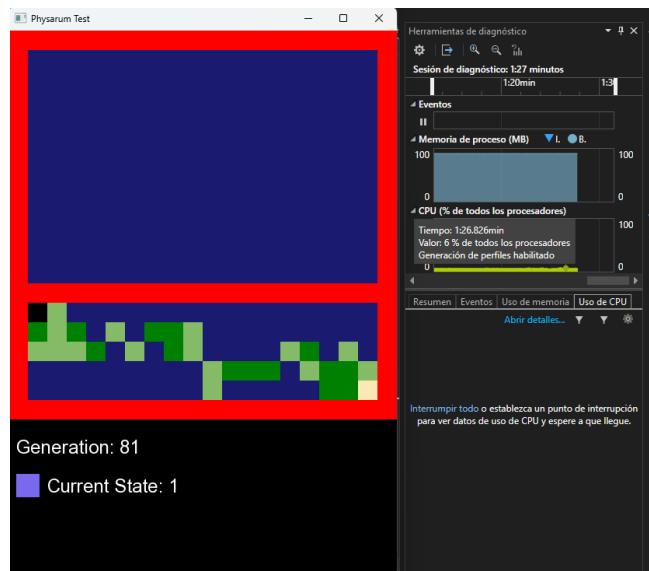


Figura 65: Rendimiento al generar la ruta con el Physarum Windows

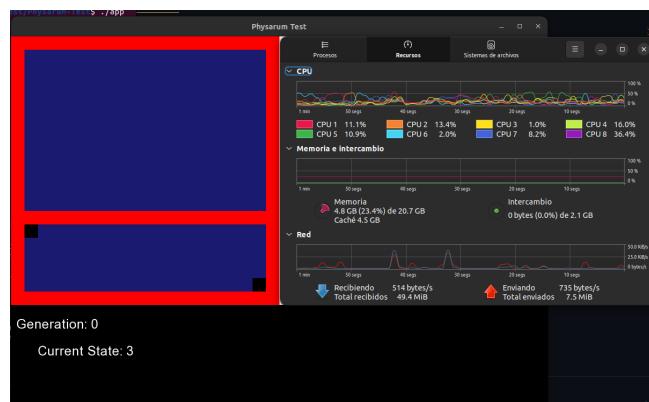


Figura 66: Rendimiento de la aplicación durante el establecimiento del estado inicial en Linux

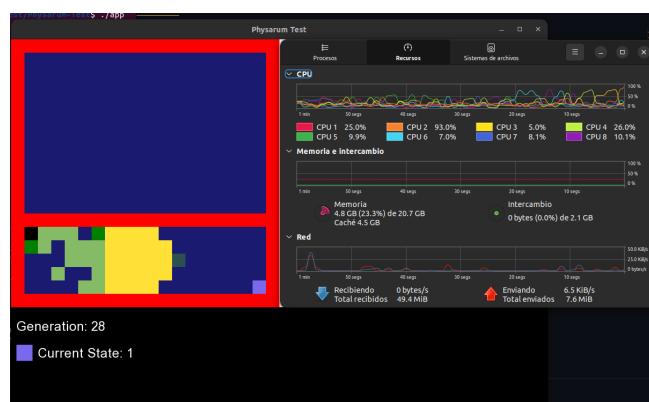


Figura 67: Rendimiento de la aplicación durante la expansión del Physarum en Linux



Figura 68: Rendimiento al generar la ruta con el Physarum en Linux

6.8. Recopilación de datos del software y desempeño 2

Los datos que han sido recopilados hasta el momento con los que están en base a las pruebas con los mapas que representan ya un escenario de la vida real. Como se puede apreciar en la Figura 69 hay distintos estados que se pueden colocar en el lienzo del estado inicial antes de disparar el simulador para obtener la ruta. Uno de estos estados es el estado 2, el cual representa un repelente que, a efectos prácticos de nuestro software, son la representación de lo que fueran paredes u obstáculos en el entorno o el mapa que se coloca como estado inicial.

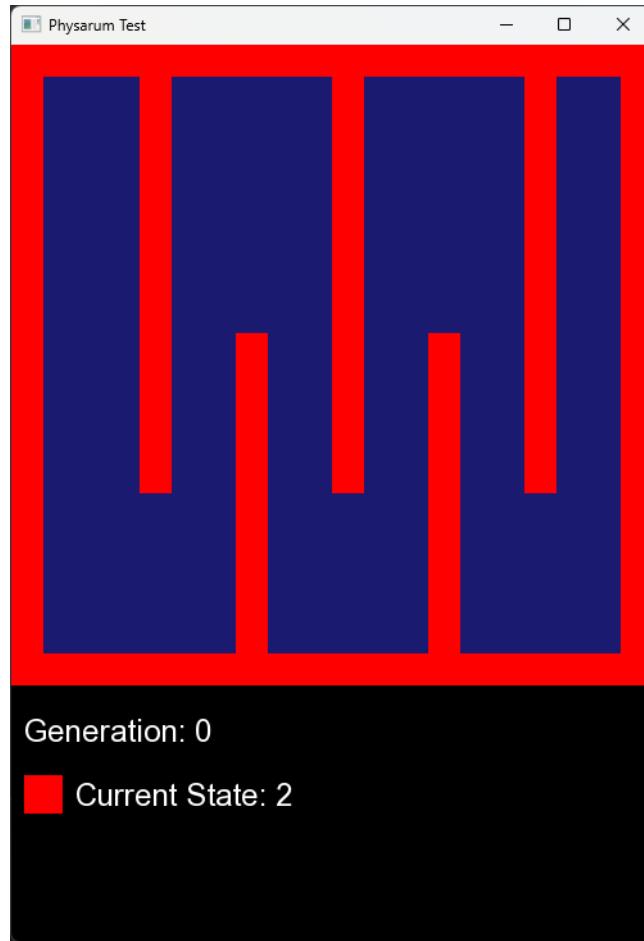


Figura 69: Una configuración inicial con osbtaculos

El repelente tiene como función en el simulador, impedir el paso del Physarum en los puntos donde este se encuentre, por lo que funciona bien como si fuera la representación de paredes o barreras de los mapas en donde no se puede avanzar en la vida real.

Así que esta vez, las pruebas para medir el desempeño, que como anteriormente se mencionó, se hacen midiendo el tiempo entre el avance de un estado y el siguiente, fueron sobre un estado inicial donde se coloca inicialmente una serie de obstáculos para llegar al destino final, esto es como si fuera un laberinto por el cual el Physarum debe de pasar para poder llegar a su destino. Esto hace que la evaluación en general tarde un poco más que si fuera un mapa o estado inicial sin algún tipo de obstáculo.

De las Figuras 70 - 77, se muestran algunos de los resultados obtenidos con algunas configuraciones.

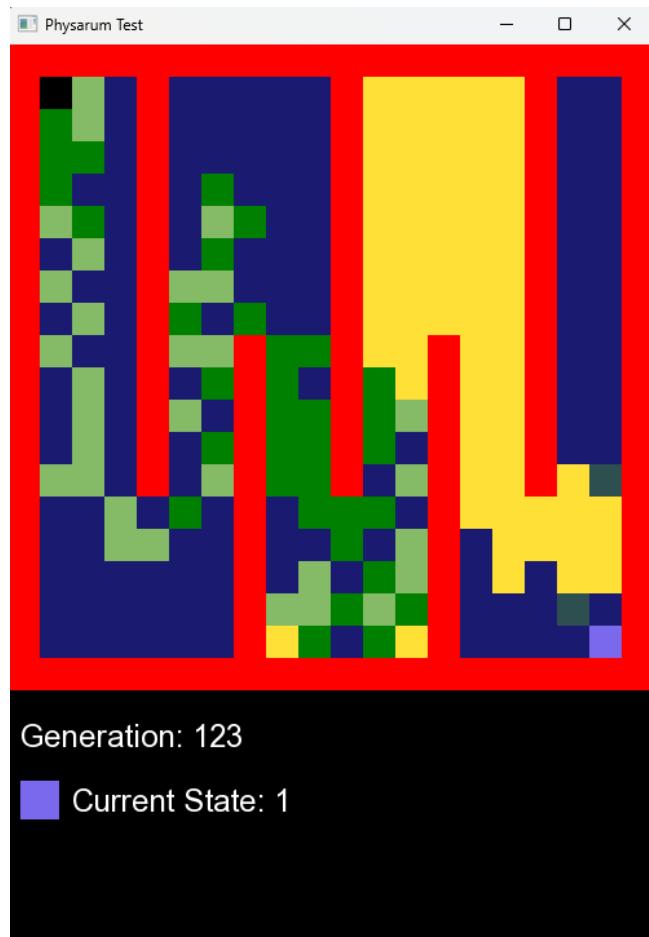


Figura 70: Expansión del algoritmo de Physarum.

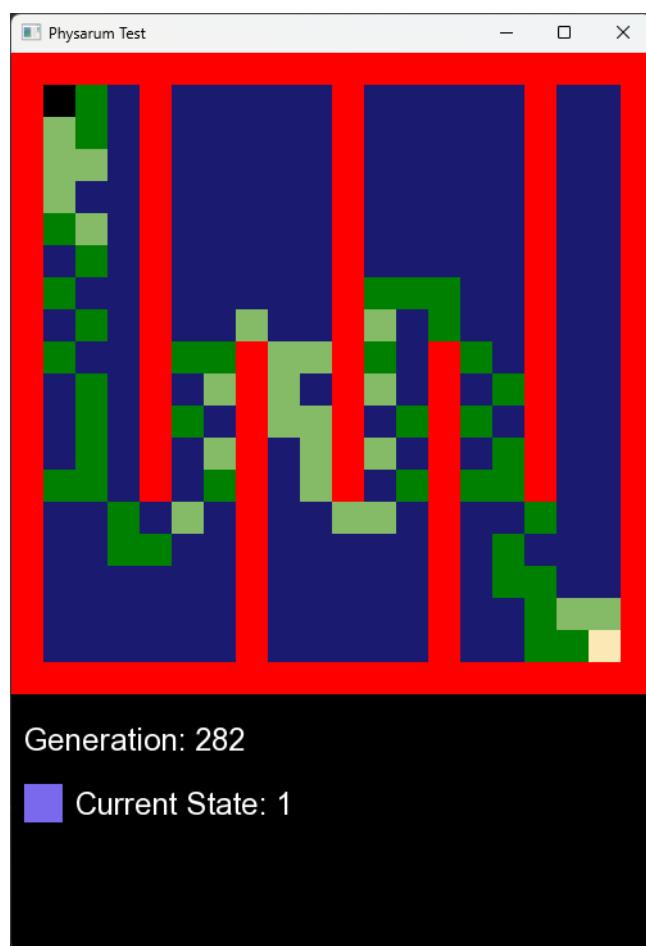


Figura 71: Ruta generada a partir de una configuración inicial complicada.

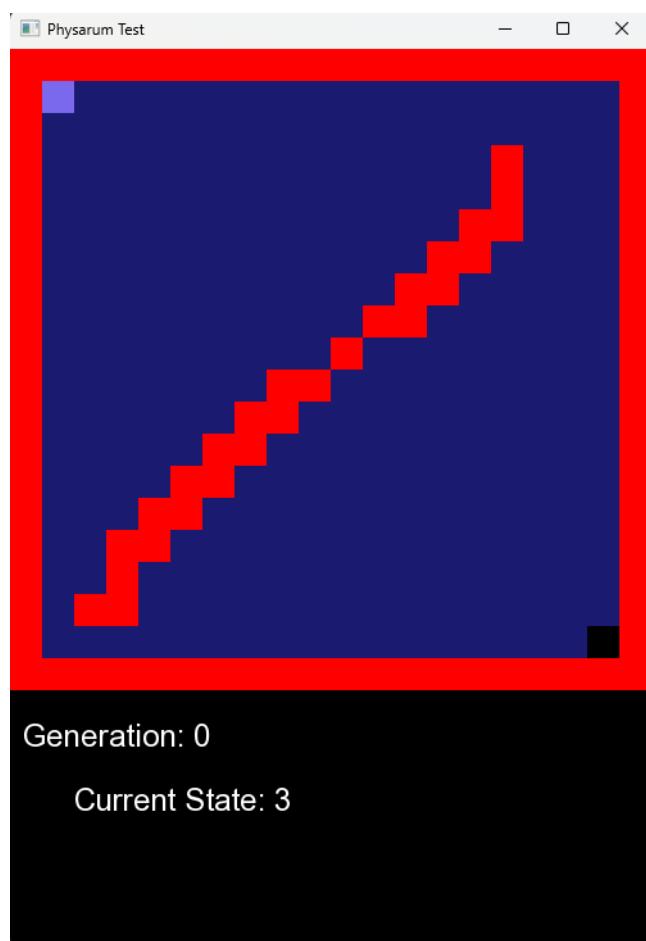


Figura 72: Estado inicial de una configuración.

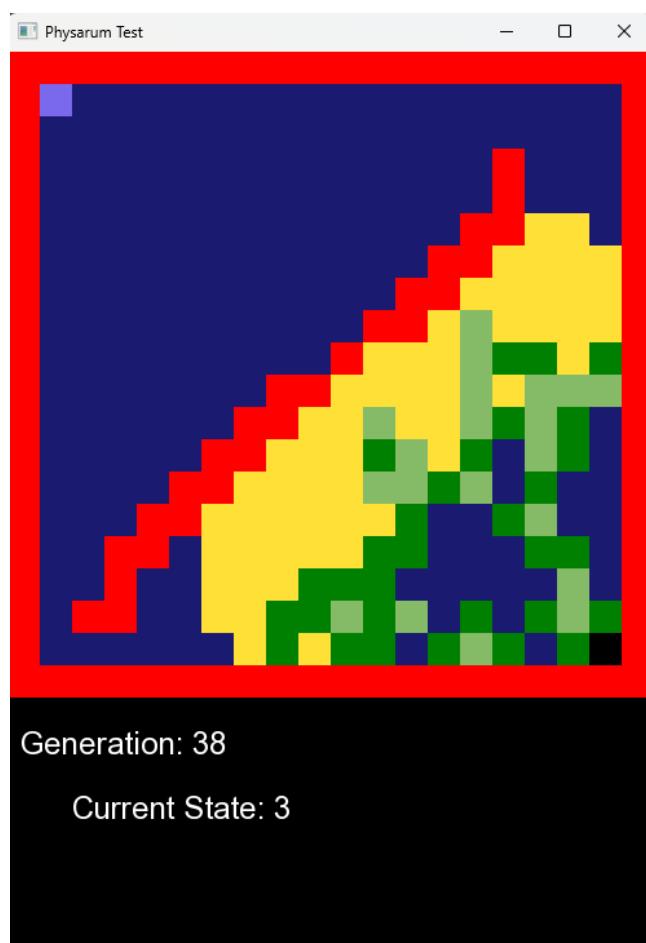


Figura 73: Expansión del algoritmo del Physarum con una pared diagonal en medio.

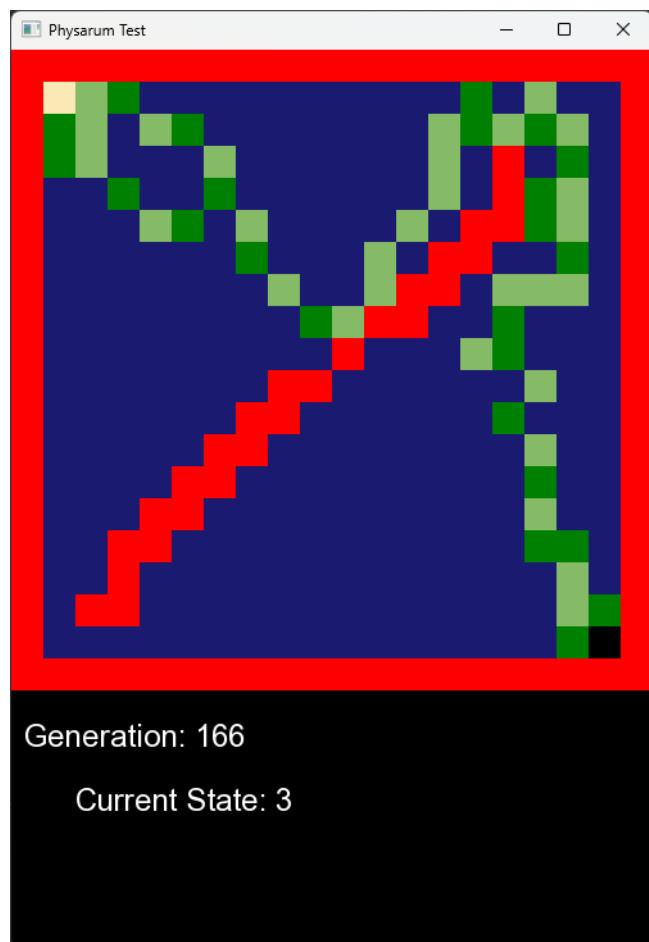


Figura 74: Generación de la ruta satisfactoria que es generada durante la ejecución y finalización del algoritmo.

