



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

Junio 2024

Índice

1. Introducción	3
1.1. Objetivo	4
1.1.1. Objetivo General	4
1.1.2. Objetivos Específicos	4
1.2. Justificación	4
2. Marco Teórico	5
2.1. Introducción a los autómatas celulares	5
2.1.1. Teoría de Autómatas	5
2.1.2. Autómatas celulares de una dimensión	6
2.1.3. Condiciones frontera	7
2.1.4. Vecindario	8
2.1.5. Definición formal	9
2.1.6. Autómatas celulares de 2 Dimensiones	10
2.1.7. Ejemplo	11
2.1.8. Entropía de Shannon	12
2.1.9. Sistemas Dinámicos	12
2.2. Physarum Polycephalum	14
2.2.1. Mixomiceto	14
2.2.2. Ciclo de vida	14
2.2.3. Physarum Polycephalum	17
2.2.4. El Physarum Polycephalum visto desde la perspectiva computacional . .	18
2.3. Modo Gráfico	19
2.3.1. SFML	20
2.4. RasberyPi	21
2.4.1. Historia y Evolución	21
2.4.2. Comparativa	23
2.4.3. RasberyPi 4 Model B	24
2.5. Protocolos de comunicación	26
2.5.1. WebSocket	26
2.5.2. HTTP	26
2.5.3. WebRTC	27
3. Estado del Arte	28
3.1. Physarum Polycephalum	28
3.1.1. Modelado de Adamatzky	28
3.1.2. Modelado Guillermo Olvera	29
3.1.3. Modelado de Yair Marin	31
3.1.4. Modelado de Jeff Jones	31
3.1.5. Modelado de Gunji	32
3.2. Robots para Monitoreo Poblacional	34
3.2.1. Uso de vehículos aéreos no tripulados VANT's para el monitoreo y manejo de los recursos naturales: una síntesis	34
3.2.2. Mobile robot's sampling algorithms for monitoring of insects' populations in agricultural fields	34
3.2.3. The Role of Robots in Environmental Monitoring	35
4. Propuesta a desarrollar	36
4.1. Diagramas	36

5. Implementación	46
5.1. Simulador del Physarum Polycephalum	46
5.2. Robot Propuesto	51
5.2.1. Desarrollo Inicial del Sistema del control del Robot	52
6. Pruebas del sistema	62
6.1. Diseño de módulos de hardware del robot para pruebas de integración	62
6.1.1. Módulo de actuadores (Motores)	62
6.1.2. Módulo de sensores (LiDAR)	63
6.1.3. Unidad de control (Raspberry Pi 4 B)	64
6.1.4. Módulo de visualización	64
6.1.5. Pruebas de integración	65
6.2. Desarrollo de módulos de hardware del robot para pruebas de integración	66
6.2.1. Módulo de actuadores	67
6.2.2. Módulo de sensores	67
6.2.3. Unidad de control	68
6.2.4. Sistema de visualización	77
6.2.5. Integración de módulos	77

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 *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^[4], que es una jerarquía de lenguajes formales. La cual sería la siguiente:

- **Tipo 3 - Gramáticas regulares:** Estos generan los lenguajes regulares. Estas se restrinjen 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 restrinjen 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 restrinjen 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 restrinjen 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 *Elementary 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.

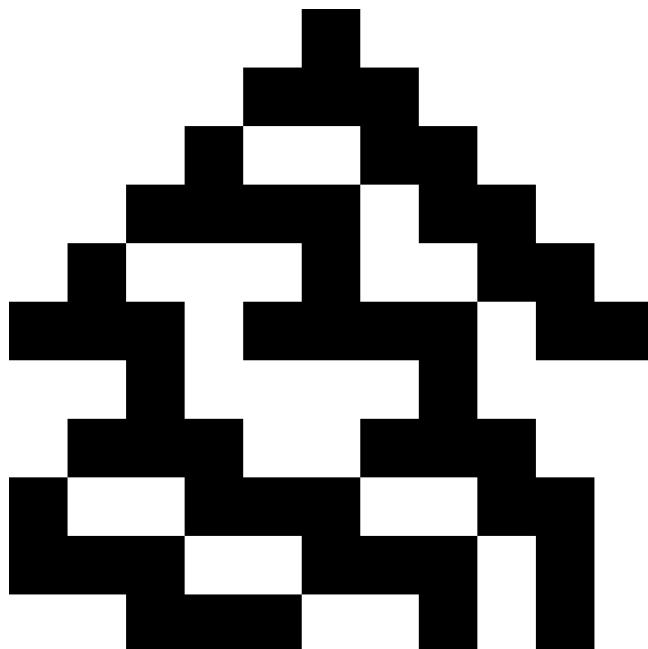


Figura 1: Regla de evolución 30 de las *Reglas de Wolfram* 10 generaciones o (t)

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 la tabla 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 linea 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 2.1.4 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.

A su vez los vecindarios más usados son los siguientes:

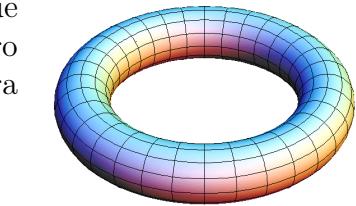
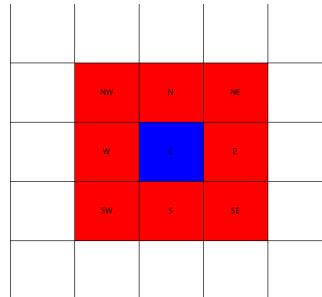
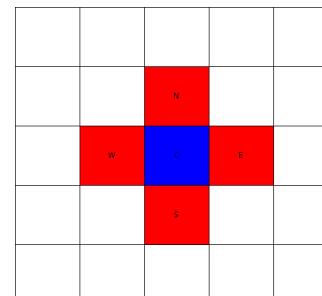


Figura 2: Representación de un toroide [6]

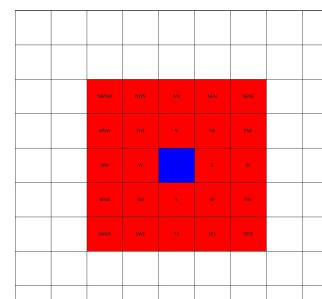
- **Vecindario de Moore** En este caso el vecindario de una celda es la celda y sus ocho vecinos.



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

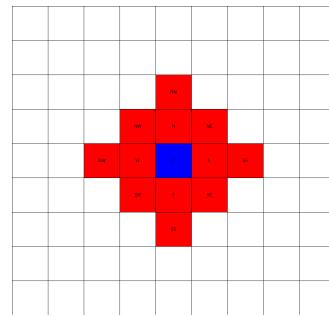


- **Vecindario de Moore extendido** En este caso el vecindario de una celda es la celda y sus veinticuatro vecinos.



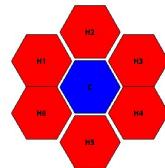
Vecindario de Von Neumann extendido

- En este caso el vecindario de una celda es la celda y sus doce vecinos.



Vecindario de hexagonal

- En este caso el vecindario de una celda es la celda y sus seis vecinos.



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 mas detalle los autómatas celulares de 2 dimensiones, los cuales son los que se usan en este trabajo terminal.

2.1.6. Autómatas celulares de 2 Dimensiones

Los autómatas celulares de 2 dimensiones son los que se usan en este trabajo terminal, por ello es necesario explicarlos con más detalle. Primero recordando lo ya explicado en anteriores subsecciones, un autómata celular de 2 dimensiones es una tupla (\mathbb{Z}^2, S, N, f) 2.1.5 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 eucliano, en suelen ser dispuestas 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 $N = 8$
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 aplinan repetidamente a lo largo del tiempo, generalmente de manera sincróna, 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:

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

Dandonos como resultado la siguiente evolución en 250 generaciones:

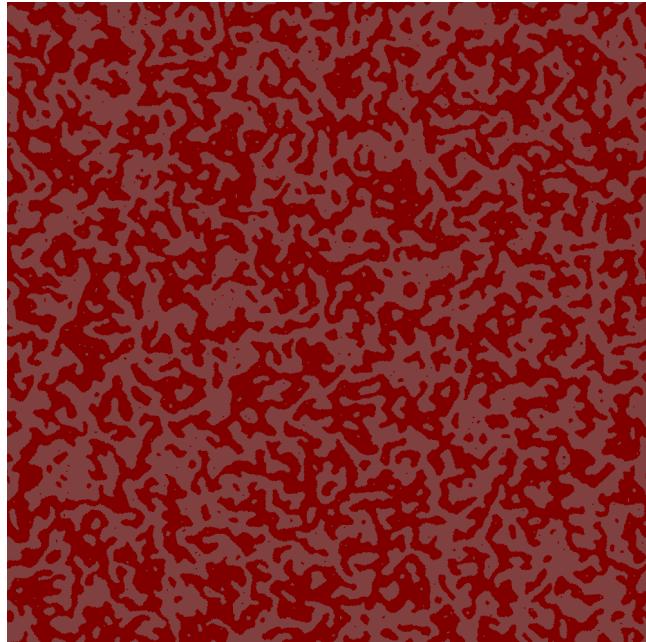


Figura 3: Evolución del autómata celular *Anneal*

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 como 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 4:

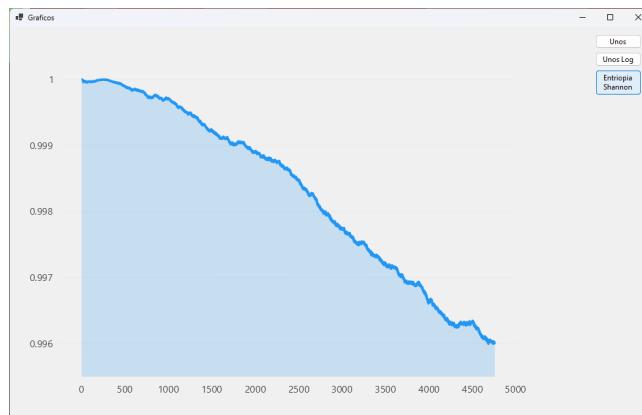


Figura 4: Entropía del autómata celular *Anneal* en 5000 generaciones

2.1.9. Sistemas Dinámicos

La teoría de Sistemas Dinámicos puede considerarse una forma de describir cómo un determinado estado se transforma o evoluciona en otro a lo largo del tiempo, es decir, cómo evoluciona un sistema en el tiempo. 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 5 podemos ver un ejemplo de la evolución de un autómata celular en el tiempo:

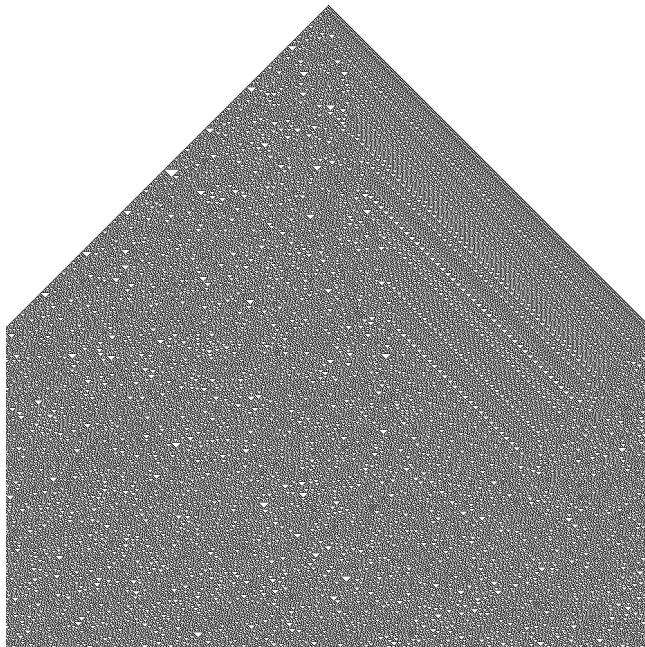


Figura 5: Evolución de la regla 30 en 1000 generaciones o pasos de tiempo

No daremos una explicación mas 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 si 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, quisieramos destacar que para hacer nuestra clasificación o decir que 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 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 y luego fue liberada de uno de los cuerpos fructíferos característicamente producidos por los mixomicetos. 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 6, 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].