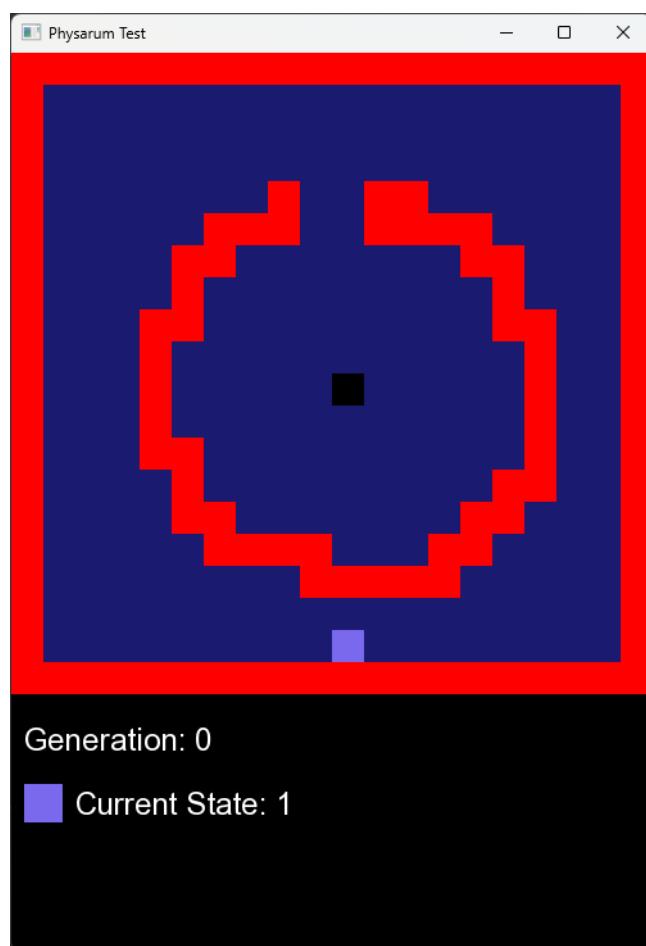


Figura 75: Estado inicial con configuración circular.

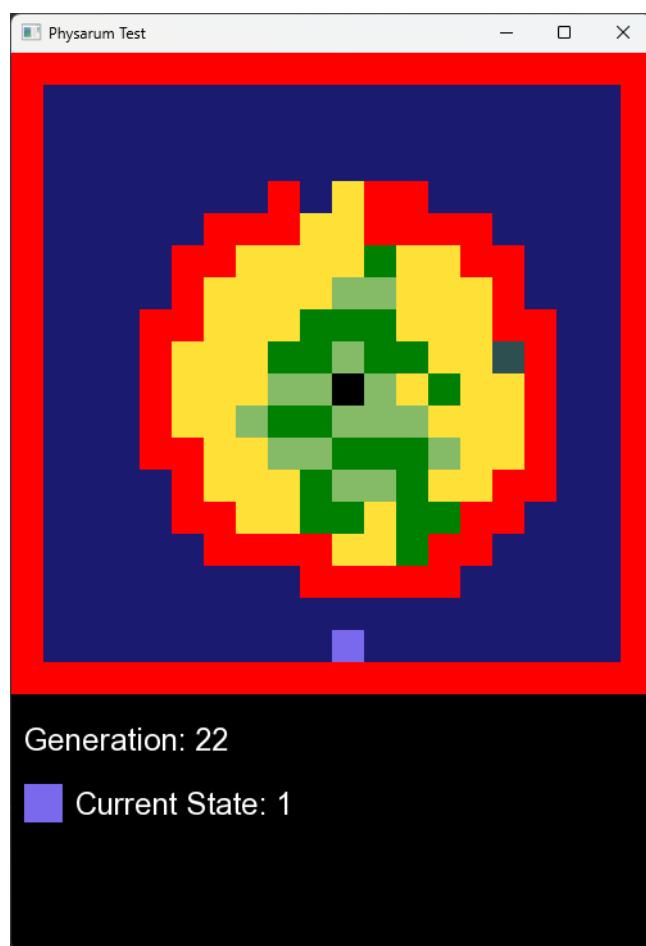


Figura 76: Expansión con una configuración de repelentes circular.

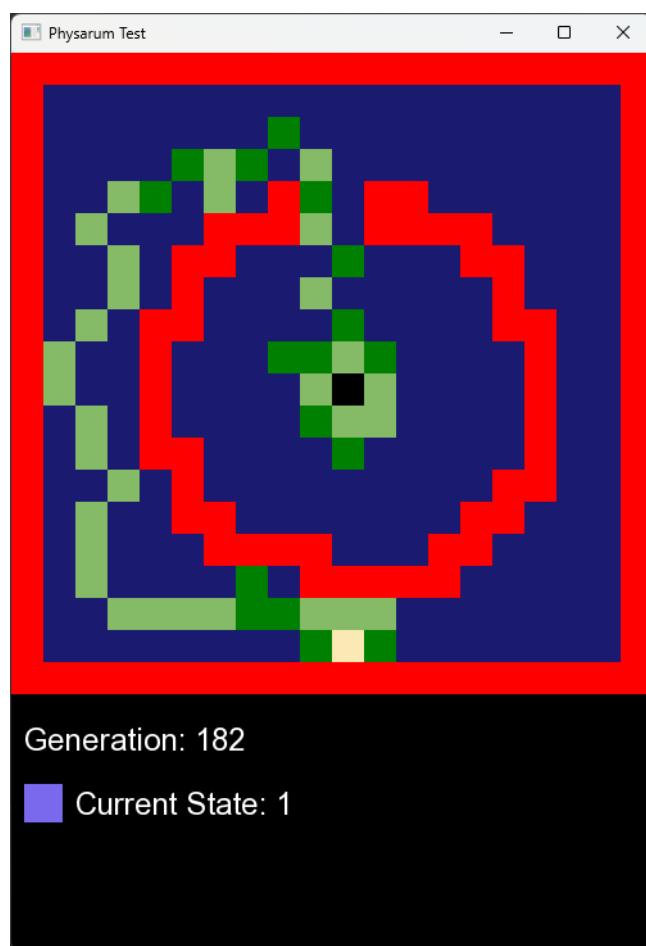


Figura 77: Ruta generada después de la finalización de la simulación.

6.9. Pruebas de aceptación en entornos medianos

Para la realización de estas pruebas, se utilizaron exactamente los mismos parámetros de las pruebas unitarias anteriores, ya que la única diferencia es que el lienzo es mucho más grande, por lo que esta prueba es agregada también.

6.9.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión $n \times n$ correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.9.2. 2da Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado.

Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección.

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.9.3. 3ra Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.9.4. 4ta Prueba de aceptación: Inicialización de la simulación.

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.10. Pruebas de aceptación en entornos complejos simulados 1

Estas pruebas comprenden a las anteriores pruebas, ya que al igual que las pruebas de aceptación de la iteración anterior, para el entorno más complejo, el tamaño de lienzo es lo único que cambia, pero lo único que se agrega ahora es la carga de un mapa de un entorno real.

6.10.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión n x n correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código.
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.10.2. 2da Prueba de aceptación: Carga de mapas

Descripción: En el código se cargan imágenes, las cuales corresponden a mapas, donde éstas se cargan para colocarlos en el lienzo al iniciar el programa

Flujo:

- Se coloca la dirección de la imagen
- Se coloca un tamaño para tanto la imagen como para el lienzo
- Se inicia el programa.
- El programa al iniciar, carga la configuración, mostrando la imagen del mapa, convertido al lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.

- Se pueden colocar estados en cualquier área correspondiente al lienzo.
- La imagen del mapa es colocada correctamente en el lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.10.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado. Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.
- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.10.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se es desplegado, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.10.5. 5ta Prueba de aceptación: Inicio de la simulación

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El numero de generaciones aumenta.
- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

6.11. Evaluación final del desempeño del software 1

Para la evaluación del desempeño, en esta ocasión fue mucho más claro la medida del tiempo transcurrido entre el paso de una generación y otra, y esto es debido al tamaño en el cual crece el arreglo de dimensión $n * n$, ya que mientras se tenga un n mucho más alto, el rango de evaluación también crece considerablemente y también lo hacen las evaluaciones en cada una de las células, donde las reglas del algoritmo del Physarum deben de hacer posible la transición entre cada uno de los estados.

De las Figuras 78 - 81, se muestran algunas de las configuraciones junto con sus resultados y los datos recopilados en tamaños mucho más extensos:

La evaluación final del desempeño del software se centró en medir el tiempo de simulación y la precisión de las rutas generadas bajo diversas condiciones iniciales y sistemas operativos. Se comprobó que el algoritmo, aunque eficiente en la mayoría de los casos, presenta diferencias de desempeño notables entre Windows y Linux, con mejores resultados en Linux debido a su mayor capacidad de procesamiento en simulaciones intensivas.

El análisis de los tiempos de ejecución entre una generación y la siguiente permitió identificar mejoras clave en la eficiencia del software. Los resultados mostraron que, en entornos de mayor complejidad, el tiempo de procesamiento incrementa proporcionalmente al número de barreras y obstáculos, lo que requiere optimizaciones adicionales en futuras iteraciones.

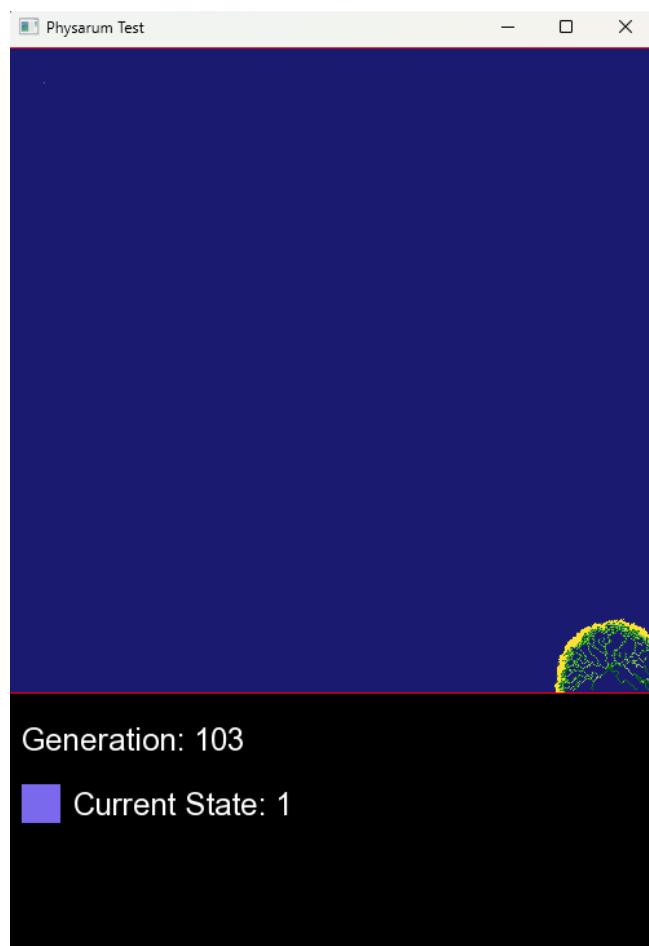


Figura 78: Expansión y evaluación de la simulación en un entorno mucho mas grande (500 x 500)

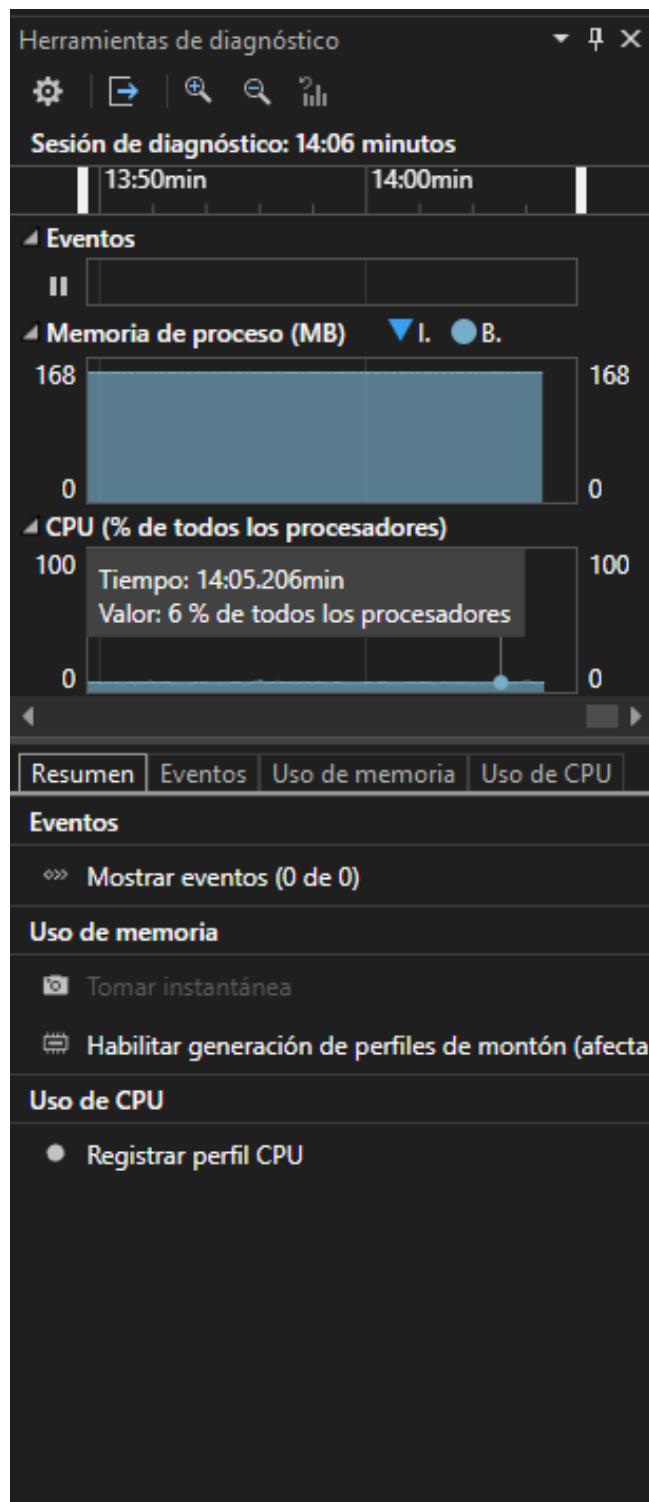


Figura 79: Cantidad de memoria y procesamiento durante la ejecución del simulador

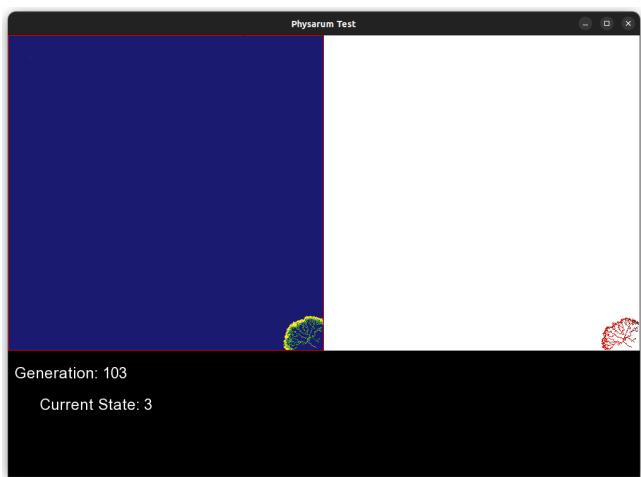


Figura 80: Expansión en el mismo tamaño del que fue realizado en Linux.



Figura 81: Uso de los recursos del computador durante la ejecución del simulador.

6.12. Recopilación y análisis de datos del rendimiento del robot de IEEE 1872.1 e IEEE 2914

En el mundo de las pruebas y otro tipo de documentación, son importantes las normas, debido a que permiten un estándar para el manejo de distintos elementos dentro de un sistema, además de que la estandarización fomenta el intercambio sencillo o interacción entre componentes los cuales no tienen mucho que ver con otros, pero que al existir una norma, un modo de estandarización, se pueden crear formas en las que ambas partes puedan coexistir entre sí y entre muchas otras más.

La importancia de la IEEE 1872.1 radica en que esta norma nos permite definir ciertos elementos y contextos en los cuales se puede realizar un intercambio de información entre pares, los cuales usualmente están relacionados a la electrónica y robótica, así como al hardware y software. En el caso del software y del robot al cual el software le genera una ruta para su posterior seguimiento, usa criterios de interoperabilidad que son los más comunes a la hora de poder conectar los resultados obtenidos por el programa, así como la forma de interpretar esos datos para cumplir el objetivo principal del algoritmo de Physarum: generar una ruta la cual el robot pueda seguir sin problemas para llegar a un destino final.

Para el caso de el envío de información entre el robot y el software, se hace uso de herramientas FTP, así como de un servidor el cual recibe peticiones HTTP con las cuales se pueden recibir y enviar los datos que sean necesarios.

A pesar de que el software pertenezca a una de las jerarquías más bajas dentro de la estructura de la IEEE, es de vital importancia que éste ultimo funcione correctamente y cumpla su función para que el robot tenga un propósito y pueda tener la utilidad necesaria.

Y dentro de las demás normas, también se toma en cuenta la norma IEEE 2914, que dentro de la parte generadora de la ruta y demás trabajo del software de ruteo a partir del Physarum, se vuelve importante tomarla en cuenta. Esto porque a diferencia de otros sistemas donde la toma de decisiones trata de ser siempre la óptima y en muchos casos están se pueden llegar a replicar para tener un registro y una idea de que decisiones te llevan a un determinado resultado, en el caso del simulador del Physarum las decisiones son aleatorias en todo momento, además, de que la aleatoriedad es aplicada a todos los elementos que componen al arreglo de dos dimensiones, por lo que se vuelve muy difícil de rastrear y obtener siempre algún mismo resultado, puesto que en cada ejecución del algoritmo, la configuración y resultados obtenidos serán distintos, por lo que lo único que se debe de revisar es que el resultado sea en su mayoría satisfactorio, que a pesar de que no siempre la ruta sea la más óptima, se acerque a hacerlo, para así generar una mejor movilidad y toma de decisiones y tratamiento de la información.

6.13. Redacción del informe con mejoras de la Iteración 2

En esta etapa se realizó un análisis exhaustivo de los resultados obtenidos en la Iteración 2, enfocándonos en la identificación de áreas clave de mejora en el software. A partir de las pruebas realizadas y los datos recopilados, se implementaron ajustes en el algoritmo para optimizar su rendimiento en entornos más complejos. Estos ajustes se centraron en mejorar la eficiencia en la generación de rutas, reduciendo los tiempos de procesamiento en mapas de mayor tamaño y con más obstáculos.

Además, se revisaron y actualizaron las pruebas unitarias para asegurar la correcta implementación de las mejoras, enfocándonos en la estabilidad del sistema al enfrentarse a diferentes

configuraciones iniciales. Los cambios implementados lograron una reducción significativa en los tiempos de simulación, especialmente en los casos donde se incluían múltiples barreras o rutas más largas.

Asimismo, se refinaron las estrategias para eliminar células innecesarias en las rutas generadas, lo que permitió un recorrido más eficiente por parte del robot, optimizando su capacidad de seguir las coordenadas calculadas por el simulador. Estos avances fueron documentados y evaluados, dejando asentadas las mejoras que serán fundamentales para las siguientes iteraciones.

El informe incluye un análisis detallado de los tiempos de ejecución, las rutas generadas en diversas configuraciones y las métricas de desempeño obtenidas en sistemas operativos tanto Windows como Linux. Estos resultados muestran un progreso claro en el desarrollo del software, destacando las mejoras implementadas para su robustez y precisión en escenarios más exigentes.

6.14. Recopilación final de datos del software para el informe 1

Los datos hasta ahora recopilados nos muestran un panorama mucho más amplio del que se tenía al inicio de la realización y desarrollo del software, esto es, se han recopilado bastantes datos los cuales ayudan a que se pueda encontrar una mucho mejor forma de realizar algunas de las funcionalidades o se hayan encontrado algunas carencias de las cuales muchas veces pasan desapercibidas en una primera instancia.

El programa, como vimos al inicio, funciona correctamente y cumple su función de generar rutas, no obstante, las rutas muchas veces tienden a ser un poco imperfectas debido a la propia naturaleza del algoritmo en sí, además de que muchas veces se generan bifurcaciones o hay demasiadas células en determinadas partes de la ruta, lo que fomenta que se encuentre una mayor dificultad a la hora de generar una ruta óptima para ser enviada al robot, además de que el proceso de generación de coordenadas se complica debido a que cada una de estas debe de tener algún tipo de orden el cual ayude a que el robot tenga una forma mucho más fácil de moverse en su entorno real y que lo haga sin tener que dar muchas vueltas.

Por lo que se empezaron a generar soluciones, como lo es limpiar y eliminar algunas de las células dentro del arreglo con el objetivo de optimizar la ruta, además de generar algoritmos los cuales puedan tener una mejor aproximación al correcto ordenamiento de las células presentes y que al final, la ruta sea una la cual el robot pueda seguir sin problemas y que, respecto a la generada inicialmente por el simulador del Physarum, sea mucho más conveniente. Por último, se pudo notar que mientras más incrementaba el tamaño del lienzo, el programa tiende a alentarse un poco más, esto es debido a la gran cantidad de información que tiene que desplegar al mismo tiempo y dificulta el correcto manejo del software, por lo que en las futuras implementaciones e iteraciones se tendrá que mejorar este aspecto para así poder optimizar los recursos que son usados y tener una mejor experiencia a la hora de usar el simulador y así, tener un mejor confort a la hora de generar una ruta.

6.15. Documentación del avance segunda iteración

Durante la segunda iteración, se realizaron mejoras significativas en la funcionalidad del simulador y su capacidad para generar rutas más óptimas y adaptarse a entornos más complejos. Se implementaron ajustes en el algoritmo basado en el Physarum Polycephalum para reducir los tiempos de simulación y mejorar la precisión en la selección de rutas. Además, se realizaron

pruebas exhaustivas en distintos sistemas operativos (Windows y Linux), documentando las diferencias de rendimiento y comportamiento del software en ambos entornos.

En esta iteración, también se introdujeron mejoras en la interfaz de control del robot, facilitando el seguimiento de rutas generadas y la interacción con los sensores del robot, como el sistema LiDAR. La documentación incluye los cambios implementados, los resultados obtenidos y las estrategias propuestas para las pruebas de aceptación en entornos complejos, que servirán de base para la próxima iteración.

Durante la segunda iteración, se realizaron mejoras significativas en la funcionalidad del simulador y su capacidad para generar rutas más óptimas y adaptarse a entornos más complejos. Se implementaron ajustes en el algoritmo basado en el Physarum Polycephalum para reducir los tiempos de simulación y mejorar la precisión en la selección de rutas. Además, se realizaron pruebas exhaustivas en distintos sistemas operativos (Windows y Linux), documentando las diferencias de rendimiento y comportamiento del software en ambos entornos.

En esta iteración, también se introdujeron mejoras en la interfaz de control del robot, facilitando el seguimiento de rutas generadas y la interacción con los sensores del robot, como el sistema LiDAR. La documentación incluye los cambios implementados, los resultados obtenidos y las estrategias propuestas para las pruebas de aceptación en entornos complejos, que servirán de base para la próxima iteración.

6.16. Pruebas de aceptación en entornos complejos simulados 3

Para validar el rendimiento del software en escenarios más desafiantes, se llevaron a cabo pruebas de aceptación en entornos complejos simulados. En estas pruebas, se introdujeron obstáculos adicionales y laberintos más elaborados en el lienzo de simulación, lo que permitió observar el comportamiento del algoritmo ante situaciones de alta densidad de barreras. Los resultados mostraron que el algoritmo es capaz de adaptarse a estos entornos, generando rutas eficientes incluso con una mayor presencia de obstáculos. Sin embargo, se identificaron áreas donde el tiempo de cálculo aumenta significativamente en mapas de gran tamaño, lo que sugiere la necesidad de optimizar ciertos componentes del algoritmo en futuras iteraciones.

6.16.1. 1ra Prueba de aceptación: Modificación del tamaño del lienzo

Descripción: En el código se colocan las dimensiones del lienzo y el arreglo de dimensión n x n correspondiente al autómata celular, por lo que se quiere comprobar si la modificación de estos valores cambia debidamente el tamaño del lienzo mostrado en pantalla.

Flujo:

- Se cambian los valores del lienzo en el código.
- Se inicia el programa.
- El programa muestra los cambios correspondientes a los valores que fueron colocados en el código al desplegarse el lienzo.

Criterios de aceptación:

- El programa inicia correctamente.

- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.16.2. 2da Prueba de aceptación: Carga de mapas

Descripción: En el código se cargan imágenes, las cuales corresponden a mapas, donde éstas se cargan para colocarlos en el lienzo al iniciar el programa

Flujo:

- Se coloca la dirección de la imagen
- Se coloca un tamaño para tanto la imagen como para el lienzo
- Se inicia el programa.
- El programa al iniciar, carga la configuración, mostrando la imagen del mapa, convertido al lienzo.

Criterios de aceptación:

- El programa inicia correctamente.
- El tamaño del lienzo es acorde a los valores colocados en el código.
- Se pueden colocar estados en cualquier área correspondiente al lienzo.
- La imagen del mapa es colocada correctamente en el lienzo.

El software cumple con cada uno de los criterios de aceptación, además de seguir el flujo con el que se había planeado desde el inicio, así que se considera a esta prueba como finalizada y con buenos resultados obtenidos.

6.16.3. 3ra Prueba de aceptación: Elección de estados a través del teclado.

Descripción: El usuario, al iniciar el programa, puede elegir por medio de las teclas el estado actual que quiera colocar en el lienzo desplegado. Flujo:

- El programa es cargado.
- Se le presenta el lienzo en pantalla.
- Al elegir los estados con las teclas numéricas, estas son reflejadas en la pantalla.

Criterios de aceptación:

- Al presionar una tecla numérica, cambia al respectivo estado que representa.

- El estado elegido es plasmado en el lienzo al dar clic.
- Puede cambiar en cualquier momento el estado de su elección

Para la prueba anterior, los resultados fueron satisfactorios puesto que cumplió con todos los criterios de aceptación, siguiendo el flujo que se describía en la prueba.

6.16.4. 4ta Prueba de aceptación: Colocación de los estados inicial y final.

Descripción: Al presionar con el botón izquierdo del mouse en el lienzo que se despliega, se colocan en pantalla el estado correspondiente al seleccionado previamente con el teclado. Particularmente se evalúa que se coloque el estado inicial y el estado que corresponde al nutriente no encontrado, siendo los principales componentes en el estado inicial que es necesario para iniciar la simulación.

Flujo:

- Se despliega el lienzo en pantalla
- Se elige por medio del teclado el estado deseado.
- Al dar clic en pantalla, este estado es puesto en pantalla

Criterios de aceptación:

- El estado es colocado en el lienzo correctamente.
- La pantalla muestra el estado actual y el color correspondiente al teclado.
- La colocación de los estados solo puede ser colocada dentro del área del lienzo y no por fuera.

En la segunda prueba, se lograron cumplir los criterios de aceptación de manera satisfactoria, por lo que la prueba se considera como realizada y cumplida correctamente.

6.16.5. 5ta Prueba de aceptación: Inicio de la simulación

Al presionar el botón de ENTER en el teclado, la simulación del organismo Physarum es iniciada, y si fueron colocados correctamente los estados correspondientes al estado inicial y al nutriente no encontrado, entonces se genera una ruta entre cada estado.

Flujo:

- El usuario coloca los estados deseados en pantalla.
- El usuario presiona la tecla ENTER.
- El programa inicia con la simulación y si son colocados los estados necesarios, se crea una ruta entre estos.

Criterios de aceptación:

- El número de generaciones aumenta.

- Si es colocado el estado 3, entonces el Physarum inicia su expansión.
- Si son colocados el estado 3 y 1, se genera una ruta entre estos una vez finalizada la simulación.

La prueba anterior fue realizada y cumplió con los criterios de aceptación que fueron solicitados, por lo que esta prueba se da por finalizada y con un resultado positivo.

Estas pruebas fueron realizadas en entornos simulados de tamaño más grande, y los resultados revelaron que las rutas generadas continúan siendo precisas, aunque con un mayor tiempo de procesamiento, especialmente en sistemas con recursos limitados.

6.17. Evaluación final del desempeño del software 2

La evaluación final del desempeño del software se centró en medir el tiempo de simulación y la precisión de las rutas generadas bajo diversas condiciones iniciales y sistemas operativos. Se comprobó que el algoritmo, aunque eficiente en la mayoría de los casos, presenta diferencias de desempeño notables entre Windows y Linux, con mejores resultados en Linux debido a su mayor capacidad de procesamiento en simulaciones intensivas. Esto se reflejó en las Figuras 82 y 83.

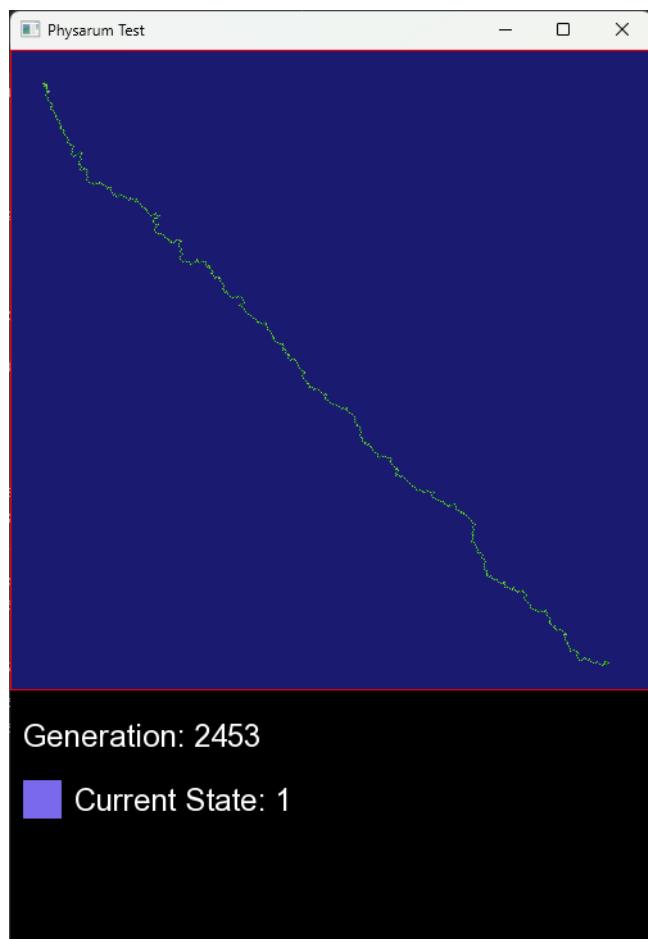


Figura 82: Generación de una ruta después de el paso de generaciones en Windows (500 x 500)

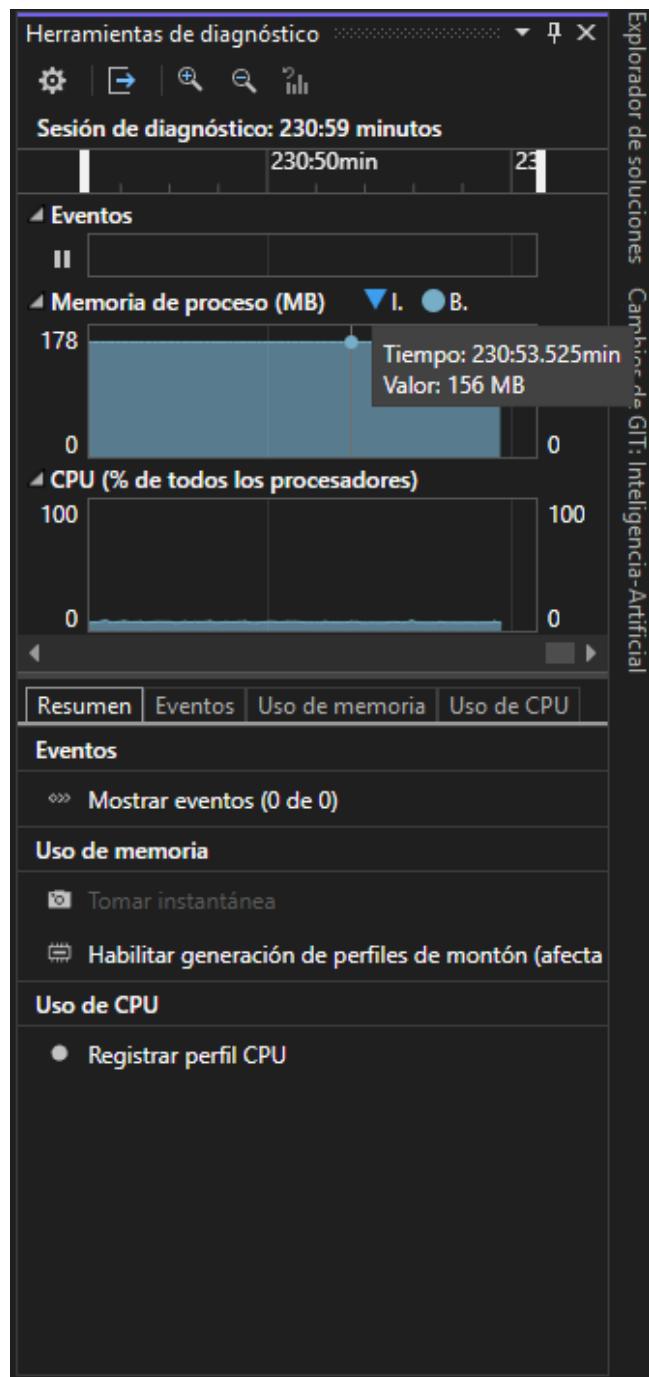


Figura 83: Uso de la memoria y CPU durante la obtención de la ruta del Physarum.

El análisis de los tiempos de ejecución entre una generación y la siguiente permitió identificar mejoras clave en la eficiencia del software. Los resultados mostraron que, en entornos de mayor complejidad, el tiempo de procesamiento incrementa proporcionalmente al número de barreras y obstáculos, lo que requiere optimizaciones adicionales en futuras iteraciones.

6.18. Recopilación final de datos del software para el informe 2

Los datos finales recopilados del software incluyen mediciones de tiempos de ejecución, precisión de rutas generadas y análisis de desempeño en distintos escenarios.

Se realizaron simulaciones en mapas tanto simples como complejos, con variaciones en los obstáculos y configuraciones iniciales, para obtener un conjunto completo de datos que permitieran evaluar el rendimiento general del software.

Además, se documentaron las diferencias entre los resultados obtenidos en Windows y Linux, mostrando cómo el sistema operativo afecta la eficiencia del algoritmo. Estos datos serán fundamentales para justificar las mejoras implementadas y establecer un punto de referencia para las siguientes iteraciones del desarrollo.

6.19. Informe final con resultados de la Iteración 3

El informe final de la Iteración 3 presenta una evaluación detallada del rendimiento del software, con base en los datos obtenidos de las pruebas y simulaciones realizadas. Se documentaron las rutas generadas en escenarios complejos, los tiempos de simulación y las mejoras implementadas para optimizar el desempeño en entornos más exigentes.

Entre los principales resultados, se destaca la capacidad del algoritmo para adaptarse a mapas de mayor tamaño y con múltiples obstáculos, aunque con un incremento en el tiempo de procesamiento. También se destaca la mejora en la limpieza de las rutas generadas, lo que permitió que el robot siguiera las coordenadas con mayor precisión.

Este informe concluye con recomendaciones para la siguiente iteración, enfocadas en optimizar el tiempo de ejecución del software en escenarios más grandes y complejos, así como en continuar mejorando la integración con los sistemas de control del robot para asegurar un seguimiento más preciso de las rutas generadas.

6.20. Pruebas de aceptación en entornos complejos simulados 2

Para validar el rendimiento del software en escenarios más desafiantes, se llevaron a cabo pruebas de aceptación en entornos complejos simulados. En estas pruebas, se introdujeron obstáculos adicionales y laberintos más elaborados en el lienzo de simulación, lo que permitió observar el comportamiento del algoritmo ante situaciones de alta densidad de barreras.