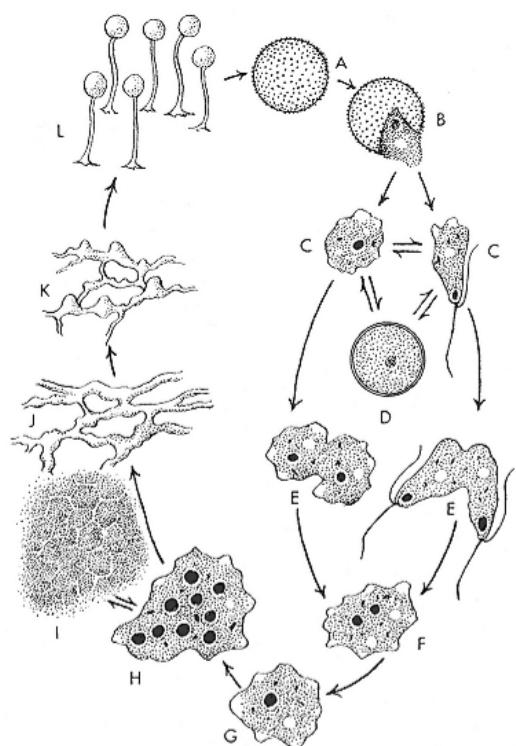


Figura 6: Ciclo de vida de los mixomicetos, extraido de "Myxomycetes: A Handbook of Slime Molds" de Stephenson y Stempel [14].

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 7.

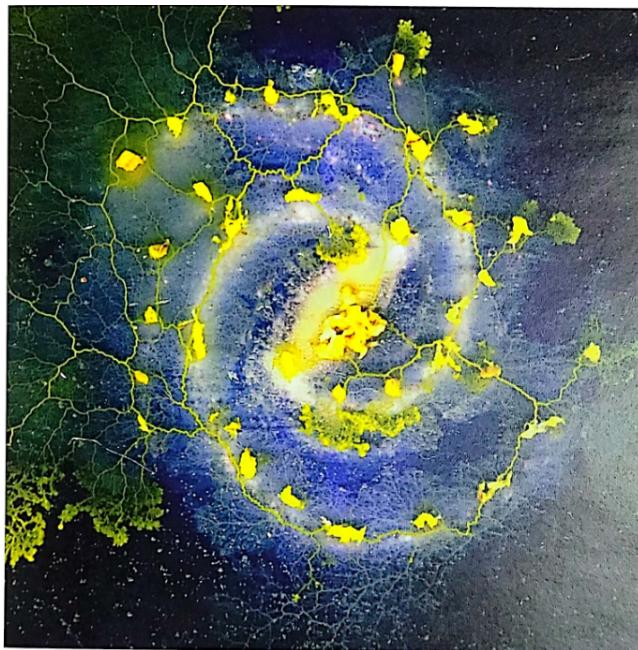


Figura 7: *Physarum* propagandose en una impresion artistica de una galaxia. Imagen extraida de 'Atlas of *Physarum Computing*' de A. Adamatzky [16].

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 8, encontrar la ruta más corta entre dos puntos, y tomar decisiones complejas basadas en la informa-



Figura 8:
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, 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 (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 viajante (Traveling Salesman Problem). 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 (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 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. SFML

Para el desarrollo de nuestro Trabajo Terminal, se utilizó la biblioteca gráfica SFML (Simple and Fast Multimedia Library), la cual es una biblioteca gráfica multiplataforma de código abierto, que proporciona una 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 9. 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.

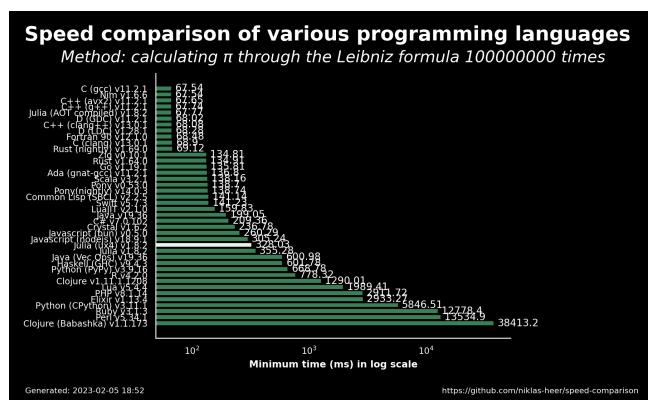


Figura 9: Comparación de tiempo de ejecución entre diferentes lenguajes de Programación. Esta gráfica fue generada Heer [22]

2.4. RasberryPi

En nuestro Trabajo Terminal, 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. A continuación se muestra una tabla comparativa entre todos 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 anterior subsubsección, la Raspberry Pi 4 Model B es una computadora de placa única (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.

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 videos 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, 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.

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 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 tener desafíos de seguridad, como la exposición a ataques de "Cross-Site WebSocket Hijacking" (CSWSH) o vulnerabilidades de inyección. Por esta razón, es importante integrar medidas de seguridad, como el uso de WebSockets sobre TLS (WSS) para cifrar las comunicaciones, y políticas adecuadas de validación del origen de las conexiones. [28]

2.5.2. HTTP

El protocolo HTTP (Hypertext Transfer Protocol) 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 TCP (Transmission Control Protocol) 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 MQTT, que están diseñados específicamente para este propósito. [29]

2.5.3. WebRTC

WebRTC (Web Real-Time Communication) 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 de datos en tiempo real (RTP), el protocolo de control de transmisión en tiempo real (RTCP) y el protocolo de control de sesión (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 IoT (Internet of Things), 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 10 se muestra un ejemplo de puertas lógicas 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 quimioatraventante y generando patrones complejos. Además, se considera la resistencia al movimiento y la influencia de estímulos externos como quimioatraventantes y luz, controlando el comportamiento del colectivo.

Una característica destacada es la capacidad del colectivo para cambiar y recuperar su forma,

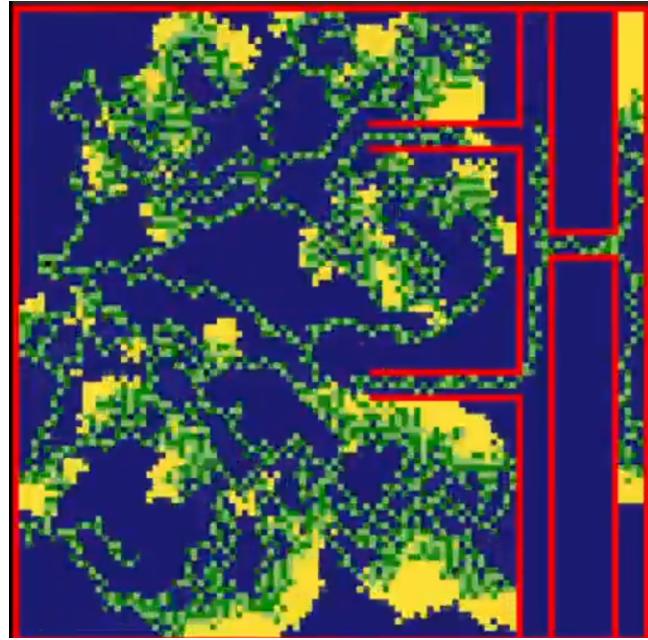


Figura 10: Ejemplo de puertas lógicas (OR) creadas por *Physarum*.

navegar obstáculos y dividirse en fragmentos independientes que pueden fusionarse nuevamente, lo cual es deseable en aplicaciones robóticas [31]. La Figura 11 ilustra cómo el colectivo de *Physarum* se adapta a obstáculos.



Figura 11: Colectivo de *Physarum* adaptándose a obstáculos. [32]

Para más detalles, ver [33].

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 y 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.

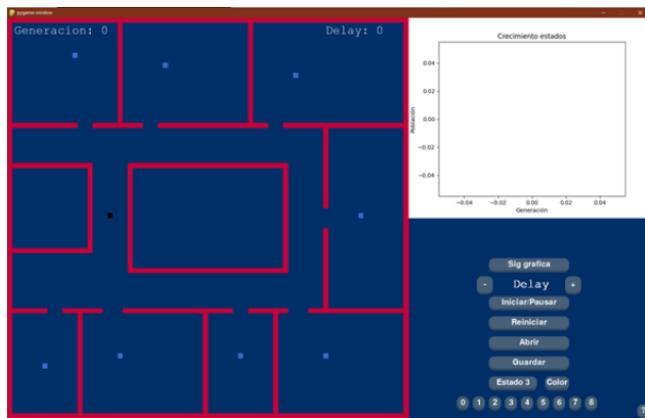


Figura 12: Configuración inicial del *Physarum* en la cuadrícula.

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.

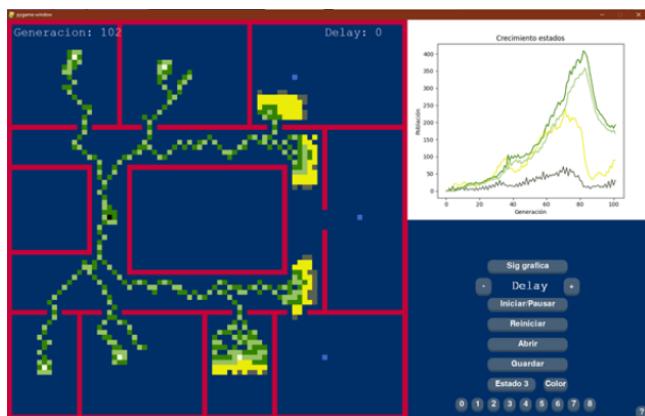


Figura 13: Expansión del *Physarum* y formación de rutas. [34]

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

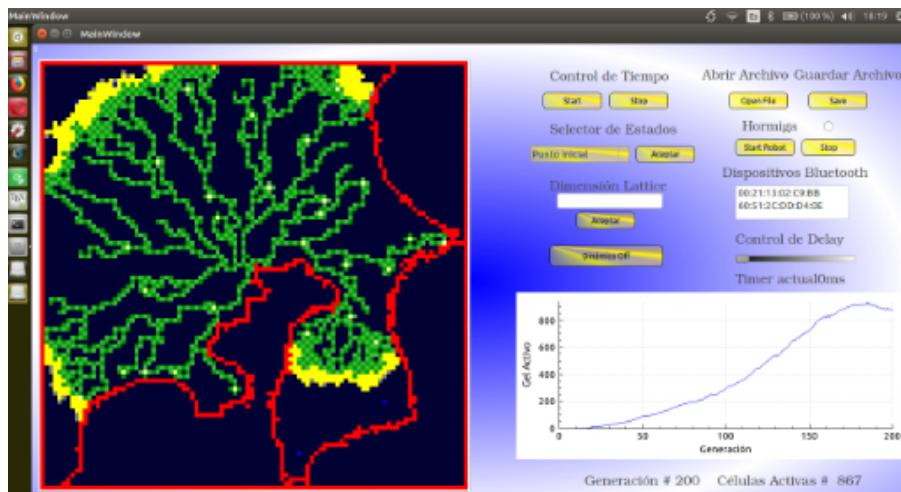


Figura 14: Ejecución del algoritmo de *Physarum Polycephalum* encontrando nutrientes. [35]

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*.

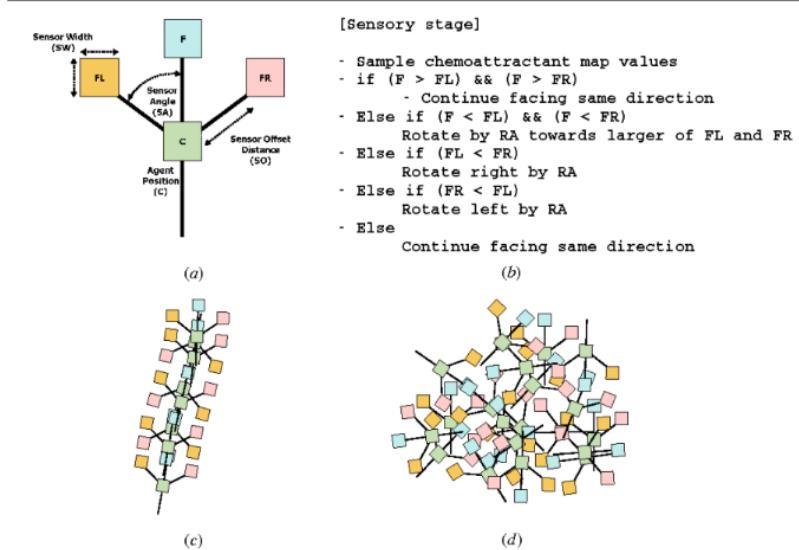


Figura 15: Representación del agente de acuerdo con el modelo de Jeff Jones. [36]

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 16, 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.