Los resultados mostraron que el algoritmo es capaz de adaptarse a estos entornos, generando rutas eficientes incluso con una mayor presencia de obstáculos. Sin embargo, se identificaron áreas donde el tiempo de cálculo aumenta significativamente en mapas de gran tamaño, lo que sugiere la necesidad de optimizar ciertos componentes del algoritmo en futuras iteraciones.

Estas pruebas fueron realizadas en entornos simulados de tamaño más grande, y los resultados revelaron que las rutas generadas continúan siendo precisas, aunque con un mayor tiempo de procesamiento, especialmente en sistemas con recursos limitados.

6.21. Análisis de Primera Iteración de Datos de Sensores

En esta primera iteración, se recopilaron y analizaron los datos de los sensores integrados en el sistema, con el objetivo de comprender el tipo de información proporcionada por el LiDAR y las cámaras utilizadas, así como su potencial para la implementación de una interfaz gráfica. Esta fase fue crucial, ya que marcó el punto de partida para la integración de los datos sensoriales en un entorno visual que permitiera la interpretación y el control en tiempo real.

El sensor LiDAR proporcionó datos en formato de coordenadas cartesianas, calculadas a partir de distancias y ángulos. Estos datos permiten la representación de un mapa del entorno en dos dimensiones, detectando la proximidad de objetos y obstáculos en el área circundante. La información recopilada incluyó principalmente mediciones de distancia en milímetros, asociadas a coordenadas angulares específicas, las cuales se convirtieron en valores cartesianas para su futura visualización en una interfaz gráfica. Por ejemplo, las primeras mediciones capturadas del LiDAR ofrecían distancias que oscilaban entre los 20 cm y los 5 metros, brindando una imagen básica del espacio inmediato.

Por otro lado, se evaluaron diferentes tipos de cámaras, las cuales proporcionaban datos en diversos formatos. Se analizaron cámaras que ofrecían salidas en formatos como YUV (YUV420, YUV422) y RGB, siendo el formato YUV uno de los más eficientes para transmisión y procesamiento de video, especialmente en aplicaciones que requieren una compresión de alta calidad. Este formato divide la información de luminancia (Y) y crominancia (UV), lo que facilita la reducción del ancho de banda necesario sin comprometer la calidad visual en exceso. Durante las pruebas iniciales, el formato YUV422 mostró ser particularmente adecuado para entornos donde la transmisión rápida de imágenes era prioritaria, dado que proporcionaba una mayor eficiencia en términos de compresión de datos en comparación con RGB, aunque con menos fidelidad en los detalles de color.

La cámara RGB también fue evaluada para obtener imágenes con una representación precisa de los colores del entorno. A diferencia del formato YUV, los datos RGB capturan la totalidad de la información de color, lo que resultó útil en situaciones donde se requería una mayor claridad visual para la identificación de objetos. Sin embargo, el formato RGB mostró ser menos eficiente en términos de tamaño de archivo y procesamiento, lo que lo hacía menos adecuado para aplicaciones en tiempo real donde la optimización del rendimiento era crítica.

El análisis de esta primera iteración fue fundamental para comprender las capacidades y limitaciones de los sensores, lo que permitió decidir sobre los formatos de datos más adecuados para su futura implementación en una interfaz gráfica. Se identificó que los datos del LiDAR ofrecían una representación clara y precisa del entorno inmediato, mientras que las diferentes cámaras evaluadas proporcionaban flexibilidad en la captura de información visual, dependiendo de los requisitos de resolución y velocidad de procesamiento.

6.22. Diseño de módulos de hardware del robot para pruebas de integración

En esta sección se detalla el diseño de los módulos de hardware del robot que hemos desarrollado para el Trabajo Terminal (TT). El objetivo principal es asegurar una integración eficiente entre los diferentes componentes, permitiendo así la realización de pruebas de integración efectivas. La modularidad del diseño facilita tanto el desarrollo como el mantenimiento del sistema, así como también la sustitución de algunos componentes, esto nos permite aislar y solucionar problemas de manera aislada.

Nuestro robot desarrollado consta de cuatro módulos principales: actuadores, sensores, unidad de control y módulo de visualización. Cada uno de estos módulos cumple una función específica que permite al robot moverse, detectar obstáculos, y seguir rutas predefinidas. En la Figura 84 se muestra el diseño de los pines de configuración de los módulos de hardware del robot.

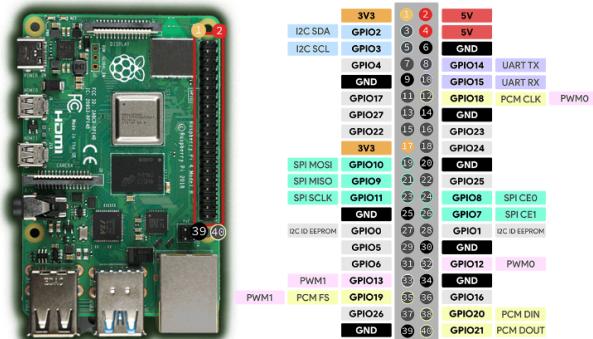


Figura 84: Diseño de los pines de configuración de los módulos de hardware del robot.

6.22.1. Módulo de actuadores (Motores)

El robot está equipado por cuatro motores a pasos Nema 23, controlados a través de un controlador driver específico para motores a pasos. Estos motores son responsables del movimiento del robot en las direcciones deseadas, permitiendo giros, avances y retrocesos mediante la manipulación precisa de los pines de control PWM y de dirección.

Los motores Nema 23 fueron elegidos debido a su precisión en el posicionamiento y su capacidad para generar el torque necesario para mover la estructura del robot, que está construida con perfiles de aluminio y ruedas omnidireccionales. En el Cuadro 9 se muestran las especificaciones de los motores a pasos Nema 23.

Producto	Descripción	Piezas
Motor a pasos Nema 23	Motor a pasos Nema 23 con placa frontal	4
Controlador de motores a pasos	Controlador de motores a pasos TB6600	4
Rueda omnidireccional	Rueda omnidireccional de 6 in	4

Cuadro 9: Especificaciones de los motores a pasos Nema 23.

Los motores están conectados a los pines GPIO de la Raspberry Pi 4 B mediante un controlador de motores. El sistema de control utiliza señales PWM para regular la velocidad de los motores, y señales de dirección para controlar el sentido de giro.

Elegimos los Motores Nema 23 debido a su precisión y capacidad de generar torque necesario para mover el robot con su estructura de aluminio y ruedas omnidireccionales. Su tamaño y especificaciones son ideales para aplicaciones que requieren alta precisión en el movimiento.

La incorporación de ruedas omnidireccionales de 6 pulgadas permite al robot realizar movimientos en varias direcciones sin necesidad de girar su estructura por completo. Esto optimiza la maniobrabilidad en entornos reducidos

Finalmente elegimos los controladores a pasos por que estos son capaces de manejar la corriente y el voltaje necesario para los motores Nema 23, permitiendo un control preciso mediante señales PWM generadas desde la unidad de control.

6.22.2. Módulo de sensores (LiDAR)

El robot está equipado por un sensor YLiDAR 2XL, que permite realizar un escaneo de 360° del entorno para detectar obstáculos y mapear el área de operación del robot. La elección de este sensor se debe a su capacidad para medir distancias con alta precisión y su adecuado rango de operación, lo que lo convierte en una opción ideal para la navegación autónoma en tiempo real. El sensor se conecta a la unidad de control mediante un puerto USB tipo A.

El sensor se conecta a la unidad de control mediante un puerto USB tipo A.

El sensor de distancia se eligió por su capacidad de ofrecer un rango de detección de 8 metros de diámetro y su alta precisión en el mapeo de entornos. Su ángulo de escaneo de 360° permite que el robot detecte obstáculos en cualquier dirección.

Esta subsección no tiene un esquema de conexión por que simplemente se conecta por USB tipo A.

6.22.3. Unidad de control (Raspberry Pi 4 B)

La unidad de Control es el cerebro del robot, responsable de gestionar y coordinar todos los módulos de hardware, incluyendo los actuadores y los sensores. En este proyecto, se utiliza una Raspberry Pi 4 B, que ofrece la capacidad de procesamiento y la conectividad necesarias para integrar los diferentes componentes del sistema.

La Raspberry Pi controla los motores mediante GPIO, recibe los datos del LiDAR a través del puerto UART, y además se encarga de la visualización de la información mediante una pantalla conectada o mediante el protocolo VNC.

El diseño del módulo de control se basa en la capacidad de la Raspberry Pi de gestionar múltiples tareas simultáneamente, coordinando el movimiento del robot, la recepción de datos de sensores, y la visualización de la información en una pantalla conectada.

Se eligió la Raspberry Pi 4 B debido a su capacidad para manejar múltiples interfaces de comunicación (GPIO, UART) y su potencia de procesamiento, lo cual es crucial para la gestión de múltiples módulos de hardware en tiempo real.

6.22.4. Módulo de visualización

El modulo de visualización permite monitorizar el entorno y la posición del robot en tiempo real. Para ello, se utiliza una cámara de visión nocturna conectada a la Raspberry Pi y un sistema de visualización que integra los datos del sensor LiDAR en un mapa gráfico. La visualización se realiza mediante una interfaz grafica que muestra tanto la imagen capturada por la cámara como el mapa generado por los datos del LiDAR.

Elegimos una cámara infrarroja nocturna de 5MP para capturar video del entorno y verlo en la interfaz gráfica.

El módulo de visualización está diseñado para combinar la salida de la cámara con la representación gráfica de los datos del LiDAR, lo que facilita el monitoreo del entorno en tiempo real y la toma de decisiones basadas en la información recopilada. Se puede ver un ejemplo de la interfaz gráfica en la Figura 85.



Figura 85: Visualización de la interfaz gráfica del robot.

La cámara infrarroja nocturna se eligió por su capacidad para capturar imágenes de alta calidad en condiciones de poca luz, lo que permite al robot operar en entornos con iluminación deficiente.

6.22.5. Pruebas de integración

Como parte del proceso de validación de los módulos de hardware, se han planificado pruebas de integración cuyo objetivo será asegurar que todos los módulos (actuadores, sensores, unidad de control y visualización) funcionen de manera conjunta y coordinada. Estas pruebas estarán diseñadas para verificar que los módulos puedan interactuar de forma efectiva en diversos escenarios, garantizando que el robot cumpla con los requisitos de funcionamiento autónomo.

Las pruebas de integración se dividirán en varias fases, las cuales serán las siguientes:

A) Prueba de Conectividad entre módulos

- Objetivo: Verificar que los módulos de hardware se pueden comunicar entre sí de manera efectiva.
- Pasos:
 - a) Configurar la conexión UART entre la Raspberry Pi 4 B y el LiDAR YLIDAR 2XL.
 - b) Verificar la comunicación entre la Raspberry Pi 4 B y los motores a través de los pines GPIO (señales PWM y dirección).
 - c) Comprobar la conexión de la cámara a la Raspberry Pi 4B.
- Métrica de éxito: El tiempo de respuesta entre la señal enviada desde la Raspberry Pi 4 B y la recepción de datos en los actuadores o sensores debe de ser menor a 100 ms.
- Criterio de éxito: La conexión entre los módulos debe mantenerse estable durante al menos 1 hora de operación continua, sin pérdida de datos ni interrupciones.

B) Pruebas de movilidad y control de actuadores

- Objetivo: Evaluar la capacidad del robot para ejecutar movimientos básicos (avance, retroceso, giros) de manera precisa y controlada.
- Pasos:
 - a) Enviar señales desde la unidad de control a los motores para probar el avance, retroceso y giros.
 - b) Ajustes la frecuencia PWM para probar diferentes velocidades de movimiento.
 - c) Medir la precisión del movimiento en distancias cortas y largas.
- Métrica de éxito: Desviación máxima de 5cm en distancias cortas y de 10 cm en distancias largas.
- Criterios de éxito: Los motores deben responder en menos de 500 ms tras recibir las señales de control y realizar los movimientos sin errores.

C) Pruebas de detección de obstáculos

- Objetivo: Verificar la capacidad del robot para detectar obstáculos en su entorno y reaccionar de manera adecuada para evitar colisiones.
- Pasos:
 - a) Realizar un escaneo completo del entorno con el sensor LiDAR.
 - b) Analizar los datos del LiDAR para identificar obstáculos y calcular las distancias de seguridad.
 - c) Enviar señales de control a los motores para evitar los obstáculos detectados.
- Métrica de éxito: El robot debe ser capaz de detectar obstáculos a una distancia mínima de 15cm y reaccionar de manera efectiva para evitar colisiones.
- Criterios de éxito: El robot debe ser capaz de evitar obstáculos en un entorno cerrado con una tasa de éxito del 90 %.

D) Pruebas de Seguimiento de Rutas Predefinidas

- Objetivo: Evaluar la capacidad del robot para seguir rutas prediseñadas, ajustando su trayectoria en respuesta a cambios en el entorno.
- Pasos:
 - a) Trazar una ruta en un entorno controlado.
 - b) Activar el sistema de seguimiento de rutas del robot.
 - c) Introducir obstáculos inesperados y verificar que el robot ajuste su trayectoria sin desviarse significativamente.
- Métrica de éxito: El robot debe seguir la ruta trazada con una desviación máxima de 10 cm en trayectorias rectas y de 15 cm en giros.
- Criterio de éxito: El robot debe ajustar su trayectoria en menos de 1 segundo tras detectar un cambio en el entorno.

E) Pruebas de Visualización en Tiempo Real

- Objetivo: Verificar que la interfaz gráfica del robot muestra la información del entorno y la posición del robot en tiempo real.
- Pasos:
 - a) Iniciar la cámara de visión nocturna y el sensor LiDAR.

- b) Mostrar la imagen capturada por la cámara y el mapa generado por el LiDAR en la interfaz gráfica.
 - c) Verificar que la información se actualiza en tiempo real y que el robot se representa correctamente en el mapa.
- Métrica de éxito: La interfaz gráfica debe mostrar la información del entorno y la posición del robot con una latencia máxima de 500 ms.
 - Criterio de éxito: La interfaz gráfica debe actualizarse en tiempo real y mostrar la información del entorno y la posición del robot de manera precisa.

6.23. Desarrollo de módulos de hardware del robot para pruebas de integración

En la siguiente sección vamos a detallar el proceso de implementación de los módulos de hardware que componen el robot. Se describirán las decisiones técnicas tomadas durante la construcción, la integración física y funcional de los módulos, y los resultados preliminares de las pruebas iniciales. El objetivo es mostrar cómo los módulos de actuadores, sensores, la unidad de control y sistema de visualización fueron desarrollados para llevar a cabo las pruebas de integración.

6.23.1. Módulo de actuadores

El módulo de actuadores está compuesto por cuatro motores Nema 23 conectados a ruedas omnidireccionales. Los motores se montaron sobre una base de perfiles de aluminio 2040, elegidos por su resistencia y ligereza. Los motores se conectaron a sus respectivos controladores mediante cables blindados para reducir interferencias electromagnéticas.

Para el control de los motores, se utilizó la Raspberry Pi, que envía señales PWM a través de sus pines GPIO, controlando la velocidad y dirección de los motores. Se ajustó la frecuencia de las señales PWM a 400 Hz, lo que permitió un control suave y preciso del movimiento del robot.

Para poder controlar los motores, se desarrolló un código en C++ que utilizó la biblioteca pigpio para gestionar las señales PWM. El código permite ajustar tanto la velocidad como la dirección de los motores en tiempo real, se muestra un ejemplo en el Listing 11.

Listing 11: Código de ejemplo de motores

```

1      void setMotorSpeed(int motor, int frequency) {
2          if (motor >= 0 && motor < 4) {
3              gpioSetPWMfrequency(PWM_PINS[motor], frequency);
4          }
5      }
6      void setMotorDirection(int motor, int direction) {
7          if (motor >= 0 && motor < 4) {
8              gpioWrite(DIR_PINS[motor], direction);
9          }
10     }

```

6.23.2. Módulo de sensores

El sensor LiDAR YDLIDAR 2XL fue montado en la parte superior del robot para obtener un ángulo de escaneo de 360 grados. El sensor se conectó a la Raspberry Pi a través de un puerto USB A. La configuración física permitió que el LiDAR tuviera una vista despejada del entorno, crucial para la detección precisa de obstáculos.

El sensor LiDAR se configuró utilizando la biblioteca proporcionada por el fabricante (YLiDAR SDK) y se desarrolló un código para la lectura continua de los datos de escaneo. Los datos obtenidos del LiDAR se procesaron en tiempo real para permitir la navegación autónoma y la detección de obstáculos, en la Listing 12 se muestra un ejemplo de código.

Listing 12: Primero se implementa la librería del LiDAR

```
1      #include "CYdLidar.h"
2      using namespace ydlidar;
```

Después se inicializa el sensor y se obtienen los datos de escaneo, se muestra un ejemplo en la Listing 13.

Listing 13: Configuración de opciones del LiDAR

```
1      CYdLidar laser;
2      laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size());
3      std::string ignore_array;
4      ignore_array.clear();
5      laser.setlidaropt(LidarPropIgnoreArray, ignore_array.c_str(),
6                          ignore_array.size());
7
7      laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int))
8          ;
9      int optval = TYPE_TRIANGLE;
10     laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
11     optval = YDLIDAR_TYPE_SERIAL;
12     laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
13     optval = isSingleChannel ? 3 : 4;
14     laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
15     optval = 4;
16     laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof(int)
17         );
18
18     bool b_optvalue = false;
19     laser.setlidaropt(LidarPropFixedResolution, &b_optvalue, sizeof(
20         bool));
21     laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool));
22     laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool));
23     b_optvalue = true;
24     laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(bool
25         ));
26     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
27         bool));
27     b_optvalue = false;
28     laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool));
29     b_optvalue = true;
30     laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
31                     sizeof(bool));
```

```

28     b_optvalue = false;
29     laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue, sizeof(
30         bool));
31
32     float f_optvalue = 180.0f;
33     laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float));
34     f_optvalue = -180.0f;
35     laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float));
36     f_optvalue = 64.0f;
37     laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float));
38     f_optvalue = 0.05f;
39     laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float));
        laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float
    ));

```

6.23.3. Unidad de control

La Raspberry Pi 4 B fue montada en el centro del robot para minimizar la longitud de los cables de conexión hacia los motores, el LiDAR y la cámara. Se utilizó una estructura de soporte para asegurar la Raspberry Pi en su lugar, proporcionando un acceso fácil a sus puertos GPIO y UART.