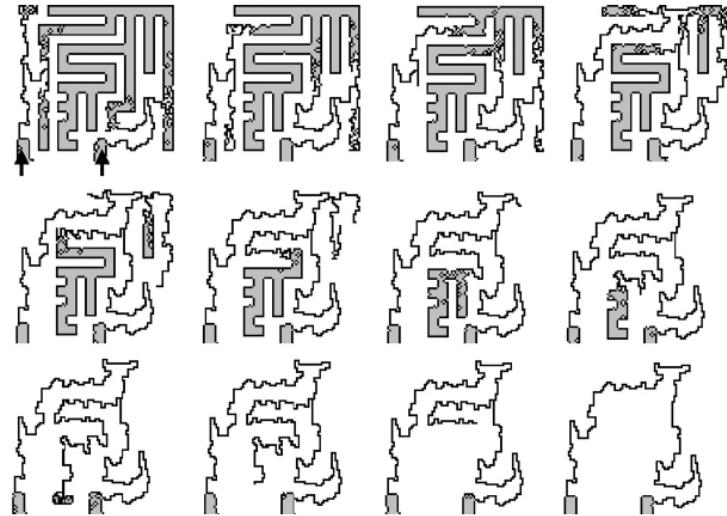


Figura 16: Aplicación del algoritmo para resolver un laberinto utilizando el modelo de *Physarum polycephalum*. [37]

Las reglas del modelo se describen en la Figura 17, mostrando los diferentes estados de los sitios y cómo interactúan durante las fases de desarrollo y búsqueda de alimento.

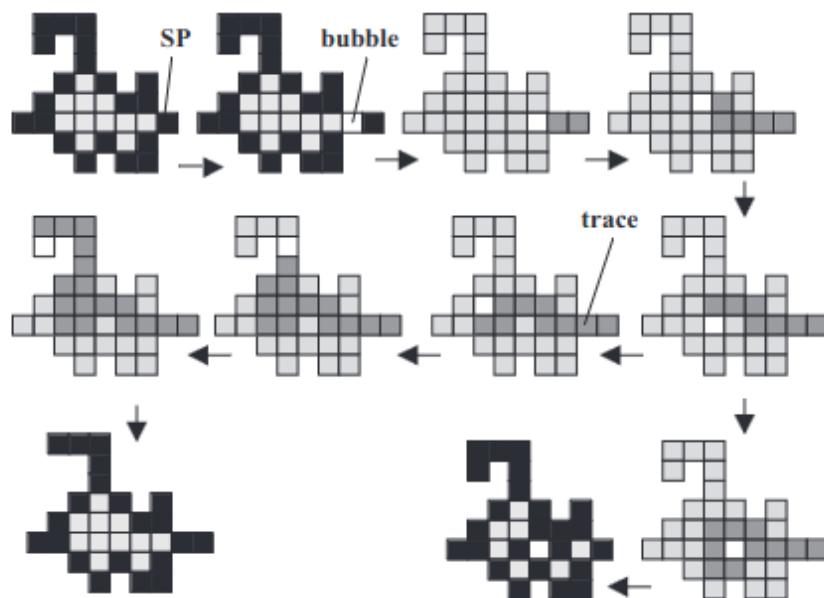


Figura 17: Reglas del modelo de célula mínima, mostrando los estados de los sitios y las interacciones. [37]

3.2. Robots para Monitoreo Poblacional

En esta sección, presentamos un resumen de los trabajos previos relacionados con robots o dispositivos aereos 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 funcionara de la siguiente manera: los robots autónomos recibirán la ruta a seguir por parte de nuestro simulador del *Physarum Polycephalum*, el cual se encargara de determinar la ruta optima 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 permitira a los investigadores y a las autoridades locales monitorear poblaciones de manera eficiente y en tiempo real. Además, el sistema permitira 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 sera el simulador del *Physarum Polycephalum*, el cual sera un sistema que permitira determinar rutas optimas para recolectar información de la población. El segundo producto sera el sistema de monitoreo poblacional, el cual sera un sistema que permitira a los robots autónomos recolectar información de la población y de los entornos en los que se encuentran.

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

4.1. 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.

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 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 desde el robot, y el núcleo del sistema, el algoritmo de **Physarum**, encargado

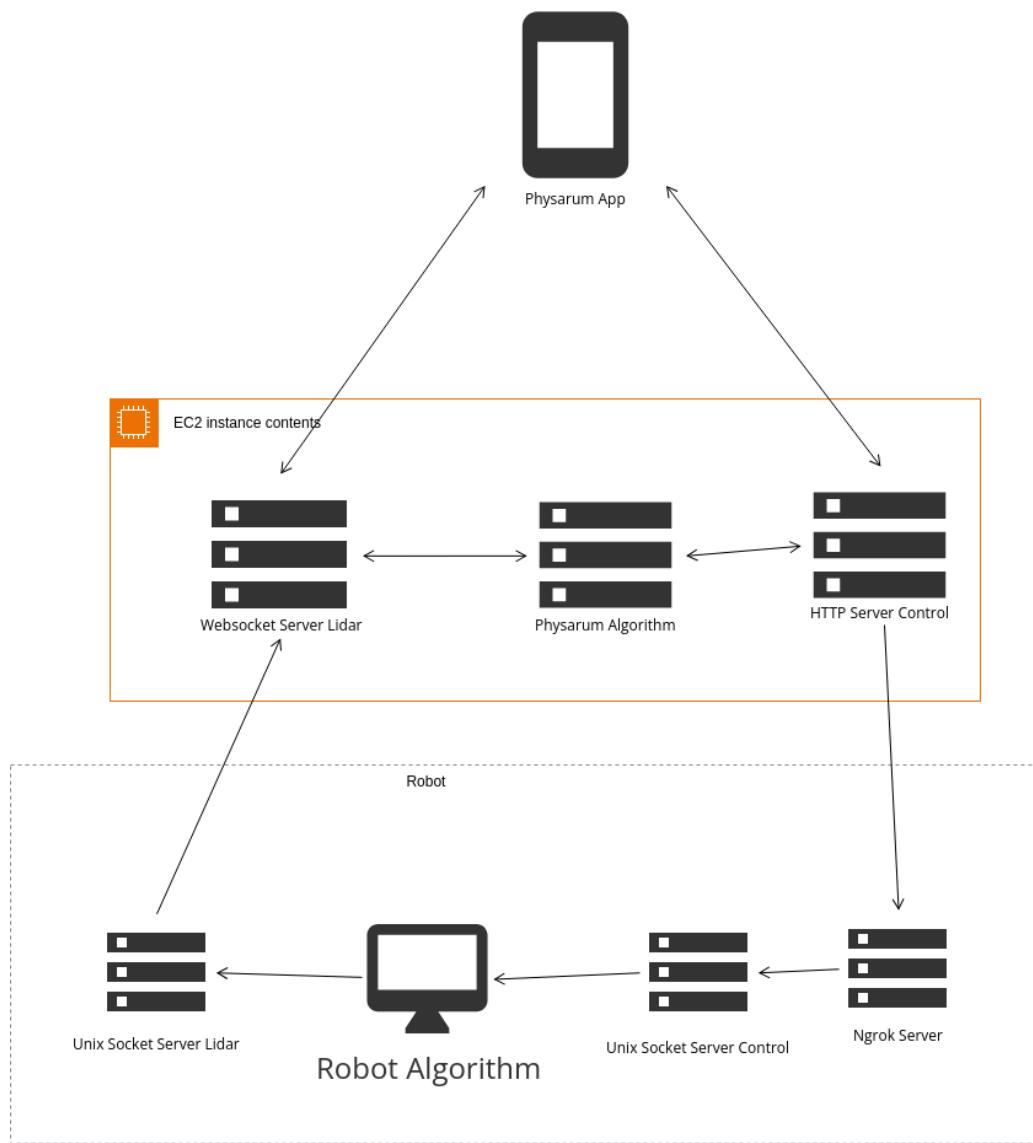


Figura 18: Diagrama de arquitectura del sistema propuesto.

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 los dos productos a desarrollar: el simulador de **Physarum polycephalum**, el sistema del robot autónomo y el diagrama de flujo de la aplicación.

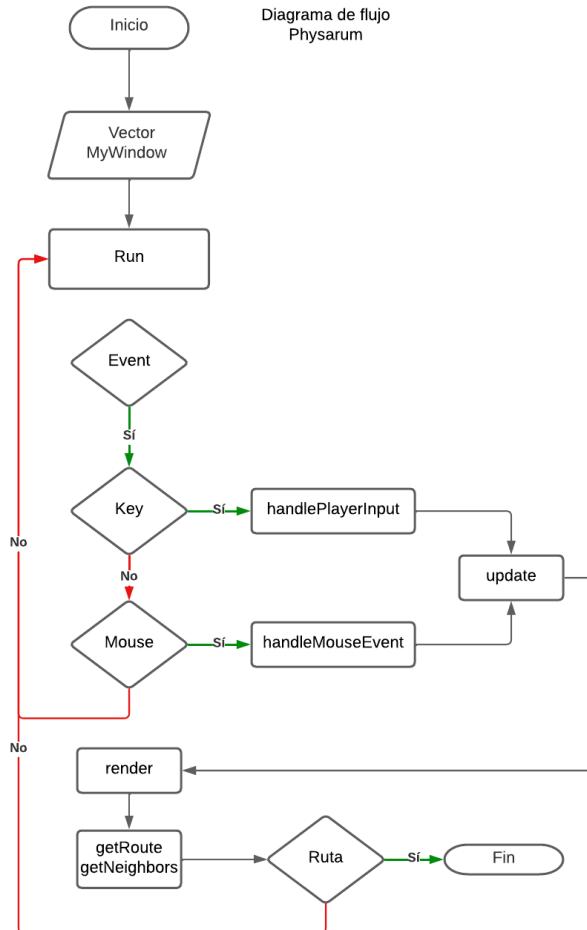


Figura 19: Diagrama de flujo del simulador de **Physarum polycephalum**.

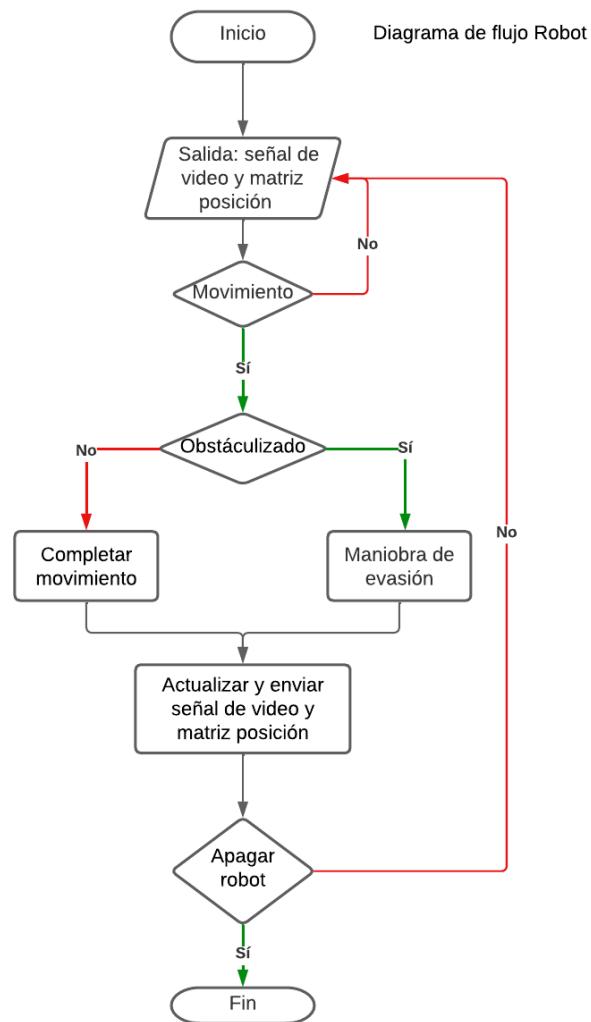


Figura 20: Diagrama de flujo del sistema del robot autónomo.