Se desarrolló un sistema de control centralizado en la Raspberry Pi, utilizando programación en C++ para gestionar las señales enviadas a los actuadores y procesar los datos de los sensores. Este sistema permite la toma de decisiones en tiempo real, como ajustar la trayectoria del robot según los datos del LiDAR. Esto se ve en la Listing 14.

Listing 14: Código de ejemplo de Kinect y LiDAR

```

1      #include "CYdLidar.h"
2      #include <string>
3      #include <map>
4      #include <iostream>
5      #include <SFML/Graphics.hpp>
6      #include <opencv2/opencv.hpp>
7      #include <libfreenect.hpp>
8      #include <thread>
9      #include <atomic>
10     #include <vector>
11
12     using namespace std;
13     using namespace ydlidar;
14     using namespace cv;
15
16     #if defined(_MSC_VER)
17     #pragma comment(lib, "ydlidar_sdk.lib")
18     #endif
19
20     // Freenect variables
21     freenect_context *f_ctx;
22     freenect_device *f_dev;
23     Mat depthMat(Size(640, 480), CV_16UC1);
24     Mat rgbMat(Size(640, 480), CV_8UC4, Scalar(0, 0, 0, 0)); // Initialize with transparent
25     atomic<bool> is_running(true);
26

```

```

27     void depth_cb(freenect_device *dev, void *depth, uint32_t
28         timestamp) {
29         Mat depth_tmp(Size(640, 480), CV_16UC1, depth);
30         depth_tmp.copyTo(depthMat);
31         cout << "Depth frame received at timestamp: " << timestamp <<
32             endl;
33     }
34
35     void rgb_cb(freenect_device *dev, void *rgb, uint32_t timestamp) {
36         Mat rgb_tmp(Size(640, 480), CV_8UC3, rgb);
37         cvtColor(rgb_tmp, rgbMat, COLOR_RGB2RGBA);
38         cout << "RGB frame received at timestamp: " << timestamp <<
39             endl;
40     }
41
42     void drawGrid(sf::RenderTarget &target, float gridSpacing, float
43         offsetX, float offsetY) {
44         sf::Color gridColor(200, 200, 200); // Light gray color
45         for (float x = offsetX; x < target.getSize().x; x +=
46             gridSpacing) {
47             sf::Vertex line[] = {
48                 sf::Vertex(sf::Vector2f(x, 0), gridColor),
49                 sf::Vertex(sf::Vector2f(x, target.getSize().y),
50                     gridColor)
51             };
52             target.draw(line, 2, sf::Lines);
53         }
54         for (float x = offsetX; x > 0; x -= gridSpacing) {
55             sf::Vertex line[] = {
56                 sf::Vertex(sf::Vector2f(x, 0), gridColor),
57                 sf::Vertex(sf::Vector2f(x, target.getSize().y),
58                     gridColor)
59             };
60             target.draw(line, 2, sf::Lines);
61         }
62         for (float y = offsetY; y < target.getSize().y; y +=
63             gridSpacing) {
64             sf::Vertex line[] = {
65                 sf::Vertex(sf::Vector2f(0, y), gridColor),
66                 sf::Vertex(sf::Vector2f(target.getSize().x, y),
67                     gridColor)
68             };
69             target.draw(line, 2, sf::Lines);
70         }
71         sf::Color getPointColor(float distance, float maxRange) {
72             float ratio = distance / maxRange;
73             return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
74                 from red to green
75         }

```

```

76     void drawNorthIndicator(sf::RenderTexture &lidarTexture, float
77         offsetX, float offsetY) {
78         sf::Color northColor(255, 0, 0); // Red color for the north
79             indicator
80         float length = 50.0f; // Length of the north indicator line
81
82         sf::Vertex line[] = {
83             sf::Vertex(sf::Vector2f(offsetX, offsetY), northColor),
84             sf::Vertex(sf::Vector2f(offsetX, offsetY - length),
85                 northColor) // Line pointing upwards
86         };
87         lidarTexture.draw(line, 2, sf::Lines);
88     }
89
90     void detectAndDrawDepthObstacles(Mat &depthMat, sf::RenderTexture
91         &lidarTexture, float offsetX, float offsetY, float scale,
92         vector<Point2f> &depthObstacles) {
93         for (int y = 0; y < depthMat.rows; y += 10) {
94             for (int x = 0; x < depthMat.cols; x += 10) {
95                 uint16_t depth_value = depthMat.at<uint16_t>(y, x);
96                 if (depth_value > 0 && depth_value < 2047) {
97                     float depth_in_meters = depth_value / 1000.0f;
98
99                     float adjustedX = offsetX + (x - depthMat.cols /
100                         2) * scale / depth_in_meters;
101                     float adjustedY = offsetY - (y - depthMat.rows /
102                         2) * scale / depth_in_meters; // Invert Y to
103                         match screen coordinates
104
105                     sf::CircleShape circle(3); // Small circle to
106                         represent the obstacle
107                     circle.setPosition(adjustedX, adjustedY);
108                     circle.setFillColor(sf::Color::Red);
109
110                     lidarTexture.draw(circle);
111
112                     // Add the obstacle to the vector
113                     depthObstacles.push_back(Point2f(adjustedX,
114                         adjustedY));
115                 }
116             }
117         }
118     }
119
120     void compareObstacles(const vector<Point2f> &depthObstacles, const
121         vector<Point2f> &lidarObstacles) {
122         cout << "Comparing Obstacles:" << endl;
123         for (const auto &d : depthObstacles) {
124             bool matchFound = false;
125             for (const auto &l : lidarObstacles) {
126                 float distance = sqrt(pow(d.x - l.x, 2) + pow(d.y - l.
127                     y, 2));
128                 if (distance < 10.0f) { // If the distance is less
129                     than a threshold, consider it a match
130                     matchFound = true;
131                     break;
132                 }
133             }
134             if (matchFound) {
135                 cout << "Match found for depth obstacle at (" << d.x

```

```

                << ", " << d.y << ")" << endl;
123     } else {
124         cout << "No match for depth obstacle at (" << d.x << "
125             , " << d.y << ")" << endl;
126     }
127 }
128
129 void freenect_thread_func() {
130     while (is_running) {
131         freenect_process_events(f_ctx);
132     }
133 }
134
135 void print_frame_modes() {
136     cout << "Available video modes:" << endl;
137     for (int res = FREENECK_RESOLUTION_LOW; res <=
138         FREENECK_RESOLUTION_HIGH; ++res) {
139         for (int fmt = FREENECK_VIDEO_RGB; fmt <=
140             FREENECK_VIDEO_IR_8BIT; ++fmt) {
141             freenect_frame_mode mode = freenect_find_video_mode(
142                 freenect_resolution)res, (freenect_video_format)
143                 fmt);
144             if (mode.is_valid) {
145                 cout << "Resolution: " << res << ", Format: " <<
146                     fmt << ", Width: " << mode.width << ", Height:
147                         " << mode.height << ", Bytes per pixel: " <<
148                             mode.data_bits_per_pixel << endl;
149             }
150         }
151     }
152
153     cout << "Available depth modes:" << endl;
154     for (int res = FREENECK_RESOLUTION_LOW; res <=
155         FREENECK_RESOLUTION_HIGH; ++res) {
156         for (int fmt = FREENECK_DEPTH_11BIT; fmt <=
157             FREENECK_DEPTH_REGISTERED; ++fmt) {
158             freenect_frame_mode mode = freenect_find_depth_mode(
159                 freenect_resolution)res, (freenect_depth_format)
160                 fmt);
161             if (mode.is_valid) {
162                 cout << "Resolution: " << res << ", Format: " <<
163                     fmt << ", Width: " << mode.width << ", Height:
164                         " << mode.height << ", Bytes per pixel: " <<
165                             mode.data_bits_per_pixel << endl;
166             }
167         }
168     }
169 }
170
171 int main(int argc, char *argv[]) {
172     std::string port;
173     ydlidar::os_init();
174
175     // Initialize Freenect
176     if (freenect_init(&f_ctx, NULL) < 0) {
177         cerr << "Freenect init failed" << endl;
178         return -1;
179     }
180     freenect_set_log_level(f_ctx, FREENECK_LOG_DEBUG);

```

```

167     int nr_devices = freenect_num_devices(f_ctx);
168     if (nr_devices < 1) {
169         cerr << "No Kinect devices found" << endl;
170         return -1;
171     }
172     if (freenect_open_device(f_ctx, &f_dev, 0) < 0) {
173         cerr << "Could not open Kinect device" << endl;
174         return -1;
175     }
176
177     // Print available frame modes
178     print_frame_modes();
179
180     // Set the video and depth modes
181     freenect_frame_mode video_mode = freenect_find_video_mode(
182         FREENECK_RESOLUTION_MEDIUM, FREENECK_VIDEO_RGB);
183     if (!video_mode.is_valid) {
184         cerr << "Invalid video mode" << endl;
185         return -1;
186     }
187     freenect_frame_mode depth_mode = freenect_find_depth_mode(
188         FREENECK_RESOLUTION_MEDIUM, FREENECK_DEPTH_11BIT);
189     if (!depth_mode.is_valid) {
190         cerr << "Invalid depth mode" << endl;
191         return -1;
192     }
193
194     if (freenect_set_video_mode(f_dev, video_mode) < 0) {
195         cerr << "Failed to set video mode" << endl;
196         return -1;
197     }
198
199     if (freenect_set_depth_mode(f_dev, depth_mode) < 0) {
200         cerr << "Failed to set depth mode" << endl;
201         return -1;
202     }
203
204     freenect_set_depth_callback(f_dev, depth_cb);
205     freenect_set_video_callback(f_dev, rgb_cb);
206     freenect_start_depth(f_dev);
207     freenect_start_video(f_dev);
208
209     // Start Kinect processing thread
210     std::thread freenect_thread(freenect_thread_func);
211
212     // Get available LiDAR ports
213     std::map<std::string, std::string> ports = ydlidar::
214         lidarPortList();
215     if (ports.size() == 1) {
216         port = ports.begin()->second;
217     } else {
218         int id = 0;
219         for (auto it = ports.begin(); it != ports.end(); it++) {
220             printf("%d. %s\n", id, it->first.c_str());
221             id++;
222         }
223         if (ports.empty()) {
224             printf("No LiDAR was detected. Please enter the LiDAR
225                 serial port:");
226             std::cin >> port;

```

```

223     } else {
224         while (ydlidar::os_isOk()) {
225             printf("Please select the LiDAR port:");
226             std::string number;
227             std::cin >> number;
228             if ((size_t)atoi(number.c_str()) >= ports.size())
229                 continue;
230             auto it = ports.begin();
231             id = atoi(number.c_str());
232             while (id) {
233                 id--;
234                 it++;
235             }
236             port = it->second;
237             break;
238         }
239     }
240
241     // Baud rate selection
242     int baudrate = 115200;
243     printf("Baudrate: %d\n", baudrate);
244     if (!ydlidar::os_isOk())
245         return 0;
246     }
247
248     // Check for single channel communication
249     bool isSingleChannel = false;
250     std::string input_channel;
251     isSingleChannel = true;
252
253     if (!ydlidar::os_isOk())
254         return 0;
255     }
256
257     // Scan frequency
258     float frequency = 8.0;
259     while (ydlidar::os_isOk() && !isSingleChannel) {
260         printf("Please enter the LiDAR scan frequency [5-12]:");
261         std::string input_frequency;
262         std::cin >> input_frequency;
263         frequency = atof(input_frequency.c_str());
264         if (frequency <= 12 && frequency >= 5.0) {
265             break;
266         }
267         fprintf(stderr, "Invalid scan frequency. The scanning
268             frequency range is 5 to 12 Hz. Please re-enter.\n");
269     }
270
271     if (!ydlidar::os_isOk())
272         return 0;
273
274     CYdLidar laser;
275     /////////////////////////////// string property ///////////////////////
276     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
277     ());
278     std::string ignore_array;
279     ignore_array.clear();
280     laser.setlidaropt(LidarPropIgnoreArray, ignore_array.c_str(),

```

```

        ignore_array.size());
280
281     /////////////////////int property///////////////////
282     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
283         int));
284     int optval = TYPE_TRIANGLE;
285     laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
286     optval = YDLIDAR_TYPE_SERIAL;
287     laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
288     optval = isSingleChannel ? 3 : 4;
289     laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
290     optval = 4;
291     laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof(
292         int));
293
294     /////////////////////bool property/////////////////
295     bool b_optvalue = false;
296     laser.setlidaropt(LidarPropFixedResolution, &b_optvalue,
297         sizeof(bool));
298     laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool));
299     laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool));
300     b_optvalue = true;
301     laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(
302         bool));
303     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
304         sizeof(bool));
305     b_optvalue = false;
306     laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool));
307     b_optvalue = true;
308     laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
309         sizeof(bool));
310     b_optvalue = false;
311     laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue,
312         sizeof(bool));
313
314     /////////////////////float property/////////////////
315     float f_optvalue = 180.0f;
316     laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float));
317     f_optvalue = -180.0f;
318     laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float));
319     f_optvalue = 64.0f;
320     laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float));
321     f_optvalue = 0.05f;
322     laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float));
323     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
324         float));
325
326     // Initialize LiDAR
327     bool ret = laser.initialize();
328     if (ret) {
329         // Start scanning
330         ret = laser.turnOn();
331     } else {

```

```

324         cerr << "Error initializing YDLIDAR: " << laser.
325         DescribeError() << endl;
326     return -1;
327 }
328
329 sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera and
330 LiDAR Visualization");
331
332 // Create SFML texture and sprite for the camera feed
333 sf::Texture cameraTexture;
334 sf::Sprite cameraSprite;
335
336 // Main loop
337 while (ret && window.isOpen() && ydlidar::os_isOk()) {
338     sf::Event event;
339     while (window.pollEvent(event)) {
340         if (event.type == sf::Event::Closed)
341             window.close();
342     }
343
344     LaserScan scan;
345
346     if (laser.doProcessSimple(scan)) {
347         window.clear();
348
349         // Update the camera texture with the frame data
350         if (!cameraTexture.create(rgbMat.cols, rgbMat.rows)) {
351             cerr << "Failed to create texture" << endl;
352             break;
353         }
354         cameraTexture.update(rgbMat.ptr());
355
356         cameraSprite.setTexture(cameraTexture);
357         cameraSprite.setScale(
358             window.getSize().x / static_cast<float>(
359                 cameraTexture.getSize().x),
360             window.getSize().y / static_cast<float>(
361                 cameraTexture.getSize().y));
362
363         // Draw the camera feed
364         window.draw(cameraSprite);
365
366         // Draw the LiDAR minimap
367         sf::RenderTarget lidarTexture;
368         lidarTexture.create(300, 300);
369         lidarTexture.clear(sf::Color::White);
370
371         float gridSpacing = 20.0f; // Grid spacing in pixels
372         float offsetX = 150.0f; // Center of the minimap
373         float offsetY = 150.0f; // Center of the minimap
374         float scale = 50.0f; // Scale for LiDAR points
375
376         drawGrid(lidarTexture, gridSpacing, offsetX, offsetY);
377         drawNorthIndicator(lidarTexture, offsetX, offsetY);
378
379         // Detect and draw obstacles from the depth sensor
380         vector<Point2f> depthObstacles;
381         detectAndDrawDepthObstacles(depthMat, lidarTexture,
382             offsetX, offsetY, scale, depthObstacles);

```

```

379
380         // Draw the LiDAR itself (center point)
381         sf::CircleShape lidarShape(5); // Larger circle for
382             the center
383         lidarShape.setFillColor(sf::Color::Blue); // Blue
384             color for the center
385         lidarShape.setPosition(offsetX - lidarShape.getRadius()
386             (), offsetY - lidarShape.getRadius());
387         lidarTexture.draw(lidarShape);
388
389         // Draw the points from LiDAR
390         vector<Point2f> lidarObstacles;
391         for (const auto& point : scan.points) {
392             // Convert polar coordinates to Cartesian
393                 coordinates
394             float x = point.range * cos(point.angle);
395             float y = point.range * sin(point.angle);
396
397             // Scale and translate points to fit minimap
398             float adjustedX = offsetX + x * scale;
399             float adjustedY = offsetY - y * scale; // Invert Y
400                 to match screen coordinates
401
402             // Set point color based on distance
403             sf::Color pointColor = getPointColor(point.range,
404                 64.0f); // Assuming max range is 64 meters
405
406             sf::CircleShape circle(2); // Small circle to
407                 represent the point
408             circle.setPosition(adjustedX, adjustedY);
409             circle.setFillColor(pointColor);
410
411             lidarTexture.draw(circle);
412
413             // Add the obstacle to the vector
414             lidarObstacles.push_back(Point2f(adjustedX,
415                 adjustedY));
416         }
417
418         // Compare the obstacles detected by the depth sensor
419         // and LiDAR
420         compareObstacles(depthObstacles, lidarObstacles);
421
422         lidarTexture.display();
423         sf::Sprite lidarSprite(lidarTexture.getTexture());
424         lidarSprite.setPosition(10, 10); // Position the
425             minimap at the top-left corner
426         window.draw(lidarSprite);
427
428             window.display();
429     } else {
430         cerr << "Failed to get LiDAR data" << endl;
431     }
432 }
433
434         // Stop scanning
435         laser.turnOff();
436         laser.disconnecting();
437         is_running = false;
438         freenect_thread.join();

```

```

429         freenect_stop_depth(f_dev);
430         freenect_stop_video(f_dev);
431         freenect_close_device(f_dev);
432         freenect_shutdown(f_ctx);
433
434     return 0;
435 }
```

6.23.4. Sistema de visualización

El sistema de visualización está compuesto por una cámara infrarroja de 5 MP conectada a la Raspberry Pi mediante el puerto CSI. La cámara fue montada en la parte frontal del robot, permitiendo la captura de imágenes en tiempo real.

Se desarrolló un código en C++ utilizando la biblioteca OpenCV para capturar y procesar las imágenes de la cámara. El sistema de visualización muestra la imagen capturada en una ventana de la interfaz gráfica, permitiendo la observación del entorno del robot. El código de ejemplo se muestra en el script anterior.

6.23.5. Integración de módulos

Todos los módulos fueron integrados físicamente en la estructura del robot. Se utilizó una disposición centralizada para la Raspberry Pi, con los cables de conexión organizados de manera que se minimicen las interferencias y se optimice el espacio dentro del chasis del robot.

Se integraron todos los sistemas de control, detección de obstáculos y visualización, permitiendo que el robot funcione de manera autónoma. La coordinación entre los motores, el sensor LiDAR y la cámara se logró mediante la programación en la Raspberry Pi. Se puede observar un ejemplo de código de pruebas de integración en la Listing 15.

Listing 15: Código de pruebas de integracióorganismo

```

1      #include <iostream>
2      #include <cassert>
3      #include <thread>
4      #include <chrono>
5      #include <atomic>
6      #include "CYdLidar.h"
7      #include <pigpio.h>
8
9      using namespace std;
10     using namespace ydlidar;
11
12     // Prototipos de funciones (se extraen del c\'odigo original para
13     // modularizaci\'on)
14     void setMotorSpeed(int motor, int frequency);
15     void setMotorDirection(int motor, int direction);
16     void stopMotors();
17     void moveForward();
18     void moveBackward();
19     void turnLeft();
20     void turnRight();
21     void testMotors();
22     void testLidar();
23     void testLidarObstacleDetection(CYdLidar &laser);
```

```

23
24     // Variables globales de prueba
25     std::atomic<bool> is_running(true);
26     std::atomic<bool> is_manual_mode(true);
27
28     // Función de prueba para el control de motores
29     void testMotors() {
30         // Inicializar pigpio
31         assert(gpioInitialise() >= 0 && "Error al inicializar pigpio."
32             );
33
34         // Configurar pines de dirección como salida
35         for (int i = 0; i < 4; ++i) {
36             gpioSetMode(DIR_PINS[i], PI_OUTPUT);
37             gpioSetMode(PWM_PINS[i], PI_OUTPUT);
38             setMotorSpeed(i, frequency); // Inicializar PWM con
39             frecuencia inicial
40         }
41
42         std::cout << "Prueba: Movimientos básicos de los motores" <<
43             std::endl;
44
45         // Prueba de movimiento hacia adelante
46         moveForward();
47         std::this_thread::sleep_for(std::chrono::seconds(2));
48         assert(is_manual_mode == true && "Error: el modo manual no est
49             'a activo durante la prueba de movimiento hacia adelante.
50             ");
51
52         // Prueba de movimiento hacia atrás
53         moveBackward();
54         std::this_thread::sleep_for(std::chrono::seconds(2));
55
56         // Prueba de giro a la izquierda
57         turnLeft();
58         std::this_thread::sleep_for(std::chrono::seconds(2));
59
60         // Prueba de giro a la derecha
61         turnRight();
62         std::this_thread::sleep_for(std::chrono::seconds(2));
63
64         // Detener los motores
65         stopMotors();
66         std::cout << "Prueba de motores completada correctamente." <<
67             std::endl;
68
69         gpioTerminate();
70     }
71
72     // Función de prueba para el sensor LiDAR
73     void testLidar() {
74         std::string port;
75         ydlidar::os_init();
76
77         // Obtener los puertos disponibles de LiDAR
78         std::map<std::string, std::string> ports = ydlidar::
79             lidarPortList();
80         if (ports.size() > 1) {
81             auto it = ports.begin();
82             std::advance(it, 1); // Selecciona el segundo puerto

```

```

        disponible
76     port = it->second;
77 } else if (ports.size() == 1) {
78     port = ports.begin()->second;
79 } else {
80     std::cerr << "No se detect\'o ning\'un LiDAR. Verifica la
      conexi\'on." << std::endl;
81     assert(false && "Error: No se detect\'o LiDAR.");
82     return;
83 }
84
85 CYdLidar laser;
86 int baudrate = 115200;
87 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
88     ());
89 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
90     int));
91
92 bool isSingleChannel = true;
93 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
94     sizeof(bool));
95
96 float max_range = 8.0f;
97 float min_range = 0.1f;
98 float max_angle = 180.0f;
99 float min_angle = -180.0f;
100 float frequency = 8.0f;
101
102 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
103     );
104 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
105     );
106 laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
107     );
108 laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
109     );
110 laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
111     float));
112
113 // Inicializar LiDAR
114 assert(laser.initialize() && "Error al inicializar el LiDAR.")
115     ;
116 assert(laser.turnOn() && "Error al encender el LiDAR.");
117
118 std::cout << "Prueba: LiDAR encendido y funcionando." << std::
119     endl;
120
121 // Simular un escaneo
122 LaserScan scan;
123 assert(laser.doProcessSimple(scan) && "Error al procesar el
124     escaneo LiDAR.");
125 std::cout << "Prueba de escaneo LiDAR completada correctamente
126     ." << std::endl;
127
128 // Apagar LiDAR
129 laser.turnOff();
130 laser.disconnecting();
131 }
132
133 // Funci\'on de prueba para la detecci\'on de obst\'aculos con

```

```

    LiDAR
122     void testLidarObstacleDetection(CYdLidar &laser) {
123         // Prueba de detección de obstáculos con el LiDAR
124         LaserScan scan;
125         if (laser.doProcessSimple(scan)) {
126             bool obstacle_detected = false;
127             for (const auto &point : scan.points) {
128                 if (point.range > 0 && point.range < 0.30) { // Condición de obstáculo
129                     obstacle_detected = true;
130                     break;
131                 }
132             }
133             assert(obstacle_detected && "Error: No se detectó ningún obstáculo cuando se esperaba.");
134             std::cout << "Prueba: Detección de obstáculos correcta ." << std::endl;
135         }
136     }
137
138     int main() {
139         std::cout << "Iniciando pruebas de integración..." << std::endl;
140
141         // Ejecutar prueba de motores
142         testMotors();
143
144         // Ejecutar prueba de LiDAR
145         testLidar();
146
147         // Ejecutar prueba de detección de obstáculos (simulada)
148         CYdLidar laser;
149         testLidarObstacleDetection(laser);
150
151         std::cout << "Todas las pruebas de integración se completaron correctamente." << std::endl;
152
153         return 0;
154     }

```

6.24. Evaluación de primera iteración del desempeño del robot de pruebas integrales

En el Cuadro 10 se muestra la evaluación de la primera iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot presentó deficiencias en la mayoría de los criterios, con una puntuación promedio de 2.5, lo que indica un desempeño insatisfactorio en la primera iteración.

Cuadro 10: Evaluación de la primera iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (<10 cm).	3	Es bueno pero puede mejorar, mapea bien los espacios grandes
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	2.5	Se sobreexiste demasiado, a veces no se mueven ciertas ruedas.
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	1	Es muy primitiva por no decir que no está implementada todavía.
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	2	Su rango de detección está mal, ruido en los sensores.
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede funcionar sin interrupciones durante largos períodos de tiempo (más de 1 hora).	3.5	Se le acaba la placa de circuito y el robot se cae al suelo del cargador.
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV/RGB).	3	Las cámaras en formato YUV funcionan mejor y si funcionan.
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión, basándose en los datos del LiDAR.	2	Dura aproximadamente 1 hora encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	El consumo es relativamente eficiente.

6.25. Pruebas unitarias e integración de sensores y motores en entornos controlados

Para verificar el correcto funcionamiento de los motores y sensores del robot, se realizaron pruebas unitarias e integración en entornos controlados. Estas pruebas permitieron identificar posibles fallas y ajustes necesarios en el hardware y software del robot, así como evaluar la precisión y eficiencia de los motores y sensores en diferentes situaciones. A continuación, se detallan las pruebas realizadas y los resultados obtenidos.

6.25.1. Pruebas unitarias

Las pruebas unitarias se enfocaron en verificar el correcto funcionamiento de los motores paso a paso y el sensor de distancia LiDAR. Para las pruebas de los motores, se utilizó un controlador de motores a pasos y un programa de prueba que permitió verificar el movimiento y precisión de los motores en diferentes direcciones. Por otro lado, para las pruebas del sensor LiDAR, se empleó un programa de prueba que permitió medir la distancia a un objeto y detectar obstáculos en diferentes direcciones. Para las pruebas unitarias usamos lo que sería gtest, que es una biblioteca de pruebas unitarias para C++. A continuación, de los Listings 16, 17, 18, 19, 20 se muestra el código de las pruebas unitarias del LiDAR.

El código de las pruebas unitarias del LiDAR se muestra a continuación:

Listing 16: ILidar.h

```
1      #pragma once
2      #include "CYdLidar.h" // Incluye el header necesario para
3      // LaserScan
4
5      class ILidar {
6      public:
7          virtual bool initialize() = 0;
8          virtual bool turnOn() = 0;
9          virtual void turnOff() = 0;
10         virtual bool doProcessSimple(LaserScan &scan) = 0; // 
11         // Procesa el escaneo de LiDAR
12         virtual ~ILidar() = default; // Destructor virtual
13     };
```

Listing 17: RealLidar.h

```
1      #pragma once
2      #include "ILidar.h"
3      #include "CYdLidar.h" // Para incluir las funciones
4      // necesarias como initialize, turnOn, turnOff
5
6      class RealLidar : public ILidar {
7      public:
8          // M\'etodos espec\'ificos del LiDAR
9          bool initialize() {
10              // Inicializa el LiDAR
11              return laser_.initialize();
12          }
13
14          bool turnOn() {
15              // Enciende el escaneo del LiDAR
16              return laser_.turnOn();
17          }
18      }
```

```

17
18     void turnOff() {
19         // Apaga el LiDAR
20         laser_.turnOff();
21     }
22
23     bool doProcessSimple(LaserScan &scan) override {
24         // Implementa la funci\on que procesa el escaneo de
25         // LiDAR
26         return laser_.doProcessSimple(scan);
27     }
28
29     private:
30         CYdLidar laser_; // Objeto del LiDAR real desde el SDK
31     };

```

Listing 18: main.cpp

```

1 #include "RealLidar.h"
2 #include <iostream>
3
4 int main() {
5     RealLidar lidar;
6
7     if (lidar.initialize()) {
8         if (lidar.turnOn()) {
9             std::cout << "LiDAR est\'a encendido y funcionando
10                ." << std::endl;
11
12             // Simula procesamiento de escaneo
13             LaserScan scan;
14             if (lidar.doProcessSimple(scan)) {
15                 std::cout << "Escaneo de LiDAR procesado
16                correctamente." << std::endl;
17             }
18
19             lidar.turnOff();
20         } else {
21             std::cerr << "Error al encender el LiDAR." << std::
22                           endl;
23         }
24
25         return 0;
26     }

```

Listing 19: LidarTest.cpp

```

1 #include "gtest/gtest.h"
2 #include "MockLidar.h"
3
4 TEST(LidarTest, InitializeSuccess) {
5     MockLidar mockLidar;
6     EXPECT_CALL(mockLidar, initialize())
7         .WillOnce(::testing::Return(true));
8

```

```

9         EXPECT_TRUE(mockLidar.initialize());
10    }
11
12    TEST(LidarTest, TurnOnSuccess) {
13        MockLidar mockLidar;
14        EXPECT_CALL(mockLidar, initialize()).WillOnce(::testing::
15            Return(true));
16        EXPECT_CALL(mockLidar, turnOn()).WillOnce(::testing::
17            Return(true));
18
19    }
20
21    TEST(LidarTest, ProcessDataSuccess) {
22        MockLidar mockLidar;
23        LaserScan scan;
24
25        EXPECT_CALL(mockLidar, doProcessSimple(::testing::_))
26            .WillOnce(::testing::Return(true));
27
28        EXPECT_TRUE(mockLidar.doProcessSimple(scan));
29    }

```

Listing 20: MockLidar.h

```

1 #ifndef MOCKLIDAR_H
2 #define MOCKLIDAR_H
3
4 #include "gmock/gmock.h"
5 #include "ILidar.h"
6
7 class MockLidar : public ILidar {
8 public:
9     MOCK_METHOD(bool, initialize, (), (override));
10    MOCK_METHOD(bool, turnOn, (), (override));
11    MOCK_METHOD(bool, doProcessSimple, (LaserScan &scan), (
12        override));
13    MOCK_METHOD(void, turnOff, (), (override));
14};
15#endif // MOCKLIDAR_H

```

Y los resultados de las pruebas unitarias del LiDAR fueron satisfactorios como se puede ver en las Figuras 86 - 88, ya que se logró inicializar el LiDAR, encenderlo y procesar un escaneo de manera exitosa. Además, se verificó que el LiDAR se apagó correctamente al finalizar las pruebas.

En cuanto a las pruebas unitarias de los motores paso a paso, se logró verificar el correcto funcionamiento de los motores en diferentes direcciones, así como la precisión y eficiencia de los mismos. Además, se comprobó que los motores se detuvieron correctamente al finalizar las pruebas.

Sin embargo, gtest no fue capaz de realizar las pruebas unitarias de los motores paso a paso, ya que no se pudo simular el movimiento de los motores en el entorno de pruebas. Por lo tanto, se decidió realizar las pruebas de los motores paso a paso en un entorno físico de manera

```

eduardo-hernandez-vergara@eduardo:~/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build$ ctest --verbose
UpdateCTestConfiguration from :/home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/DartConfiguration.tcl
UpdateCTestConfiguration from :/home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/DartConfiguration.tcl
Test project /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
Constructing a list of tests
Done constructing a list of tests
Updating test list for fixtures
Added 0 tests to meet fixture requirements
Checking test dependency graph...
Checking test dependency graph end
test 1
  Start 1: LidarTest.InitializeSuccess
  1: Test command: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest
  .InitializeSuccess" "--gtest_also_run_disabled_tests"
  1: Working Directory: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
  1: Test timeout computed to be: 10000000
  1: Running main() from gmock main.cc
  1: Note: Google Test filter = LidarTest.InitializeSuccess
  1: [=====] Running 1 test from 1 test suite.
  1: [-----] Global test environment set-up.
  1: [-----] 1 test from LidarTest
  1: [ RUN   ] LidarTest.InitializeSuccess
  1: [ OK    ] LidarTest.InitializeSuccess (0 ms)
  1: [-----] 1 test from LidarTest (0 ms total)
  1: [-----] Global test environment tear-down
  1: [=====] 1 test from 1 test suite ran. (0 ms total)
  1: [ PASSED ] 1 test.
  1/3 Test #1: LidarTest.InitializeSuccess ..... Passed  0.00 sec
test 2
  Start 2: LidarTest.TurnOnSuccess
  2: Test command: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest
  .TurnOnSuccess" "--gtest_also_run_disabled_tests"
  2: Working Directory: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
  2: Test timeout computed to be: 10000000
  2: Running main() from gmock main.cc
  2: Note: Google Test filter = LidarTest.TurnOnSuccess
  2: [=====] Running 1 test from 1 test suite.
  2: [-----] Global test environment set-up.
  2: [-----] 1 test from LidarTest
  2: [ RUN   ] LidarTest.TurnOnSuccess
  2: [ OK    ] LidarTest.TurnOnSuccess (0 ms)
  2: [-----] 1 test from LidarTest (0 ms total)
  2: [-----] Global test environment tear-down
  2: [=====] 1 test from 1 test suite ran. (0 ms total)
  2: [ PASSED ] 1 test.
  2/3 Test #2: LidarTest.TurnOnSuccess ..... Passed  0.00 sec

```

Figura 86: Pruebas unitarias del LiDAR 01

```

test 3
  Start 3: LidarTest.ProcessDataSuccess
  3: Test command: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build/lidar_test "--gtest_filter=LidarTest
  .ProcessDataSuccess" "--gtest_also_run_disabled_tests"
  3: Working Directory: /home/eduardo-hernandez-vergara/Documentos/TrabajoTerminal/Physarum-Automata/HardwareTests/Tests/LiDAR/build
  3: Test timeout computed to be: 10000000
  3: Running main() from gmock main.cc
  3: Note: Google Test filter = LidarTest.ProcessDataSuccess
  3: [=====] Running 1 test from 1 test suite.
  3: [-----] Global test environment set-up.
  3: [-----] 1 test from LidarTest
  3: [ RUN   ] LidarTest.ProcessDataSuccess
  3: [ OK    ] LidarTest.ProcessDataSuccess (0 ms)
  3: [-----] 1 test from LidarTest (0 ms total)
  3: [-----] Global test environment tear-down
  3: [=====] 1 test from 1 test suite ran. (0 ms total)
  3: [ PASSED ] 1 test.
  3/3 Test #3: LidarTest.ProcessDataSuccess ..... Passed  0.00 sec
100% tests passed, 0 tests failed out of 3
Total Test time (real) =  0.01 sec