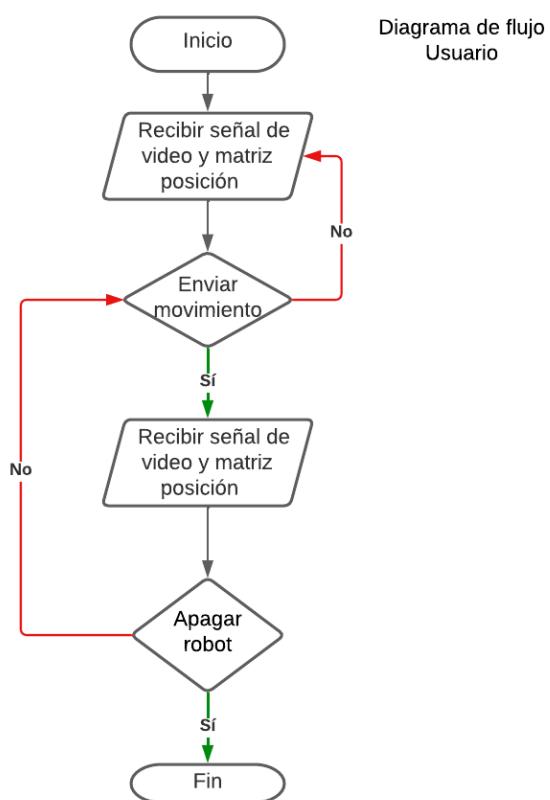


Figura 21: Diagrama de flujo de la aplicación móvil.

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

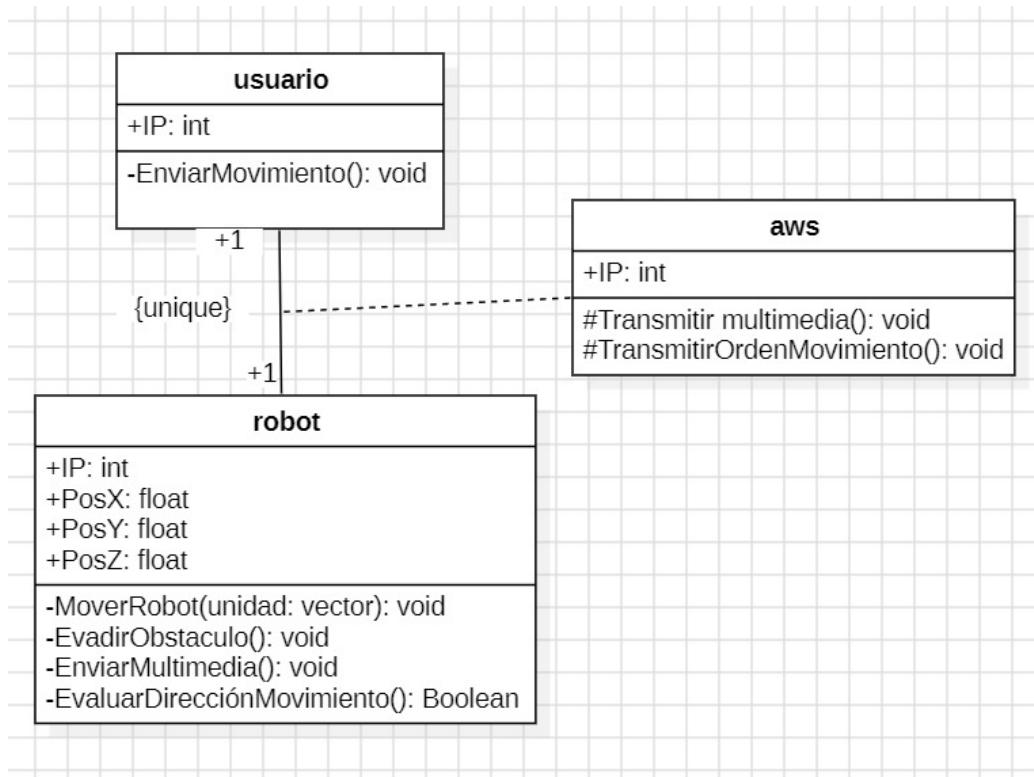


Figura 22: Diagrama de clases del sistema propuesto.

El diagrama de clases muestra la interacción entre el usuario, el robot y un servidor en 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()*), evitar 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.

Se presenta el diagrama de casos de usos del sistema propuesto.

Cuadro 3: 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	<p>A) El conductor selecciona el modo de operación del robot.</p> <p>B) El conductor envía una señal de movimiento.</p> <p>C) El sistema AWS recibe la señal y la transmite al robot.</p> <p>D) El robot recibe la señal y ejecuta el movimiento según las instrucciones.</p> <p>E) El robot evita colisiones mientras se mueve.</p> <p>F) El robot transmite la señal multimedia en tiempo real.</p> <p>G) El conductor observa la multimedia transmitida en tiempo real.</p>
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.

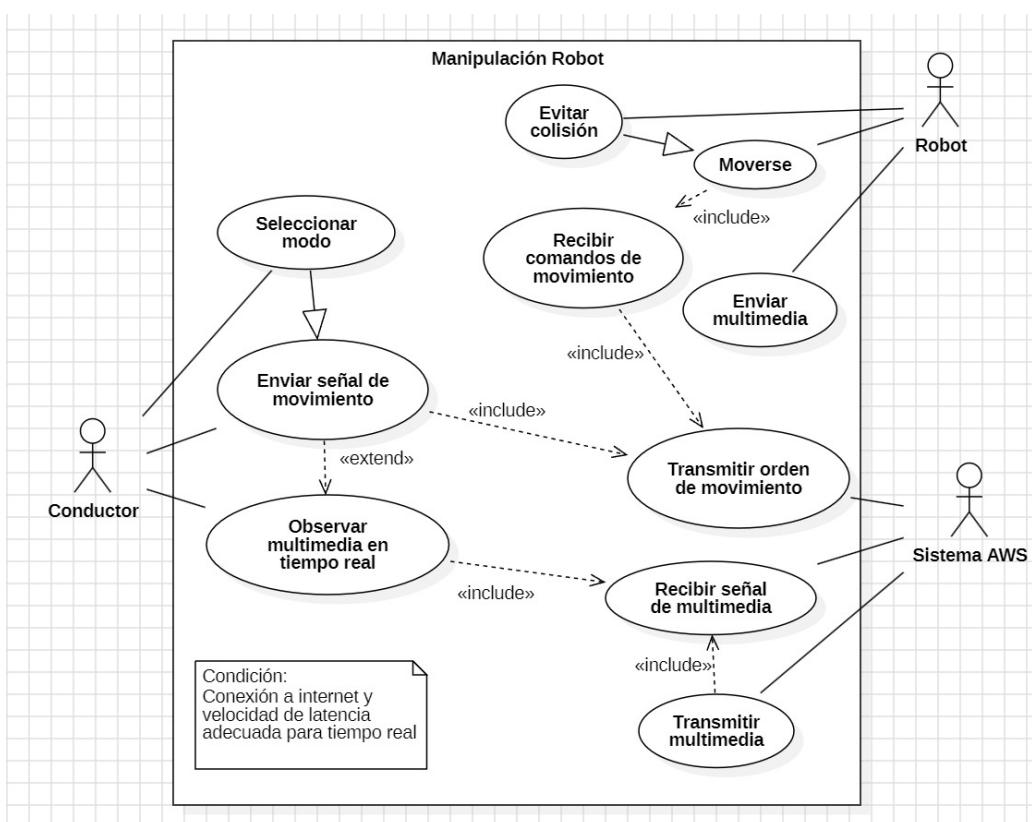


Figura 23: Diagrama de casos de uso del sistema propuesto.

Finalmente, se presenta el diagrama de secuencia del sistema propuesto.

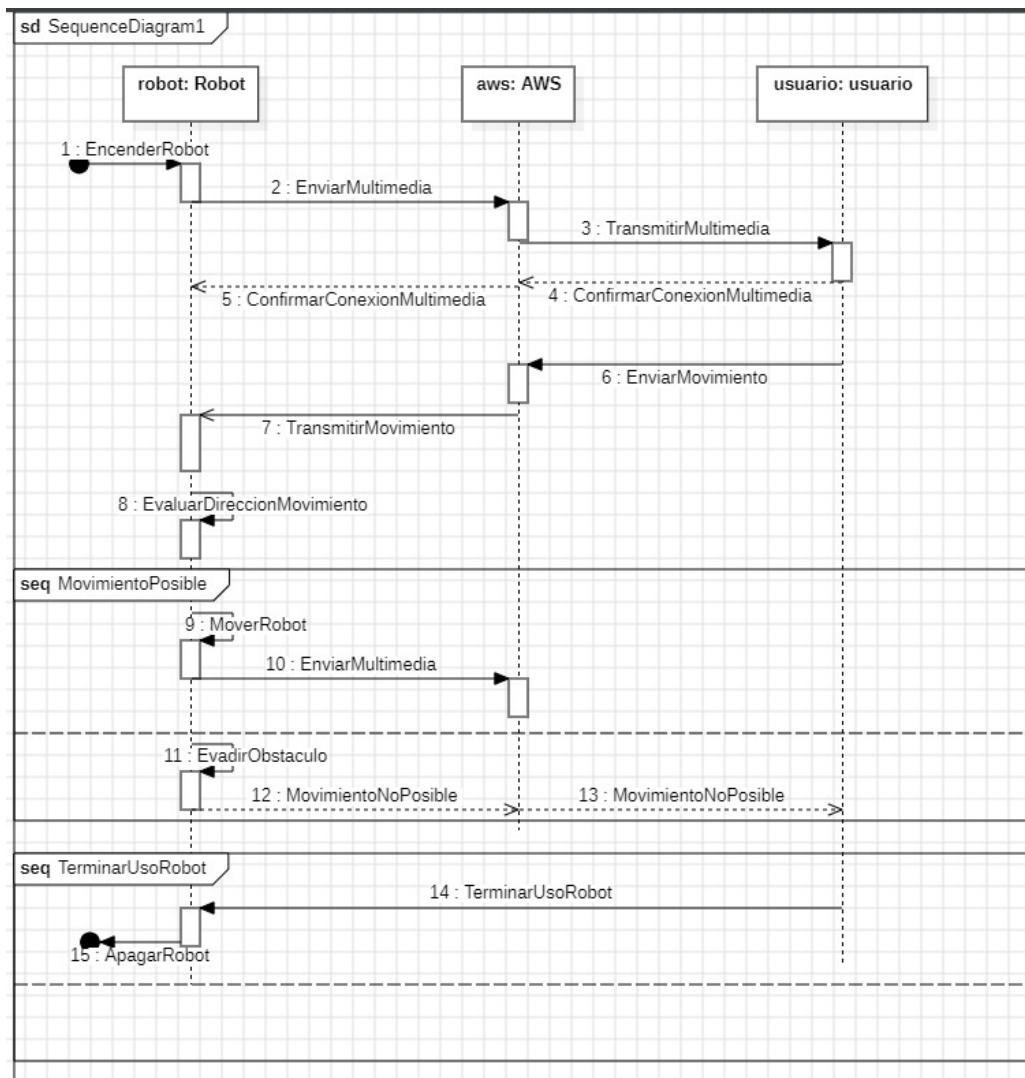


Figura 24: Diagrama de secuencia del sistema propuesto.

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 encargara de determinar la ruta optima para recolectar información de la población. Por otro lado, el robot con una Raspberry Pi se encargara de recolectar información de la población y de los entornos en los que se encuentran.

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

5.1. Simulador del Physarum Polycephalum

El algoritmo propuesto se basa en el descrito por Jeff Jones en su obra 'From Pattern Formation to Material Computation: Multi-agent Modelling of Physarum Polycephalum' [36].

El algoritmo es un autómata celular que simula el comportamiento de Physarum Polycephalum en un laberinto, por lo que necesitamos definir algunos conceptos. Primero, denotemos \mathbb{Z} como el conjunto de enteros, es decir, $\mathbb{Z} = (-\infty, -1, 0, 1, \infty)$. y la longitud de cualquier tupla x como $|x|$. Para todas las tuplas x e y de la misma longitud, denotemos $x \oplus y$ como la tupla que resulta de la suma componente a componente de x e 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 dimensión n es al menos 1 donde $n \in \mathbb{Z}^+$, S es un conjunto finito y no vacío de estados, N es un conjunto finito y no vacío 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 de vecindarios posibles 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 $f : \{0, 1, 2, 3, 4, 5, 6, 7, 8\}^9 \rightarrow \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$, $(C(x, y : t), N(x, y : t), M(x, y : t))$ representa el estado combinado, vecindario y memoria de la célula en la posición (x, y) en el tiempo t . La función de transición f

se define como sigue:

$$f(C(x, y : t), N(x, y : t), M(x, y : t)) = \begin{cases} 7 & \text{si } C(x, y : t) = 0 \text{ y } \exists N \in \{3, 4, 6\} \text{ y } M(x, y : t) = 0 \\ 6 & \text{si } C(x, y : t) = 1 \text{ y } \exists N \in \{5, 6\} \\ 2 & \text{si } C(x, y : t) = 2 \\ 3 & \text{si } C(x, y : t) = 3 \\ 5 & \text{si } C(x, y : t) = 4 \text{ y } \exists N \in \{3, 5, 6\} \\ & \text{y } M(x, y : t) = 0 \text{ y } N \notin \{0, 7\} \\ 0 & \text{si } C(x, y : t) = 5 \text{ y } M(x, y : t) \notin \{5, 8\} \\ & \text{y } N \notin \{1, 3, 4, 6\} \\ 8 & \text{si } C(x, y : t) = 5 \\ & \text{y la condición anterior no se cumple} \\ 4 & \text{si } C(x, y : t) = 7 \text{ y } \exists N \in \{3, 4, 6\} \\ 5 & \text{si } C(x, y : t) = 8 \\ 1 & \text{si } C(x, y : t) = 1 \end{cases}$$

Donde los estados del autómata celular se definen en la **Tabla 6**.

Color	Estado	Descripción
Dark Blue	0	Campo libre
Light Blue	1	Nutriente no encontrado
Red	2	Repelente
Black	3	Punto inicial
Yellow	4	Gel contrayéndose
Green	5	Gel compuesto
Light Yellow	6	Nutriente encontrado
Grey	7	Expansión de Physarum
Green	8	Gel no compuesto

Cuadro 4: Estados del autómata celular

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 5: Estados del autómata celular

En la fuente mencionada, se detalla el algoritmo básico de *Physarum Polycephalum*, diseñado originalmente para operar con un vecindario de Von Neumann. Sin embargo, en la versión

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 Green	6	0
Grey	7	0
Green	8	0

Cuadro 6: Estados del autómata celular

propuesta aquí, optamos por modificar el vecindario a uno 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 Figs 25, 26 y 27. En estas imágenes, el número total de células es de 50 x 50.

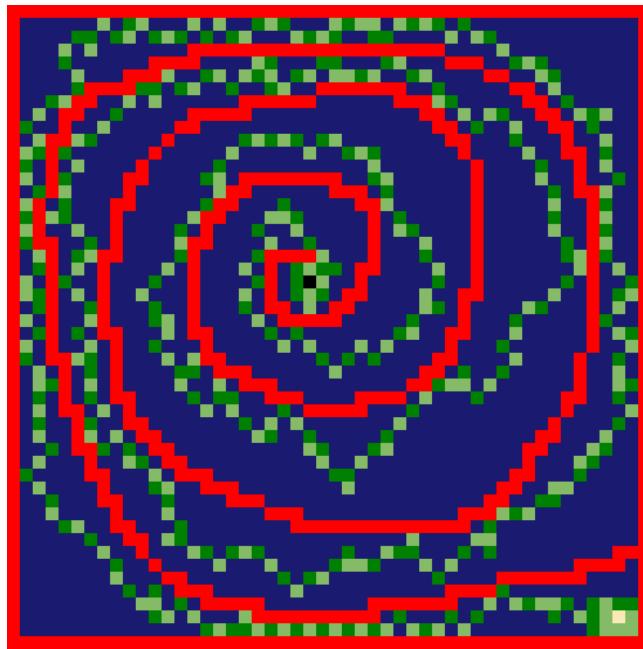


Figura 25: Physarum Polycephalum resolviendo un laberinto en espiral.

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

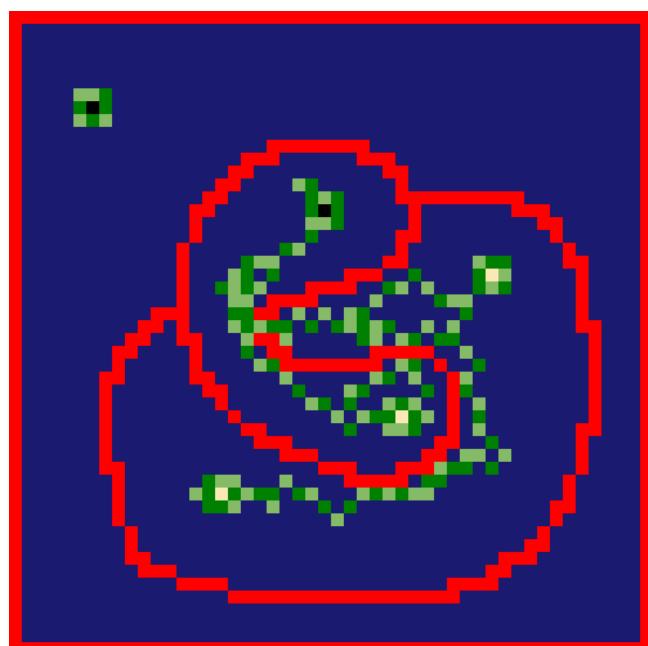


Figura 26: *Physarum Polycephalum* resolviendo un laberinto de tipo circular.

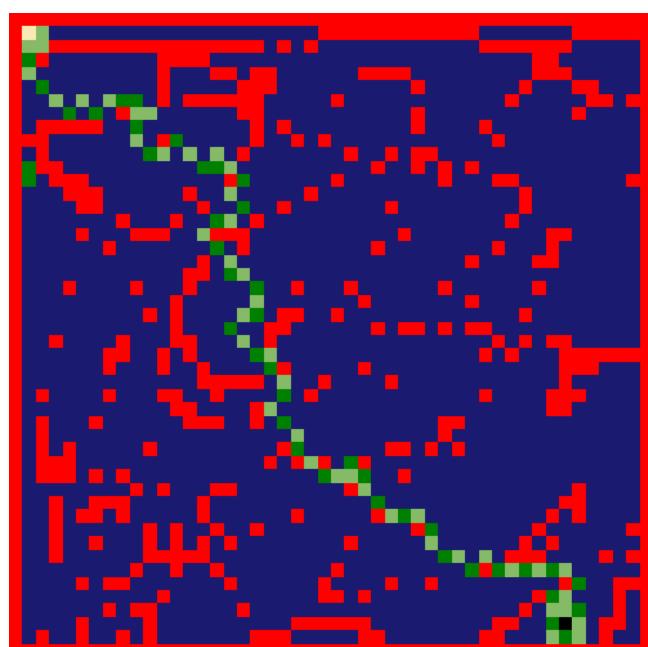


Figura 27: *Physarum Polycephalum* resolviendo un laberinto con obstáculos.

mediante un algoritmo de mapeo de imágenes que ayuda en la representación gráfica de dicha topografía, como se muestra en la Fig 28.

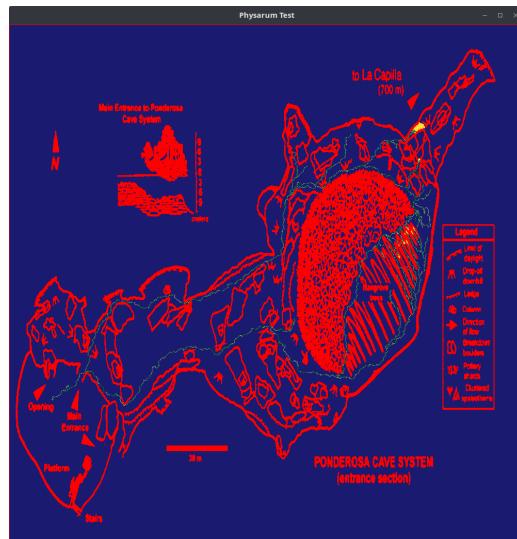


Figura 28: Mapeo del sistema de cuevas usando Physarum Polycephalum.

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 Fig 29. El espacio explorado por el algoritmo es de 1000 x 1000 células.

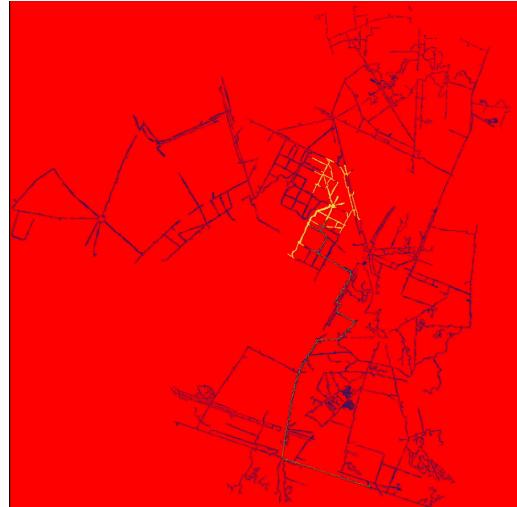


Figura 29: Mapeo de la catacumba usando Physarum Polycephalum, utiliza 5264 pasos para obtener la ruta.

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 de la siguiente manera:

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 código 7, 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.

5.2. 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 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 es el siguiente código:

Listing 2: 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     #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 int frequency = 400; // Frecuencia inicial
30
31 std::atomic<bool> is_running(true);
32 std::atomic<bool> is_manual_mode(true);
33
34 void setMotorSpeed(int motor, int frequency) {
35     if (motor >= 0 && motor < 4) {
36         gpioSetPWMfrequency(PWM_PINS[motor], frequency);
37     }
38 }
39
40 void setMotorDirection(int motor, int direction) {
41     if (motor >= 0 && motor < 4) {
42         gpioWrite(DIR_PINS[motor], direction);
43     }
44 }
45
46 void stopMotors() {
47     for (int i = 0; i < 4; ++i) {
48         gpioPWM(PWM_PINS[i], 0);
49     }
50 }
51
52 void moveForward() {
53     for (int i = 0; i < 4; ++i) {
54         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
             al 50%
55     }
56     setMotorDirection(0, 0); // Motor 1
57     setMotorDirection(2, 0); // Motor 3
58     setMotorDirection(1, 1); // Motor 2
59     setMotorDirection(3, 1); // Motor 4
60 }
61
62 void moveBackward() {
63     for (int i = 0; i < 4; ++i) {
64         gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
             al 50%
65     }
66     setMotorDirection(0, 1); // Motor 1
67     setMotorDirection(2, 1); // Motor 3
68     setMotorDirection(1, 0); // Motor 2
69     setMotorDirection(3, 0); // Motor 4
70 }
71

```

```

72     void turnLeft() {
73         for (int i = 0; i < 4; ++i) {
74             gpioPWM(PWM_PINS[i], 128); // Establecer ciclo de trabajo
75                         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
86                         al 50%
87         }
88         setMotorDirection(0, 0); // Motor 1
89         setMotorDirection(2, 0); // Motor 3
90         setMotorDirection(1, 0); // Motor 2
91         setMotorDirection(3, 0); // Motor 4
92     }
93
94     void increaseSpeed() {
95         if (frequency < 2000) {
96             frequency += 100;
97             if (frequency > 2000) frequency = 2000;
98             for (int i = 0; i < 4; ++i) {
99                 setMotorSpeed(i, frequency);
100            }
101        }
102    }
103
104    void decreaseSpeed() {
105        if (frequency > 400) {
106            frequency -= 100;
107            if (frequency < 400) frequency = 400;
108            for (int i = 0; i < 4; ++i) {
109                setMotorSpeed(i, frequency);
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
117                         de rojo a verde
118    }
119
120    int contadorObstaculoTotal = 0;
121    int obstaculoHola = 0;
122    int obsRelativo = 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
127                         la probabilidad de movimiento
128
129        const float FRONT_MIN_ANGLE = -10.0f * (M_PI / 180.0f); // -15
130                         grados en radaianes

```

```

127     const float FRONT_MAX_ANGLE = 10.0f * (M_PI / 180.0f); // 15
128         grados en radianes
129
130     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
143                     // relativo al "sur" del robot
144                     float adjusted_angle = point.angle + PI;
145                     //std::cout << "Er " << point.range << std::endl;
146
147                     // Verificar si el punto est dentro del rango
148                     // frontal de 30
149                     //std::cout << point.angle << std::endl;
150                     if (point.range > 0 && point.range < 0.30 &&
151                         point.angle <= -0.5235f && point.angle
152                         >= -2.617f) { // Norte
153                         obstacle_detected = true;
154                         currentScanPoints.push_back(point.range);
155                         std::cout << "Obstacle distance N: " <<
156                             (float)point.range << " Y en el angulo
157                             " << point.angle << std::endl;
158                         break;
159                     } else if (point.range > 0 && point.range <
160                         0.25 && (point.angle <= 0.872f && point.
161                         angle >= -0.5235f)){ // Este
162                         obstacle_detected = true;
163                         currentScanPoints.push_back(point.range);
164                         std::cout << "Obstacle distance E: " <<
165                             (float)point.range << " Y en el angulo
166                             " << point.angle << std::endl;
167                         break;
168                     } else if (point.range > 0 && point.range <
169                         0.45 && (point.angle <= 2.26f && point.angle
170                         >= 0.872f)){ // Sur
171                         obstacle_detected = true;
172                         currentScanPoints.push_back(point.range);
173                         std::cout << "Obstacle distance S:" <<
174                             (float)point.range << " Y en el angulo
175                             " << point.angle << std::endl;
176                         break;
177                     } else if (point.range > 0 && point.range <
178                         0.25 && (point.angle <= -2.61f || point.
179                         angle >= 2.27f)){ // Oeste
180                         obstacle_detected = true;
181                         currentScanPoints.push_back(point.range);
182                         std::cout << "Obstacle distance O:" <<
183                             (float)point.range << " Y en el angulo
184                             " << point.angle << std::endl;
185                         break;
186                     }
187                 }

```