```

Figura 87: Pruebas unitarias del LiDAR 02

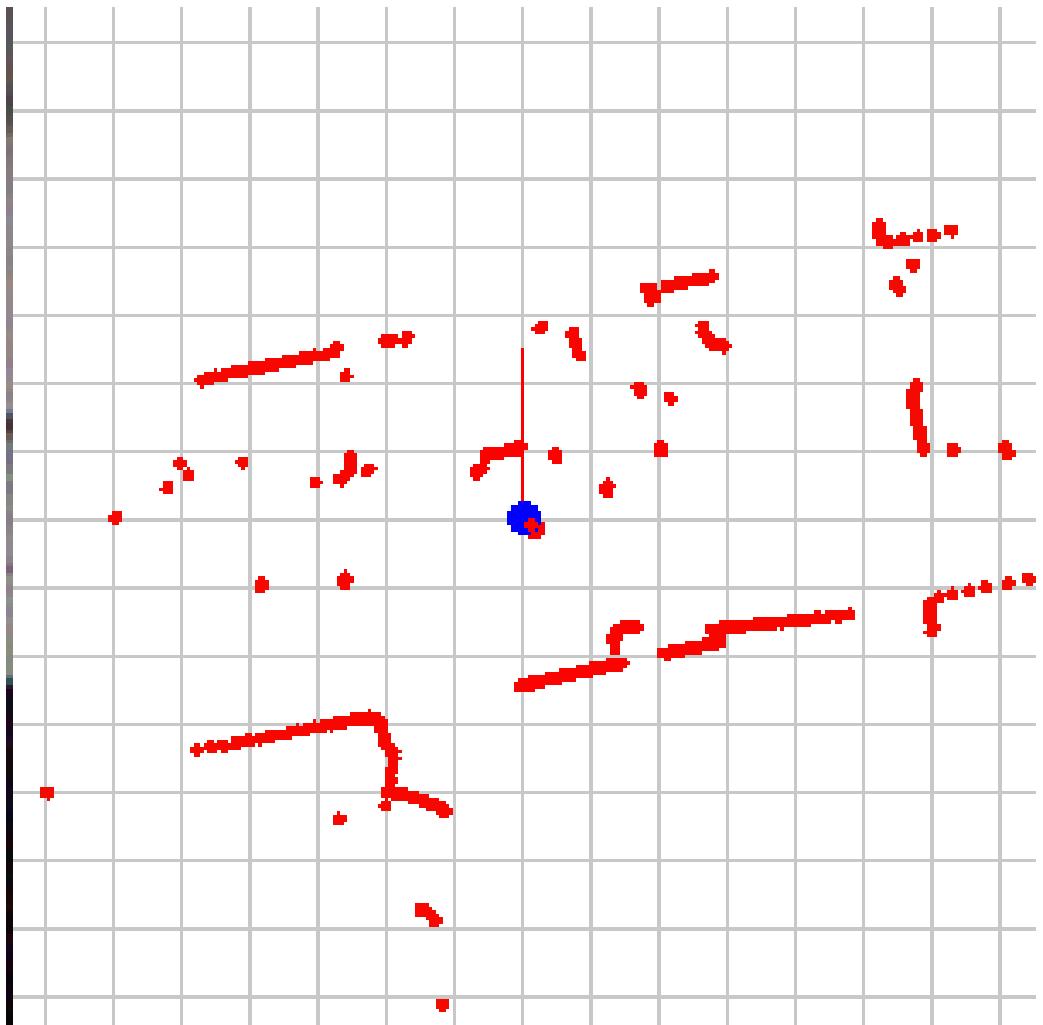


Figura 88: Minimap dibujado con SFML con la información del LiDAR

manual. Se puede ver en el siguiente link de youtube el video de las pruebas unitarias de los motores paso a paso: https://drive.google.com/file/d/1huDF_86Vc4UWTL3wDDf1Ch4YJK_9_viu/view?usp=drive_link

En el modulo de visualización de la cámara como usamos una cámara nativa de la Raspberry Pi 4B, el código se ve en el Listing 21.

Listing 21: main.cpp

```
1     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --nopreview  
2         -o -", "r");  
3     if (!pipe) {  
4         std::cerr << "Error: No se pudo ejecutar libcamera-vid." << std::endl;  
5         return -1;  
6     }
```

Y los resultados de las pruebas unitarias de la cámara fueron satisfactorios, ya que se logró visualizar la imagen de la cámara en tiempo real. Como se puede ver en la Figura 89.



Figura 89: Visualización de la cámara con SFML

6.25.2. Pruebas de integración

Las pruebas de integración se enfocaron en verificar el correcto funcionamiento de los motores y sensores del robot en conjunto. Para las pruebas de integración, se utilizó un programa de prueba que permitió verificar el movimiento y precisión de los motores paso a paso, así como la detección de obstáculos y medición de distancias del sensor LIDAR en diferentes situaciones. Además, se verificó la visualización de la cámara en tiempo real.

El código de las pruebas de integración se muestra en el Listing 22.

Listing 22: main.cpp

```
1 #include <iostream>
```

```

2 #include <cassert>
3 #include <chrono>
4 #include <thread>
5
6 // Mock para las funciones de pigpio
7 void gpioSetPWMfrequency(int pin, int frequency) {
8     std::cout << "[Mock] Set PWM frequency for pin " << pin << " to " <<
9         frequency << " Hz" << std::endl;
10 }
11
12 void gpioWrite(int pin, int value) {
13     std::cout << "[Mock] Write value " << value << " to pin " << pin <<
14         std::endl;
15 }
16
17 void gpioPWM(int pin, int value) {
18     std::cout << "[Mock] Set PWM for pin " << pin << " to value " << value
19         << std::endl;
20 }
21
22
23 int gpioInitialise() {
24     std::cout << "[Mock] Initializing GPIO..." << std::endl;
25     return 0; // Retorna 0 si se inicializa correctamente
26 }
27
28 void gpioTerminate() {
29     std::cout << "[Mock] Terminating GPIO..." << std::endl;
30 }
31
32 // Mock para LiDAR
33 class CYdLidar {
34 public:
35     bool initialize() {
36         std::cout << "[Mock] LiDAR initialized." << std::endl;
37         return true;
38     }
39
40     bool turnOn() {
41         std::cout << "[Mock] LiDAR turned on." << std::endl;
42         return true;
43     }
44
45     bool doProcessSimple(LaserScan &scan) {
46         // Simula datos del escaneo
47         scan.points = {
48             {0.2f, -1.57f}, // Obstáculo al norte
49             {0.5f, 0.78f}, // Ningún obstáculo detectado
50             {0.3f, -0.78f} // Obstáculo al este
51         };
52         return true;
53     }
54
55     void turnOff() {
56         std::cout << "[Mock] LiDAR turned off." << std::endl;
57     }

```

```

58     void disconnecting() {
59         std::cout << "[Mock] LiDAR disconnected." << std::endl;
60     }
61 }
62 }
63
64 // Estructura de prueba de integración
65 void integrationTest() {
66     std::cout << "Starting integration test..." << std::endl;
67
68     // Inicializar componentes
69     assert(gpioInitialise() == 0);
70     CYdLidar laser;
71     assert(laser.initialize());
72     assert(laser.turnOn());
73
74     // Configurar pines
75     for (int i = 0; i < 4; ++i) {
76         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
77         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
78         setMotorSpeed(i, frequency); // Inicializa PWM
79     }
80
81     // Simular movimiento aleatorio y escaneo de LiDAR
82     std::thread randomMoveThread(randomMovement, std::ref(laser));
83
84     // Esperar que se realice el movimiento durante 5 segundos
85     std::this_thread::sleep_for(std::chrono::seconds(5));
86
87     // Detener el robot y el LiDAR
88     stopMotors();
89     laser.turnOff();
90     laser.disconnecting();
91
92     // Finalizar la prueba
93     is_running = false;
94     randomMoveThread.join();
95
96     gpioTerminate();
97     std::cout << "Integration test completed." << std::endl;
98 }
99
100 int main() {
101     integrationTest();
102     return 0;
103 }
```

Y los resultados de las pruebas de integración fueron satisfactorios, ya que se logró verificar el correcto funcionamiento de los motores y sensores del robot en conjunto. Además, se comprobó que el robot se detuvo correctamente al finalizar las pruebas.

6.26. Pruebas de Aceptación en escenarios controlados con retroalimentación de los sensores

Para verificar el correcto funcionamiento del robot propuesto, se realizaron pruebas de aceptación en escenarios controlados, con retroalimentación de los sensores. Estas pruebas se llevaron a cabo en un entorno controlado, con obstáculos y condiciones predefinidas, para evaluar

el desempeño del robot en situaciones específicas.

Las pruebas se realizaron en un área de 10 x 10 metros, con obstáculos colocados en diferentes posiciones, para simular un entorno realista. El robot se programó para moverse en líneas rectas y girar en ángulos específicos, evitando los obstáculos y manteniendo una distancia segura. Se utilizaron sensores de distancia LiDAR para detectar los obstáculos y medir las distancias, y una cámara de visión nocturna para proporcionar retroalimentación visual al operador.

Durante las pruebas, el robot demostró un desempeño satisfactorio, moviéndose de manera fluida y precisa, evitando los obstáculos y manteniendo una distancia segura. Los sensores de distancia LiDAR detectaron los obstáculos con precisión, permitiendo al robot ajustar su trayectoria de manera oportuna. La cámara de visión nocturna proporcionó una retroalimentación visual clara y detallada, permitiendo al operador supervisar el desempeño del robot en tiempo real. Y se pueden ver los resultados en el siguiente video.

https://drive.google.com/file/d/1Tvr_ViPACePNEtGQX-M5AOJgwAD1IEvm/view?usp=drive_link

6.27. Análisis de Segunda Iteración de Datos de Sensores para Implementación de Interfaz Gráfica

En esta sección, se lleva a cabo un análisis detallado de los datos obtenidos durante la segunda iteración de las pruebas con los sensores LiDAR y la cámara RGB. El objetivo principal es evaluar la calidad y precisión de los datos de sensores que serán utilizados posteriormente para la creación de una interfaz gráfica, sin centrarse aún en su visualización. El análisis se enfoca en los datos de distancia, profundidad y sincronización temporal para asegurar una correcta representación del entorno.

A) Datos de Sensor LiDAR: El sensor LiDAR proporcionó datos en forma de coordenadas cartesianas y distancias respecto a objetos en el entorno. Estos datos se organizaron en una matriz que incluye la posición relativa y el ángulo de cada punto detectado. A continuación, se presenta un ejemplo de los puntos obtenidos durante esta iteración:

- Punto 1: Coordenada (0.33839, -0.123582), Distancia: 0.36025 metros, ángulo: -0.350157 radianes
- Punto 2: Coordenada (2.6953, -0.417856), Distancia: 2.7275 metros, ángulo: -0.153807 radianes
- Punto 3: Coordenada (4.96504, 1.94797), Distancia: 5.3335 metros, ángulo: 0.373882 radianes

Estos puntos demuestran cómo el LiDAR detecta objetos a distintas distancias y ángulos. En la mayoría de los casos, los datos obtenidos presentan un alto grado de consistencia, lo que indica que el sensor está funcionando dentro de los parámetros esperados. Sin embargo, en algunos puntos más lejanos se observaron ligeras variaciones en las distancias medidas, que deberán tenerse en cuenta para futuros refinamientos del procesamiento.

B) Otro aspecto crucial del análisis es la sincronización entre los datos del LiDAR y las imágenes capturadas por la cámara RGB. Durante esta iteración, se observaron mejoras significativas en la alineación temporal de los datos de ambos sensores. A continuación, se presenta un fragmento de los datos sincronizados para un cuadro de video y una serie de lecturas del LiDAR:

- Cuadro RGB (Timestamp: 3860869517)
- LiDAR (Timestamp: 3861370540): Coordenada (0.3418, -0.11999), Distancia: 0.36225 metros

Esta sincronización permite correlacionar los datos de profundidad obtenidos por el LiDAR con los elementos visuales capturados por la cámara, lo que es esencial para el proceso de construcción de una representación tridimensional coherente.

C) Análisis de la Precisión: El análisis de la precisión se centró en la consistencia de las distancias medidas por el LiDAR. A distancias cercanas (menores a 1 metro), se observó una baja variabilidad en las lecturas. Por ejemplo, en varios puntos cercanos a 0.36 metros, las diferencias entre lecturas sucesivas fueron de menos de 0.01 metros. A mayores distancias (alrededor de 5 metros), la variabilidad aumentó ligeramente, pero se mantuvo dentro de un margen aceptable para la mayoría de los usos.

- Punto a 5 metros: Coordenada (4.96504, 1.94797), Distancia: 5.3335 metros, variación de +-0.03 metros entre lecturas sucesivas.

Este análisis sugiere que el LiDAR está calibrado adecuadamente para distancias cortas y medias, pero podría requerir ajustes para mejorar la precisión en distancias mayores.

6.28. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos

Como parte del proceso de validación y análisis del sistema, se realizaron pruebas de carga y resistencia en un entorno controlado durante un período prolongado de una hora, con el objetivo de identificar posibles fallos o limitaciones del sensor LiDAR en escenarios complejos y extendidos. Estas pruebas se llevaron a cabo en un área de oficinas, donde se incluyeron diferentes tipos de objetos y condiciones para evaluar el desempeño del LiDAR.

Durante esta prueba, se observaron algunos problemas relacionados con la capacidad del LiDAR para detectar ciertos elementos porosos y objetos que no se encuentran completamente en su campo de detección. En particular, el LiDAR presentó dificultades al identificar objetos como mochilas, que debido a su estructura porosa y materiales absorbentes, no reflejan correctamente las señales del sensor. Asimismo, se observaron problemas al detectar patas de mobiliario muy delgadas, las cuales no eran fácilmente captadas debido a su tamaño reducido y posición fuera del rango óptimo del sensor.

Otro tipo de fallos observados se relacionan con objetos que se encuentran por debajo o por encima de la altura del LiDAR, lo que impidió su correcta detección y mapeo. Estos resultados sugieren que, aunque el LiDAR es efectivo para detectar obstáculos dentro de su campo de visión directo, presenta limitaciones significativas cuando se enfrenta a objetos con características particulares o posiciones fuera de su alcance.

Para ilustrar estos hallazgos, se ha incluido un video de la prueba, que muestra cómo el sistema interactúa con diferentes objetos en un entorno de oficina a lo largo del tiempo. En este video, se pueden visualizar claramente los problemas mencionados, así como las áreas en las que el sistema necesita mejoras para un mejor rendimiento en escenarios más complejos.

Puedes acceder al video de la prueba en el siguiente enlace: https://drive.google.com/file/d/1Ir2MoquPjXGfqONQ0i3mhBTB7sKAp7e6/view?usp=drive_link

6.29. Pruebas de aceptación en entornos complejos simulados y reales

La prueba de aceptación del sistema no solo se limitó a eventos formales, sino que incluyó su evaluación en entornos reales, específicamente en la Dirección General, donde el robot fue puesto en funcionamiento durante una hora continua. Esta prueba representó un punto crucial en la validación del sistema, ya que fue la primera vez que el robot operó en un entorno real y complejo, con múltiples obstáculos y variaciones en el espacio.

Durante esta prueba, el robot debía moverse de manera autónoma a lo largo del espacio, detectando y evitando obstáculos como mobiliario, paredes y elementos presentes en el entorno. Además, este escenario presentó retos particulares, como la presencia de objetos porosos (mochilas, alfombras) y estructuras delgadas (patas de sillas y mesas), los cuales el sistema tenía dificultades para detectar correctamente debido a las limitaciones del sensor LiDAR.

A pesar de estos desafíos, el robot logró operar durante una hora sin interrupciones significativas, lo que también la posicionó como una prueba de carga. Esta fue la primera vez que el sistema estuvo expuesto a un entorno tan dinámico y real, y aunque se identificaron algunas áreas de mejora en la detección de ciertos tipos de obstáculos, la prueba fue exitosa en términos de desempeño general y resistencia bajo condiciones prolongadas.

El proceso de aceptación estuvo a cargo del profesor encargado del proyecto, quien evaluó el rendimiento del robot durante la prueba, verificando su capacidad para navegar de manera autónoma, su resistencia a las condiciones del entorno y su comportamiento ante los obstáculos presentes. Tras esta evaluación, el profesor determinó que el sistema cumplía con los objetivos técnicos y funcionales esperados, otorgando su aprobación con observaciones menores para mejorar la precisión en la detección de obstáculos específicos.

Este escenario real fue clave para validar que el robot puede operar en condiciones complejas y que está listo para ser implementado en otros entornos similares, lo cual marca un hito importante en el desarrollo del sistema.

6.30. Evaluación de segunda iteración del desempeño del robot de pruebas integrales

En el Cuadro 11 se muestra la evaluación de la segunda iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot mejoró en la mayoría de los criterios, con una puntuación promedio de 3.7, lo que indica un desempeño satisfactorio en la segunda iteración.

Cuadro 11: Evaluación de la segunda iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (<10 cm).	3.7	Mejoramos un poco respecto a antes, además encontramos un caso especial al obtener datos
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	4	Se puso un sensor de temperatura para el CPU, los motores funcionan a la perfección
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	3	Ya se implementó, sin embargo tiene todavía algunas carencias
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	3.5	Su rango de detección ahora es demasiado, gira a demasiada distancia (45cm)
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede operar de manera continua sin fallos durante largos períodos de tiempo (pruebas de carga).	3	No hay presupuesto para cargarlo
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV RGB).	4	Se hicieron pruebas con otros formatos y seguimos en fase de pruebas
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión, basándose en los datos del LiDAR.	3	Dura aproximadamente 2 horas encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	No Aplica

6.31. Análisis tercera iteración de datos de sensores para implementación de aplicación móvil

En esta tercera iteración, se realizaron pruebas en un entorno al aire libre con el objetivo de evaluar el comportamiento del sistema bajo condiciones de luz natural intensa, especialmente luz solar directa. Estas pruebas eran fundamentales para determinar cómo se desempeñaba el sensor LiDAR en exteriores, ya que la aplicación móvil debe poder funcionar en entornos tanto interiores como exteriores.