```

167
168     }
169
170     if (obstacle_detected) {
171         // Si detecta un obstaculo, retrocede por 4
172         // segundos
173         contadorObstaculoTotal++;
174         std::cout << "Obs: " << contadorObstaculoTotal
175             << std::endl;
176         //obsRelativo++;
177         stopMotors();
178         //Luego gira aleatoriamente a la izquierda o
179         //derecha
180
181         turnRight();
182         std::this_thread::sleep_for(std::chrono::
183             seconds(5));
184
185         std::this_thread::sleep_for(std::chrono::
186             seconds(2));
187         stopMotors();
188
189         if(!previousScanPoints.empty() &&
190             previousScanPoints.size() ==
191             currentScanPoints.size()){
192             bool mismoObs = true;
193             for(size_t i = 0; i <
194                 currentScanPoints.size(); ++i){
195                 if(fabs(currentScanPoints[i] -
196                     previousScanPoints[i]) > 0.05)
197                 {
198                     mismoObs = false;
199                     break;
200                 }
201             }
202             if(mismoObs){
203                 obsRelativo++;
204             }else{
205                 obsRelativo = 0;
206             }
207         }
208
209         previousScanPoints = currentScanPoints;
210
211         if((obsRelativo > 3)){
212             //obsRelativo = 0;
213             moveBackward();
214             std::this_thread::sleep_for(std::chrono::
215                 seconds(3));
216             stopMotors();
217             int turn = dist(gen) % 2;
218             if (turn == 0) {
219                 //turnLeft();
220                 turnRight();
221                 std::this_thread::sleep_for(std::
222                     chrono::seconds(5));
223             } else {
224                 turnRight();
225                 std::this_thread::sleep_for(std::
226                     chrono::seconds(5));

```

```

214         }
215         std::this_thread::sleep_for(std::chrono::seconds(2));
216         stopMotors();
217     }
218 } else {
219     moveForward();
220     // Si no hay obstculo, elige un movimiento aleatorio
221     //
222     //int move = dist(gen);
223     //switch (move) {
224     //    case 0: moveBackward(); break;
225     //    case 1: turnRight(); break;
226     //    case 2: turnLeft(); break;
227     //    case 3: moveForward(); break;
228     //}
229 }
230 }
231
232     std::this_thread::sleep_for(std::chrono::milliseconds
233                             (500));
234     //stopMotors();
235 }
236     std::this_thread::sleep_for(std::chrono::milliseconds(100)
237                             );
238 }
239
240 int main() {
241     // Inicializar pigpio
242     if (gpioInitialise() < 0) {
243         std::cerr << "Error: No se pudo inicializar pigpio." <<
244             std::endl;
245         return -1;
246     }
247
248     // Configurar pines de direccin como salida
249     for (int i = 0; i < 4; ++i) {
250         gpioSetMode(DIR_PINS[i], PI_OUTPUT);
251         gpioSetMode(PWM_PINS[i], PI_OUTPUT);
252         setMotorSpeed(i, frequency); // Inicializar PWM con
253                                     frecuencia inicial
254     }
255
256     // Asegrate de establecer XDG_RUNTIME_DIR
257     if (getenv("XDG_RUNTIME_DIR") == nullptr) {
258         setenv("XDG_RUNTIME_DIR", "/tmp/runtime-$(id -u)", 1);
259     }
260
261     // Ejecuta libcamera-vid en un proceso separado y captura la
262     // salida en YUV, sin previsualizacin
263     FILE* pipe = popen("libcamera-vid -t 0 --codec yuv420 --
264                         nopreview -o -", "r");
265     if (!pipe) {
266         std::cerr << "Error: No se pudo ejecutar libcamera-vid."
267             << std::endl;
268         return -1;
269     }

```

```

265 // Configura la ventana SFML
266 sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera
267     Visualization with LiDAR");
268 sf::Texture cameraTexture;
269 sf::Sprite cameraSprite;
270
271 // Buffer para leer los datos de video
272 const int width = 640;
273 const int height = 480;
274 std::vector<uint8_t> buffer(width * height * 3 / 2); // Ajusta
275             el tamao del buffer para YUV420
276
277 cv::Mat yuvImage(height + height / 2, width, CV_8UC1, buffer.
278             data());
279 cv::Mat rgbImage(height, width, CV_8UC3);
280
281 std::string port;
282 ydlidar::os_init();
283
284 // Obtener los puertos disponibles de LiDAR
285 std::map<std::string, std::string> ports = ydlidar::
286             lidarPortList();
287 if (ports.size() > 1) {
288     auto it = ports.begin();
289     std::advance(it, 1); // Selecciona el segundo puerto
290             disponible
291     port = it->second;
292 } else if (ports.size() == 1) {
293     port = ports.begin()->second;
294 } else {
295     std::cerr << "No se detect ningn LiDAR. Verifica la
296             conexin." << std::endl;
297     return -1;
298 }
299
300 // Configuracin del LiDAR
301 int baudrate = 115200;
302 std::cout << "Baudrate: " << baudrate << std::endl;
303
304 CYdLidar laser;
305 laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
306             ());
307 laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
308             int));
309
310 bool isSingleChannel = true;
311 laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
312             sizeof(bool));
313
314 float max_range = 8.0f;
315 float min_range = 0.1f;
316 float max_angle = 180.0f;
317 float min_angle = -180.0f;
318 float frequency = 8.0f;
319
320 laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
321             );
322 laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)
323             );

```

```

313     laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float))
314         );
314     laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float))
315         );
315     laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
316         float));
316
317     // Inicializar LiDAR
318     if (!laser.initialize()) {
319         std::cerr << "Error al inicializar el LiDAR." << std::endl
320             ;
320         return -1;
321     }
322
323     // Iniciar el escaneo
324     if (!laser.turnOn()) {
325         std::cerr << "Error al encender el LiDAR." << std::endl;
326         return -1;
327     }
328
329     // Thread para movimiento aleatorio
330     std::thread randomMoveThread(randomMovement, std::ref(laser));
331
332     while (window.isOpen()) {
333         sf::Event event;
334         while (window.pollEvent(event)) {
335             if (event.type == sf::Event::Closed)
336                 window.close();
337
338             if (event.type == sf::Event::KeyPressed) {
339                 switch (event.key.code) {
340                     case sf::Keyboard::W:
341                         is_manual_mode = true;
342                         moveForward();
343                         break;
344                     case sf::Keyboard::S:
345                         is_manual_mode = true;
346                         moveBackward();
347                         break;
348                     case sf::Keyboard::A:
349                         is_manual_mode = true;
350                         turnLeft();
351                         break;
352                     case sf::Keyboard::D:
353                         is_manual_mode = true;
354                         turnRight();
355                         break;
356                     case sf::Keyboard::V:
357                         is_manual_mode = true;
358                         increaseSpeed();
359                         break;
360                     case sf::Keyboard::B:
361                         is_manual_mode = true;
362                         decreaseSpeed();
363                         break;
364                     case sf::Keyboard::Space:
365                         is_manual_mode = true;
366                         stopMotors();
367                         break;
368                     case sf::Keyboard::K:

```

```

369                     is_manual_mode = false;
370                     break;
371                 case sf::Keyboard::M:
372                     is_manual_mode = true;
373                     break;
374                 default:
375                     break;
376             }
377         }
378     }
379
380     // Leer los datos del video desde la tubera
381     size_t bytesRead = fread(buffer.data(), 1, buffer.size(),
382                               pipe);
383     if (bytesRead != buffer.size()) {
384         std::cerr << "Error: No se pudo leer suficientes datos
385         de video." << std::endl;
386         continue;
387     }
388
389     // Convertir YUV420 a RGB
390     cv::cvtColor(yuvImage, rgbImage, cv::COLOR_YUV2RGB_I420);
391
392     // Convertir a RGBA añadiendo un canal alfa
393     cv::Mat frame_rgba;
394     cv::cvtColor(rgbImage, frame_rgba, cv::COLOR_RGB2RGBA);
395
396     // Actualizar la textura de la cámara con los datos del
397     // frame
398     if (!cameraTexture.create(frame_rgba.cols, frame_rgba.rows))
399     {
400         std::cerr << "Error: No se pudo crear la textura." <<
401         std::endl;
402         continue;
403     }
404     cameraTexture.update(frame_rgba.ptr());
405
406     cameraSprite.setTexture(cameraTexture);
407     cameraSprite.setScale(
408         window.getSize().x / static_cast<float>(cameraTexture.
409             getSize().x),
410         window.getSize().y / static_cast<float>(cameraTexture.
411             getSize().y)
412     );
413
414     window.clear();
415     window.draw(cameraSprite);
416
417     // Crear un minimapa para el LiDAR
418     sf::RectangleShape minimap(sf::Vector2f(200, 200));
419     minimap.setFillColor(sf::Color(200, 200, 200, 150)); // Fondo semitransparente
420     minimap.setPosition(10, 10); // Esquina superior izquierda
421
422     window.draw(minimap);
423
424     // Dibujar el centro del LiDAR (color azul)
425     sf::CircleShape lidarCenter(5); // Radio del círculo del
426     // LiDAR
427     lidarCenter.setFillColor(sf::Color::Blue);

```

```

420         lidarCenter.setPosition(105, 105); // Posicion del centro
421         en el minimapa
422
423
424         // Dibujar la linea hacia el norte
425         sf::Vertex line[] =
426         {
427             sf::Vertex(sf::Vector2f(110, 110), sf::Color::Black),
428             sf::Vertex(sf::Vector2f(110, 160), sf::Color::Black)
429             // Lnea hacia arriba (norte)
430         };
431
432
433         window.draw(line, 2, sf::Lines);
434
435         LaserScan scan;
436         if (laser.doProcessSimple(scan)) {
437             for (const auto& point : scan.points) {
438                 // Convertir coordenadas polares a cartesianas
439                 float x = point.range * cos(point.angle);
440                 float y = point.range * sin(point.angle);
441
442                 // Ajustar los puntos al minimapa
443                 float scale = 25.0f;
444                 float adjustedX = 110 + x * scale;
445                 float adjustedY = 110 - y * scale; // Invertir Y
446                 para coordinar con la pantalla
447
448                 // Dibujar los puntos en el minimapa, excluyendo
449                 el centro (0,0)
450                 if (point.range > 0.05) {
451                     sf::CircleShape lidarPoint(2);
452                     lidarPoint.setPosition(adjustedX, adjustedY);
453                     lidarPoint.setFillColor(getPointColor(point.
454                     range, max_range));
455
456                     window.draw(lidarPoint);
457                     //std::cout << "An " << point.range << std :: endl;
458                 }
459             } else {
460                 std::cerr << "No se pudieron obtener los datos del
461                 LiDAR." << std::endl;
462             }
463
464             window.display();
465         }
466
467         // Detener el escaneo del LiDAR
468         laser.turnOff();
469         laser.disconnecting();
470
471         // Cierra la tubera, detiene los motores y apaga el robot
472         pclose(pipe);
473         stopMotors();
474         gpioTerminate();
475
476         is_running = false;
477         randomMoveThread.join();
478

```

```
474     return 0;  
475 }
```

En este código 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.

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 sistema de monitoreo poblacional, 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. 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. 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.

6.1.1. 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.

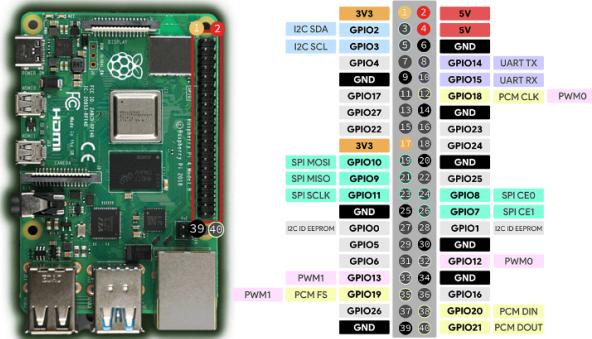


Figura 30: Diseño de los pinos de configuración de los módulos de hardware del robot.

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 7: 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.

6.1.2. 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.

6.1.3. 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.

6.1.4. 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.



Figura 31: Diseño de los pines de configuracion de los módulos de hardware 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.

6.1.5. 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.

6.2. 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.

6.2.1. 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.

Listing 3: 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     }
```

6.2.2. 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.

Listing 4: 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:

Listing 5: 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
8     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(int))
9             ;
10    int optval = TYPE_TRIANGLE;
11    laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
12    optval = YDLIDAR_TYPE_SERIAL;
13    laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
14    optval = isSingleChannel ? 3 : 4;
15    laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
16    optval = 4;
17    laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof(int)
18                      );
19
20    bool b_optvalue = false;
21    laser.setlidaropt(LidarPropFixedResolution, &b_optvalue, sizeof(
22                      bool));
23    laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool));
24    laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool));
25    b_optvalue = true;
26    laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(bool
27                      ));
28    laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel, sizeof(
29                      bool));
30    b_optvalue = false;
31    laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool));
32    b_optvalue = true;
33    laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
34                      sizeof(bool));
35    b_optvalue = false;
36    laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue, sizeof(
37                      bool));
38
39    float f_optvalue = 180.0f;
40    laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float));
41    f_optvalue = -180.0f;
42    laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float));
43    f_optvalue = 64.0f;
44    laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float));
45    f_optvalue = 0.05f;
46    laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float));
47    laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(float
48                      ));

```

6.2.3. 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.

Listing 6: 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 timestamp) {
28         Mat depth_tmp(Size(640, 480), CV_16UC1, depth);
29         depth_tmp.copyTo(depthMat);
30         cout << "Depth frame received at timestamp: " << timestamp << endl;
31     }
32
33     void rgb_cb(freenect_device *dev, void *rgb, uint32_t timestamp) {
34         Mat rgb_tmp(Size(640, 480), CV_8UC3, rgb);
35         cvtColor(rgb_tmp, rgbMat, COLOR_RGB2RGBA);
36         cout << "RGB frame received at timestamp: " << timestamp << endl;
37     }
38
39     void drawGrid(sf::RenderTarget &target, float gridSpacing, float offsetX, float offsetY) {
40         sf::Color gridColor(200, 200, 200); // Light gray color
41         for (float x = offsetX; x < target.getSize().x; x += gridSpacing) {
42             sf::Vertex line[] = {
43                 sf::Vertex(sf::Vector2f(x, 0), gridColor),
44                 sf::Vertex(sf::Vector2f(x, target.getSize().y), gridColor)
45             };
46             target.draw(line, 2, sf::Lines);
47         }
48         for (float x = offsetX; x > 0; x -= gridSpacing) {
49             sf::Vertex line[] = {
```

```

50     sf::Vertex(sf::Vector2f(x, 0), gridColor),
51     sf::Vertex(sf::Vector2f(x, target.getSize().y),
52                 gridColor)
53   };
54   target.draw(line, 2, sf::Lines);
55 }
56 for (float y = offsetY; y < target.getSize().y; y += gridSpacing) {
57   sf::Vertex line[] = {
58     sf::Vertex(sf::Vector2f(0, y), gridColor),
59     sf::Vertex(sf::Vector2f(target.getSize().x, y),
60               gridColor)
61   };
62   target.draw(line, 2, sf::Lines);
63 }
64 for (float y = offsetY; y > 0; y -= gridSpacing) {
65   sf::Vertex line[] = {
66     sf::Vertex(sf::Vector2f(0, y), gridColor),
67     sf::Vertex(sf::Vector2f(target.getSize().x, y),
68               gridColor)
69   };
70   target.draw(line, 2, sf::Lines);
71 }
72 sf::Color getPointColor(float distance, float maxRange) {
73   float ratio = distance / maxRange;
74   return sf::Color(255 * (1 - ratio), 255 * ratio, 0); // Color
75   from red to green
76 }
77 void drawNorthIndicator(sf::RenderTarget &lidarTexture, float
78 offsetX, float offsetY) {
79   sf::Color northColor(255, 0, 0); // Red color for the north
80   indicator
81   float length = 50.0f; // Length of the north indicator line
82
83   sf::Vertex line[] = {
84     sf::Vertex(sf::Vector2f(offsetX, offsetY), northColor),
85     sf::Vertex(sf::Vector2f(offsetX, offsetY - length),
86               northColor) // Line pointing upwards
87   };
88   lidarTexture.draw(line, 2, sf::Lines);
89 }
90 void detectAndDrawDepthObstacles(Mat &depthMat, sf::RenderTarget
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     }
106   }
107 }
```

```

97         sf::CircleShape circle(3); // Small circle to
98             // represent the obstacle
99             circle.setPosition(adjustedX, adjustedY);
100            circle.setFillColor(sf::Color::Red);

101           lidarTexture.draw(circle);

102           // Add the obstacle to the vector
103           depthObstacles.push_back(Point2f(adjustedX,
104               adjustedY));
105       }
106   }
107 }
108 }

109 void compareObstacles(const vector<Point2f> &depthObstacles, const
110 vector<Point2f> &lidarObstacles) {
111 cout << "Comparing Obstacles:" << endl;
112 for (const auto &d : depthObstacles) {
113     bool matchFound = false;
114     for (const auto &l : lidarObstacles) {
115         float distance = sqrt(pow(d.x - l.x, 2) + pow(d.y - l.
116             y, 2));
117         if (distance < 10.0f) { // If the distance is less
118             than a threshold, consider it a match
119             matchFound = true;
120             break;
121         }
122     }
123     if (matchFound) {
124         cout << "Match found for depth obstacle at (" << d.x
125             << ", " << d.y << ")" << endl;
126     } else {
127         cout << "No match for depth obstacle at (" << d.x << "
128             , " << d.y << ")" << endl;
129     }
130 }
131 }

132 void freenect_thread_func() {
133     while (is_running) {
134         freenect_process_events(f_ctx);
135     }
136 }

137 void print_frame_modes() {
138     cout << "Available video modes:" << endl;
139     for (int res = FREENECK_RESOLUTION_LOW; res <=
140         FREENECK_RESOLUTION_HIGH; ++res) {
141         for (int fmt = FREENECK_VIDEO_RGB; fmt <=
142             FREENECK_VIDEO_IR_8BIT; ++fmt) {
143             freenect_frame_mode mode = freenect_find_video_mode((
144                 freenect_resolution)res, (freenect_video_format)
145                 fmt);
146             if (mode.is_valid) {
147                 cout << "Resolution: " << res << ", Format: " <<
148                     fmt << ", Width: " << mode.width << ", Height:
149                         " << mode.height << ", Bytes per pixel: " <<
150                             mode.data_bits_per_pixel << endl;
151             }
152         }
153     }
154 }
```

```

143     }
144 }
145
146 cout << "Available depth modes:" << endl;
147 for (int res = FREENECK_RESOLUTION_LOW; res <=
148     FREENECK_RESOLUTION_HIGH; ++res) {
149     for (int fmt = FREENECK_DEPTH_11BIT; fmt <=
150         FREENECK_DEPTH_REGISTERED; ++fmt) {
151         freenect_frame_mode mode = freenect_find_depth_mode(
152             freenect_resolution)res, (freenect_depth_format)
153             fmt);
154         if (mode.is_valid) {
155             cout << "Resolution: " << res << ", Format: " <<
156                 fmt << ", Width: " << mode.width << ", Height:
157                     " << mode.height << ", Bytes per pixel: " <<
158                         mode.data_bits_per_pixel << endl;
159         }
160     }
161 }
162
163 int main(int argc, char *argv[]) {
164     std::string port;
165     ydlidar::os_init();
166
167     // Initialize Freenect
168     if (freenect_init(&f_ctx, NULL) < 0) {
169         cerr << "Freenect init failed" << endl;
170         return -1;
171     }
172     freenect_set_log_level(f_ctx, FREENECK_LOG_DEBUG);
173     int nr_devices = freenect_num_devices(f_ctx);
174     if (nr_devices < 1) {
175         cerr << "No Kinect devices found" << endl;
176         return -1;
177     }
178     if (freenect_open_device(f_ctx, &f_dev, 0) < 0) {
179         cerr << "Could not open Kinect device" << endl;
180         return -1;
181     }
182
183     // Print available frame modes
184     print_frame_modes();
185
186     // Set the video and depth modes
187     freenect_frame_mode video_mode = freenect_find_video_mode(
188         FREENECK_RESOLUTION_MEDIUM, FREENECK_VIDEO_RGB);
189     if (!video_mode.is_valid) {
190         cerr << "Invalid video mode" << endl;
191         return -1;
192     }
193     freenect_frame_mode depth_mode = freenect_find_depth_mode(
194         FREENECK_RESOLUTION_MEDIUM, FREENECK_DEPTH_11BIT);
195     if (!depth_mode.is_valid) {
196         cerr << "Invalid depth mode" << endl;
197         return -1;
198     }
199
200     if (freenect_set_video_mode(f_dev, video_mode) < 0) {
201         cerr << "Failed to set video mode" << endl;

```

```

194         return -1;
195     }
196
197     if (freenect_set_depth_mode(f_dev, depth_mode) < 0) {
198         cerr << "Failed to set depth mode" << endl;
199         return -1;
200     }
201
202     freenect_set_depth_callback(f_dev, depth_cb);
203     freenect_set_video_callback(f_dev, rgb_cb);
204     freenect_start_depth(f_dev);
205     freenect_start_video(f_dev);
206
207     // Start Kinect processing thread
208     std::thread freenect_thread(freenect_thread_func);
209
210     // Get available LiDAR ports
211     std::map<std::string, std::string> ports = ydlidar::
212         lidarPortList();
213     if (ports.size() == 1) {
214         port = ports.begin()->second;
215     } else {
216         int id = 0;
217         for (auto it = ports.begin(); it != ports.end(); it++) {
218             printf("%d. %s\n", id, it->first.c_str());
219             id++;
220         }
221         if (ports.empty()) {
222             printf("No LiDAR was detected. Please enter the LiDAR
223                 serial port:");
224             std::cin >> port;
225         } else {
226             while (ydlidar::os_isOk()) {
227                 printf("Please select the LiDAR port:");
228                 std::string number;
229                 std::cin >> number;
230                 if ((size_t)atoi(number.c_str()) >= ports.size())
231                     continue;
232                 auto it = ports.begin();
233                 id = atoi(number.c_str());
234                 while (id) {
235                     id--;
236                     it++;
237                 }
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
275     CYdLidar laser;
276     /////////////////////////////// string property///////////////////////
277     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
278     ());
279     std::string ignore_array;
280     ignore_array.clear();
281     laser.setlidaropt(LidarPropIgnoreArray, ignore_array.c_str(),
282     ignore_array.size());
283
284     /////////////////////////////// int property///////////////////////
285     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
286         int));
287     int optval = TYPE_TRIANGLE;
288     laser.setlidaropt(LidarPropLidarType, &optval, sizeof(int));
289     optval = YDLIDAR_TYPE_SERIAL;
290     laser.setlidaropt(LidarPropDeviceType, &optval, sizeof(int));
291     optval = isSingleChannel ? 3 : 4;
292     laser.setlidaropt(LidarPropSampleRate, &optval, sizeof(int));
293     optval = 4;
294     laser.setlidaropt(LidarPropAbnormalCheckCount, &optval, sizeof(
295         int));
296
297     /////////////////////////////// bool property///////////////////////
298     bool b_optvalue = false;
299     laser.setlidaropt(LidarPropFixedResolution, &b_optvalue,
300         sizeof(bool));
301     laser.setlidaropt(LidarPropReversion, &b_optvalue, sizeof(bool
302         ));
303     laser.setlidaropt(LidarPropInverted, &b_optvalue, sizeof(bool)
304         );
305     b_optvalue = true;
306     laser.setlidaropt(LidarPropAutoReconnect, &b_optvalue, sizeof(
307         bool));
308     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
309         sizeof(bool));
310     b_optvalue = false;