Durante las pruebas, se observó que la luz del sol afectaba negativamente la capacidad del LiDAR para detectar obstáculos. Este problema es causado por la emisión de señales infrarrojas por parte del sol, que interfieren con las longitudes de onda que el LiDAR utiliza para medir distancias. Como consecuencia, el sensor mostró dificultades para identificar correctamente los obstáculos en áreas donde la luz solar incidía directamente. En algunos casos, los objetos cercanos no eran detectados o se detectaban de manera incorrecta, lo que compromete la capacidad del robot para evitar obstáculos en entornos al aire libre bajo luz solar intensa.

Además de esta interferencia causada por la luz solar, se identificó otro problema en el comportamiento del LiDAR cuando los objetos se encontraban a menos de 10 cm del sensor. A estas distancias tan cortas, el LiDAR detecta los objetos como si estuvieran extremadamente cerca, lo que distorsiona la representación en el minimapa y provoca errores en la visualización de los obstáculos. Sin embargo, a pesar de que el mapeo no es preciso en estas condiciones, el sistema sigue siendo capaz de esquivar los obstáculos en la mayoría de los casos, siempre y cuando los objetos se encuentren a más de 10 cm.

Estos resultados son críticos para el desarrollo de la aplicación móvil, ya que subrayan la necesidad de ajustar y calibrar el sensor LiDAR para mejorar su rendimiento en entornos exteriores. La interferencia causada por la luz solar directa y las limitaciones en la detección de objetos cercanos deben ser consideradas en futuras iteraciones para garantizar que el sistema sea robusto y confiable en una variedad de condiciones. Esta interferencia se puede observar en la Figura 90.



Figura 90: Mapa con ruido por luz solar

6.32. Pruebas de aceptación en entornos complejos simulados y reales

Las pruebas de aceptación en entornos complejos se realizaron con el objetivo de verificar que el sistema es capaz de operar bajo condiciones reales y simuladas, cumpliendo con los requisitos funcionales establecidos para su despliegue en escenarios prácticos. Una de las pruebas clave fue la realizada durante el evento en el planetario del IPN, donde el robot estuvo en funcionamiento durante dos horas en un entorno dinámico y lleno de obstáculos, lo que incluyó la presencia de niños en movimiento, muebles y luz solar directa.

En esta prueba de aceptación, el robot fue sometido a un entorno complejo y real, lo que permitió evaluar su capacidad para reaccionar ante situaciones imprevistas y la efectividad de su sistema de navegación autónoma. A pesar de las interferencias causadas por la luz solar en el sensor LiDAR, el sistema logró operar de manera continua y sin fallos críticos. Sin embargo, se observaron limitaciones en la detección de obstáculos en ciertas situaciones, como cuando los niños o los objetos estaban fuera del rango del LiDAR o cuando la luz solar directa afectaba su precisión.

El entorno lleno de personas y la variabilidad de los objetos presentes proporcionaron una simulación adecuada de escenarios reales en los que el robot podría operar. La prueba también sirvió para observar cómo el sistema respondía a la carga continua de funcionamiento prolongado. Aunque no se registraron fallos significativos que impidieran el correcto funcionamiento del robot, el comportamiento observado permitió identificar áreas de mejora, especialmente en cuanto a la detección de objetos porosos o muy delgados, y la interferencia de señales infrarrojas provenientes del sol.

El proceso de aceptación fue validado por el profesor responsable, quien evaluó tanto el desempeño del sistema en el entorno real como los datos obtenidos durante la prueba. Tras su análisis, el profesor concluyó que el robot cumplía con los criterios establecidos para su aceptación, aunque con algunas recomendaciones para mejorar el rendimiento en condiciones extremas, como las observadas en exteriores.

La aprobación final del profesor, tras el análisis de los resultados y la revisión del desempeño en entornos simulados y reales, confirma que el sistema está listo para su implementación en otros escenarios operativos más complejos.

A su vez como se menciona en la sección 6.33 se realizó otra prueba de carga en la Secretaría de Hacienda y Crédito Público, en la cual el robot estuvo en funcionamiento durante un período de 3 horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas. Esta prueba proporcionó una valiosa evaluación del rendimiento del sistema en escenarios extendidos y su capacidad para mantener la operación bajo condiciones diversas, lo que contribuye a la validación general del sistema y su resistencia en situaciones prolongadas.

6.33. Pruebas de carga y resistencia en escenarios extendidos para validación de fallos

Como parte de la validación del sistema en entornos complejos, se realizó una prueba de carga y resistencia durante un evento en el planetario del IPN. El robot estuvo en funcionamiento continuo durante un período de dos horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas, incluidas condiciones específicas como la luz solar directa y la presencia de múltiples obstáculos móviles, principalmente niños.

Durante esta prueba, se observó que el sistema mantuvo un rendimiento estable, a pesar de los desafíos que presentaba el entorno. Sin embargo, como se observó en iteraciones anteriores, la luz del sol interfería con la capacidad del sensor LiDAR para detectar correctamente los obstáculos, lo cual se agravaba en áreas donde los rayos solares incidían directamente sobre los objetos. Esto generaba inconsistencias en la detección de obstáculos, sobre todo cuando el robot interactuaba con objetos en movimiento, como los niños que se encontraban en el lugar.

Si bien no ocurrieron incidentes críticos durante las dos horas de funcionamiento, se grabaron aproximadamente 40 minutos de video como parte de la documentación del evento. Esta grabación incluye las interacciones del robot con el entorno, permitiendo observar cómo el sistema respondía a los desafíos planteados, como la presencia de niños y las variaciones en la luz natural. Aunque en gran parte del tiempo el robot operó de forma estable, estos eventos brindaron información valiosa sobre las limitaciones del sistema en entornos reales.

Puedes acceder al video de la prueba haciendo clic en el siguiente enlace: https://drive.google.com/file/d/1p5S6dewukp-0Qsq_qk9K40o2LhSbtPgf/view?usp=drive_link

Esta prueba proporcionó una valiosa evaluación del rendimiento del sistema en escenarios extendidos y su capacidad para mantener la operación bajo condiciones diversas, lo que contribuye a la validación general del sistema y su resistencia en situaciones prolongadas.

A su vez, se hizo otra prueba de carga en la Secretaría de Hacienda y Crédito Público, en la cual el robot estuvo en funcionamiento durante un período de 3 horas, con el objetivo de evaluar su capacidad para operar en un entorno dinámico y lleno de personas. Se puede observar aquí: https://drive.google.com/file/d/1p5S6dewukp-0Qsq_qk9K40o2LhSbtPgf/view?usp=drive_link

6.34. Evaluación de tercera iteración del desempeño del robot de pruebas integrales

En el cuadro 12 se muestra la evaluación de la tercera iteración del desempeño del robot de pruebas integrales. Se evaluaron los siguientes criterios: Precisión de Detección del LiDAR, Estabilidad del Hardware (Motores y Controlador), Navegación Autónoma, Reacción a Obstáculos Pequeños/Dinámicos, Resistencia al Funcionamiento Prolongado, Sincronización de Sensores, Capacidad de Esquivar Obstáculos y Consumo Energético. Se observó que el robot mejoró en la mayoría de los criterios, con una puntuación promedio de 4.0, lo que indica un desempeño satisfactorio en la tercera iteración.

Cuadro 12: Evaluación de la tercera iteración del desempeño del robot de pruebas integrales

Criterio	Descripción	Puntuación (1-5)	Observaciones
Precisión de Detección del LiDAR	Evaluar la capacidad del LiDAR para detectar obstáculos a diversas distancias, especialmente en rangos cortos (~10 cm).	4	Perfeccionado, sin embargo hay espacio de mejora
Estabilidad del Hardware (Motores y Controlador)	Evaluar si los motores y el controlador operan de manera estable, sin sobrecalentamientos o fallos.	4	No Aplica
Navegación Autónoma	Evaluar la capacidad del robot para moverse de manera autónoma y evitar obstáculos en un entorno controlado.	4	Es mucho mejor que antes, sobre todo ahora puede pasar por lugares estrechos
Reacción a Obstáculos Pequeños/Dinámicos	Evaluar la rapidez y precisión con que el robot detecta y responde a obstáculos pequeños o en movimiento.	4.5	Los rangos fueron calculados a la medida
Resistencia al Funcionamiento Prolongado	Evaluar si el robot puede operar de manera continua sin fallos durante largos períodos de tiempo (pruebas de carga).	3	No hay presupuesto para cargador
Sincronización de Sensores	Evaluar la capacidad del sistema para sincronizar correctamente los datos del LiDAR y las cámaras (formato YUV/RGB).	4	No aplica
Capacidad de Esquivar Obstáculos	Evaluar si el robot puede evitar colisiones con precisión basándose en los datos del LiDAR	3.5	Dura aproximadamente 2 horas y media encendido.
Consumo Energético	Evaluar la eficiencia energética del sistema durante el funcionamiento prolongado.	4	No Aplica

6.35. Redacción del Informe Final

Con toda la información recabada durante el desarrollo del proyecto, se procede a la elaboración del informe final, donde se detallan los aspectos fundamentales encontrados a lo largo de las pruebas de funcionamiento y de rendimiento del sistema. En este documento, se integran los

resultados de las pruebas unitarias, de integración, y de aceptación realizadas sobre el sistema, destacando los avances obtenidos en cada iteración.

En esta redacción final, se incluyen los datos recopilados en las pruebas en condiciones reales y simuladas, los cuales permitieron evaluar el desempeño del sistema en términos de eficiencia, tiempo de respuesta, y capacidad de adaptación a diversos escenarios. Este análisis es crucial para reflejar el cumplimiento de los objetivos del proyecto y para identificar áreas en las que se pueden aplicar mejoras.

Asimismo, se confirma la capacidad del sistema para cumplir con su funcionalidad principal de generación y seguimiento de rutas a partir de puntos de inicio y fin definidos. Este resultado ha demostrado la eficacia del sistema en la simulación de trayectorias y en la adaptabilidad de su algoritmo de ruteo, consolidando su capacidad operativa en el entorno para el que fue diseñado.

Los escenarios evaluados en diferentes condiciones y configuraciones permitieron identificar los factores que influyen en el rendimiento del sistema. Estas observaciones han revelado ventanas de oportunidad para optimizar los algoritmos y mejorar la precisión del sistema en la generación de rutas, lo cual será de gran utilidad para el desarrollo de versiones futuras.

Finalmente, este informe recopila y documenta los resultados generales del proyecto, destacando el cumplimiento de los objetivos establecidos y los logros alcanzados. Asimismo, ofrece un análisis de las áreas de mejora identificadas, estableciendo una base para futuras investigaciones y desarrollos que puedan optimizar el sistema y expandir su aplicabilidad en nuevos contextos.

Referencias

- [1] P. G. Tzionas, A. Thanailakis, and P. G. Tsalides, “Collision-free path planning for a diamond-shaped robot using two-dimensional cellular automata,” *IEEE Xplore*, vol. 13, no. 2, pp. 237–250, 1997.
- [2] H. J. M. Lopes and D. A. Lima, “Patrolling simulation model for swarm robotics using ant memory cellular automata maps with genetic algorithms optimization,” *SSRN*, 2023.
- [3] J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 1979.
- [4] J. Aranda, N. Duro, J. L. Fernández, J. Jiménez, and F. Morilla, *Fundamentos de Lógica Matemática y Computación*. Sanz y Torres, 2006.
- [5] S. Wolfram, *A New Kind of Science*. Wolfram Media, 1959.
- [6] R. Sharma, “Torus.” <https://alchetron.com/Torus>, 2022. Accessed: 2022-02-01.
- [7] E. F. Codd, *Cellular Automata*. ACM Monograph Series, Academic Press, 1968.
- [8] M. Gardner, “The game of life,” *Scientific American*, vol. 223, no. 4, pp. 4–25, 1970.
- [9] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling*. Cambridge, MA: MIT Press, 1987.
- [10] C. E. Shannon, “A mathematical theory of communication,” *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948.
- [11] E. Ott, *Chaos in Dynamical Systems*. Cambridge, UK: Cambridge University Press, 1993.
- [12] D. G. Luenberger, *Introduction to Dynamic Systems: Theory, Models, and Applications*. New York, NY: Stanford University Press, 1979.
- [13] C. Rojas and S. L. Stephenson, *Myxomycetes: Biology, Systematics, Biogeography, and Ecology*. San Diego, CA: Academic Press, 2017. An imprint of Elsevier.
- [14] H. Stempfen and S. L. Stephenson, *Myxomycetes: A handbook of slime molds*. Portland, OR: Timber Press, 1994.
- [15] J. Dee, “A mating-type system in an acellular slime-mould,” *Nature*, vol. 184, pp. 780–781, 1960.
- [16] A. Adamatzky, *Atlas of Physarum Computing*. USA: World Scientific Publishing Co., Inc., 2015.
- [17] Y. Sun, P. N. Hameed, K. Verspoor, and S. Halgamuge, “A physarum-inspired prize-collecting steiner tree approach to identify subnetworks for drug repositioning,” *BMC Systems Biology*, vol. 10, no. 5, p. 128, 2016.
- [18] Y. Lu, Y. Liu, C. Gao, L. Tao, and Z. Zhang, “A novel physarum-based ant colony system for solving the real-world traveling salesman problem,” in *Advances in Swarm Intelligence* (Y. Tan, Y. Shi, and C. A. C. Coello, eds.), (Cham), pp. 173–180, Springer International Publishing, 2014.

- [19] S. Venkatesh, E. Braund, and E. Miranda, “Composing popular music with physarum polycephalum-based memristors,” in *Proceedings of the International Conference on New Interfaces for Musical Expression* (R. Michon and F. Schroeder, eds.), (Birmingham, UK), pp. 514–519, Birmingham City University, July 2020.
- [20] O. Elek, J. N. Burchett, J. X. Prochaska, and A. G. Forbes, “Monte Carlo Physarum Machine: Characteristics of Pattern Formation in Continuous Stochastic Transport Networks,” *Artificial Life*, vol. 28, pp. 22–57, 06 2022.
- [21] Z. Cai, G. Li, J. Zhang, and S. Xiong, “Using an artificial physarum polycephalum colony for threshold image segmentation,” *Applied Sciences*, vol. 13, no. 21, 2023.
- [22] N. Heer, “Speed comparison of programming languages.” <https://github.com/niklas-heer/speed-comparison>, 2023. Accessed: 2023-02-01.
- [23] P. Santamaría, “Raspberry pi: la historia del minipc más famoso del mundo.” <https://eloutput.com/productos/gadgets/raspberry-pi/>, 2023. Accessed: 2023-02-01.
- [24] E. Kitamura and M. Ubl, “Presentación de websockets: el ingreso de los sockets a la web.” web.dev, 2012. Accedido el 2 de octubre de 2024.
- [25] I. Fette and A. Melnikov, “The websocket protocol.” RFC 6455, Internet Engineering Task Force, December 2011.
- [26] I. Fette and A. Melnikov, “The websocket protocol.” RFC 6455, Internet Engineering Task Force, December 2011.
- [27] M. Kravkovsky, “Websockets and their impact on reducing bandwidth and latency,” *Journal of Internet Technology*, vol. 22, pp. 221–233, August 2019.
- [28] J. Smith and L. Garcia, “Security in websockets: Challenges and solutions,” *Security Protocols*, vol. 30, no. 2, pp. 155–169, 2020.
- [29] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, “Hypertext transfer protocol http/1.1.” RFC 2616, Internet Engineering Task Force, June 1999.
- [30] Google, “Webrtc - comunicación en tiempo real en la web.” <https://webrtc.org/?hl=es-419>, 2024. Accedido el 2 de octubre de 2024.
- [31] A. Adamatzky, *Physarum machines: computers from slime mould*, vol. 74. World Scientific, 2010.
- [32] A. Adamatzky, “Slime mold solves maze in one pass, assisted by gradient of chemo-attractants,” *IEEE Transactions on NanoBioscience*, vol. 11, pp. 131–134, June 2012.
- [33] J. Jones and A. Adamatzky, “Emergence of self-organized amoeboid movement in a multi-agent approximation of physarum polycephalum,” 2012.
- [34] G. R. Olvera, E. J. S. Méndez, G. J. Martínez, and L. N. O. Moreno, “Modelado del physarum polycephalum con autómatas celulares para el enrutado de robots mensajeros,” Trabajo Terminal 2021 - B013, Instituto Politécnico Nacional, Escuela Superior de Cómputo, México CDMX, enero 2023.
- [35] E. Y. Marín Alavez, “Modelado del physarum polycephalum implementado en robot basado en autómatas celulares,” Mayo 2018. Trabajo Terminal 2017 - A056.

- [36] J. Jones, *From pattern formation to material computation: multi-agent modelling of Phy-sarum Polycephalum*, vol. 15. Springer, 2015.
- [37] Y.-P. Gunji, T. Shirakawa, T. Niizato, and T. Haruna, “Minimal model of a cell connecting amoebic motion and adaptive transport networks,” *Journal of theoretical biology*, vol. 253, pp. 659–67, 05 2008.
- [38] M. Guevara-Bonilla, A. S. Meza-Leandro, E. A. Esquivel-Segura, D. Arias-Aguilar, A. Tapia-Arenas, and F. Masís-Meléndez, “Uso de vehículos aéreos no tripulados (vant’s) para el monitoreo y manejo de los recursos naturales: una síntesis,” *Tecnología en Marcha*, vol. 33, no. 4, pp. 77–88, 2020.
- [39] A. Yehoshua and Y. Edan, “Mobile robots sampling algorithms for monitoring of insects populations in agricultural fields,” 2023.
- [40] R. Bogue, “The role of robots in environmental monitoring,” *Industrial Robot: the international journal of robotics research and application*, vol. 50, no. 3, pp. 369–375, 2023.