```

```

301     laser.setlidaropt(LidarPropIntenstiy, &b_optvalue, sizeof(bool
302         ));
303     b_optvalue = true;
304     laser.setlidaropt(LidarPropSupportMotorDtrCtrl, &b_optvalue,
305         sizeof(bool));
306     b_optvalue = false;
307     laser.setlidaropt(LidarPropSupportHeartBeat, &b_optvalue,
308         sizeof(bool));
309
310     //////////////////////////////float property///////////////////////
311     float f_optvalue = 180.0f;
312     laser.setlidaropt(LidarPropMaxAngle, &f_optvalue, sizeof(float
313         ));
314     f_optvalue = -180.0f;
315     laser.setlidaropt(LidarPropMinAngle, &f_optvalue, sizeof(float
316         ));
317     f_optvalue = 64.0f;
318     laser.setlidaropt(LidarPropMaxRange, &f_optvalue, sizeof(float
319         ));
320     f_optvalue = 0.05f;
321     laser.setlidaropt(LidarPropMinRange, &f_optvalue, sizeof(float
322         ));
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 {
332         cerr << "Error initializing YDLIDAR: " << laser.
333             DescribeError() << endl;
334         return -1;
335     }
336
337     sf::RenderWindow window(sf::VideoMode(1280, 720), "Camera and
338         LiDAR Visualization");
339
340     // Create SFML texture and sprite for the camera feed
341     sf::Texture cameraTexture;
342     sf::Sprite cameraSprite;
343
344     // Main loop
345     while (ret && window.isOpen() && ydlidar::os_isOk()) {
346         sf::Event event;
347         while (window.pollEvent(event)) {
348             if (event.type == sf::Event::Closed)
349                 window.close();
350         }
351
352         LaserScan scan;
353
354         if (laser.doProcessSimple(scan)) {
355             window.clear();
356
357             // Update the camera texture with the frame data
358             if (!cameraTexture.create(rgbMat.cols, rgbMat.rows)) {
359                 cerr << "Failed to create texture" << endl;
360                 break;

```

```

351
352     }
353
354     cameraTexture.update(rgbMat.ptr());
355
356     cameraSprite.setTexture(cameraTexture);
357     cameraSprite.setScale(
358         window.getSize().x / static_cast<float>(cameraTexture.getSize().x),
359         window.getSize().y / static_cast<float>(cameraTexture.getSize().y)
360     );
361
362     // Draw the camera feed
363     window.draw(cameraSprite);
364
365     // Draw the LiDAR minimap
366     sf::RenderTexture lidarTexture;
367     lidarTexture.create(300, 300);
368     lidarTexture.clear(sf::Color::White);
369
370     float gridSpacing = 20.0f; // Grid spacing in pixels
371     float offsetX = 150.0f; // Center of the minimap
372     float offsetY = 150.0f; // Center of the minimap
373     float scale = 50.0f; // Scale for LiDAR points
374
375     drawGrid(lidarTexture, gridSpacing, offsetX, offsetY);
376     drawNorthIndicator(lidarTexture, offsetX, offsetY);
377
378     // Detect and draw obstacles from the depth sensor
379     vector<Point2f> depthObstacles;
380     detectAndDrawDepthObstacles(depthMat, lidarTexture,
381         offsetX, offsetY, scale, depthObstacles);
382
383     // Draw the LiDAR itself (center point)
384     sf::CircleShape lidarShape(5); // Larger circle for
385         the center
386     lidarShape.setFillColor(sf::Color::Blue); // Blue
387         color for the center
388     lidarShape.setPosition(offsetX - lidarShape.getRadius()
389         (), offsetY - lidarShape.getRadius());
390     lidarTexture.draw(lidarShape);
391
392     // Draw the points from LiDAR
393     vector<Point2f> lidarObstacles;
394     for (const auto& point : scan.points) {
395         // Convert polar coordinates to Cartesian
396             coordinates
397         float x = point.range * cos(point.angle);
398         float y = point.range * sin(point.angle);
399
400         // Scale and translate points to fit minimap
401         float adjustedX = offsetX + x * scale;
402         float adjustedY = offsetY - y * scale; // Invert Y
403             to match screen coordinates
404
405         // Set point color based on distance
406         sf::Color pointColor = getPointColor(point.range,
407             64.0f); // Assuming max range is 64 meters
408
409         sf::CircleShape circle(2); // Small circle to
410             represent the point

```

```

401         circle.setPosition(adjustedX, adjustedY);
402         circle.setFillColor(pointColor);
403
404         lidarTexture.draw(circle);
405
406         // Add the obstacle to the vector
407         lidarObstacles.push_back(Point2f(adjustedX,
408                                     adjustedY));
409
410         // Compare the obstacles detected by the depth sensor
411         // and LiDAR
412         compareObstacles(depthObstacles, lidarObstacles);
413
414         lidarTexture.display();
415         sf::Sprite lidarSprite(lidarTexture.getTexture());
416         lidarSprite.setPosition(10, 10); // Position the
417         // minimap at the top-left corner
418         window.draw(lidarSprite);
419
420         window.display();
421     } else {
422         cerr << "Failed to get LiDAR data" << endl;
423     }
424
425     // Stop scanning
426     laser.turnOff();
427     laser.disconnecting();
428     is_running = false;
429     freenect_thread.join();
430     freenect_stop_depth(f_dev);
431     freenect_stop_video(f_dev);
432     freenect_close_device(f_dev);
433     freenect_shutdown(f_ctx);
434
435     return 0;
}

```

6.2.4. 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.

6.2.5. 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.

Listing 7: 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);
24
25     // Variables globales de prueba
26     std::atomic<bool> is_running(true);
27     std::atomic<bool> is_manual_mode(true);
28
29     // Funci\'on de prueba para el control de motores
30     void testMotors() {
31         // Inicializar pigpio
32         assert(gpioInitialise() >= 0 && "Error al inicializar pigpio."
33             );
34
35         // Configurar pines de direcci\'on como salida
36         for (int i = 0; i < 4; ++i) {
37             gpioSetMode(DIR_PINS[i], PI_OUTPUT);
38             gpioSetMode(PWM_PINS[i], PI_OUTPUT);
39             setMotorSpeed(i, frequency); // Inicializar PWM con
40             // frecuencia inicial
41         }
42
43         std::cout << "Prueba: Movimientos b\'asicos de los motores" <<
44             std::endl;
45
46         // Prueba de movimiento hacia adelante
47         moveForward();
48         std::this_thread::sleep_for(std::chrono::seconds(2));
49         assert(is_manual_mode == true && "Error: el modo manual no est
50             \'a activo durante la prueba de movimiento hacia adelante.
51             ");
52
53         // Prueba de movimiento hacia atr\'as
54         moveBackward();
55
56         // Prueba de giro
57         turnLeft();
58         std::this_thread::sleep_for(std::chrono::seconds(2));
59         turnRight();
60         std::this_thread::sleep_for(std::chrono::seconds(2));
61
62         // Prueba de velocidad constante
63         setMotorSpeed(0, 100);
64         std::this_thread::sleep_for(std::chrono::seconds(5));
65         setMotorSpeed(0, 0);
66         std::this_thread::sleep_for(std::chrono::seconds(2));
67
68         // Prueba de detecci\'on de obst\'aculos
69         testLidarObstacleDetection(laser);
70
71         // Prueba de lectura de datos
72         std::vector<float> data = laser.read();
73         assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
74
75         // Prueba de visualizaci\'on
76         visualization::Visualizer v;
77         v.setLidar(laser);
78         v.setMotors(motors);
79         v.setCameras(cameras);
80         v.setObstacles(obstacles);
81         v.setMap(map);
82         v.setGrid(grid);
83         v.setPath(path);
84         v.setTarget(target);
85         v.setObstacle_color("red");
86         v.setTarget_color("blue");
87         v.setGrid_color("gray");
88         v.setPath_color("green");
89         v.setTarget_size(100);
90         v.setGrid_size(10);
91         v.setPath_width(2);
92         v.setTarget_alpha(0.5);
93         v.setGrid_alpha(0.2);
94         v.setPath_alpha(0.8);
95         v.setTarget_strokeWidth(2);
96         v.setGrid_strokeWidth(1);
97         v.setPath_strokeWidth(2);
98
99         v.update();
100        v.render();
101    }
102
103    // Prueba de detecci\'on de obst\'aculos
104    void testLidarObstacleDetection(CYdLidar &laser) {
105        // Configurar pines de direcci\'on como salida
106        for (int i = 0; i < 4; ++i) {
107            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
108            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
109            setMotorSpeed(i, frequency); // Inicializar PWM con
110            // frecuencia inicial
111        }
112
113        // Prueba de movimiento hacia adelante
114        moveForward();
115        std::this_thread::sleep_for(std::chrono::seconds(2));
116        assert(is_manual_mode == true && "Error: el modo manual no est
117            \'a activo durante la prueba de movimiento hacia adelante.
118            ");
119
120        // Prueba de movimiento hacia atr\'as
121        moveBackward();
122        std::this_thread::sleep_for(std::chrono::seconds(2));
123        assert(is_manual_mode == true && "Error: el modo manual no est
124            \'a activo durante la prueba de movimiento hacia atr\'as.");
125
126        // Prueba de giro
127        turnLeft();
128        std::this_thread::sleep_for(std::chrono::seconds(2));
129        turnRight();
130        std::this_thread::sleep_for(std::chrono::seconds(2));
131
132        // Prueba de velocidad constante
133        setMotorSpeed(0, 100);
134        std::this_thread::sleep_for(std::chrono::seconds(5));
135        setMotorSpeed(0, 0);
136        std::this_thread::sleep_for(std::chrono::seconds(2));
137
138        // Prueba de detecci\'on de obst\'aculos
139        laser.setRange(0.1, 10.0);
140        laser.setAngle(-180, 180);
141        laser.setResolution(100);
142        laser.setUpdateRate(10);
143        laser.setLaserPower(100);
144        laser.setLaserFrequency(100000);
145        laser.setLaserWidth(10);
146        laser.setLaserOffset(0);
147        laser.setLaserMinAngle(-180);
148        laser.setLaserMaxAngle(180);
149        laser.setLaserMinRange(0.1);
150        laser.setLaserMaxRange(10.0);
151
152        laser.start();
153
154        while (is_running) {
155            laser.read();
156            laser.print();
157        }
158
159        laser.stop();
160    }
161
162    // Prueba de lectura de datos
163    void testLidar() {
164        // Configurar pines de direcci\'on como salida
165        for (int i = 0; i < 4; ++i) {
166            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
167            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
168            setMotorSpeed(i, frequency); // Inicializar PWM con
169            // frecuencia inicial
170        }
171
172        // Prueba de movimiento hacia adelante
173        moveForward();
174        std::this_thread::sleep_for(std::chrono::seconds(2));
175        assert(is_manual_mode == true && "Error: el modo manual no est
176            \'a activo durante la prueba de movimiento hacia adelante.
177            ");
178
179        // Prueba de movimiento hacia atr\'as
180        moveBackward();
181        std::this_thread::sleep_for(std::chrono::seconds(2));
182        assert(is_manual_mode == true && "Error: el modo manual no est
183            \'a activo durante la prueba de movimiento hacia atr\'as.");
184
185        // Prueba de giro
186        turnLeft();
187        std::this_thread::sleep_for(std::chrono::seconds(2));
188        turnRight();
189        std::this_thread::sleep_for(std::chrono::seconds(2));
190
191        // Prueba de velocidad constante
192        setMotorSpeed(0, 100);
193        std::this_thread::sleep_for(std::chrono::seconds(5));
194        setMotorSpeed(0, 0);
195        std::this_thread::sleep_for(std::chrono::seconds(2));
196
197        // Prueba de lectura de datos
198        std::vector<float> data = laser.read();
199        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
200
201        // Prueba de visualizaci\'on
202        visualization::Visualizer v;
203        v.setLidar(laser);
204        v.setMotors(motors);
205        v.setCameras(cameras);
206        v.setObstacles(obstacles);
207        v.setMap(map);
208        v.setGrid(grid);
209        v.setPath(path);
210        v.setTarget(target);
211        v.setObstacle_color("red");
212        v.setTarget_color("blue");
213        v.setGrid_color("gray");
214        v.setPath_color("green");
215        v.setTarget_alpha(0.5);
216        v.setGrid_alpha(0.2);
217        v.setPath_alpha(0.8);
218        v.setTarget_strokeWidth(2);
219        v.setGrid_strokeWidth(1);
220        v.setPath_strokeWidth(2);
221        v.setTarget_size(100);
222        v.setGrid_size(10);
223        v.setPath_width(2);
224        v.setTarget_alpha(0.5);
225        v.setGrid_alpha(0.2);
226        v.setPath_alpha(0.8);
227        v.setTarget_strokeWidth(2);
228        v.setGrid_strokeWidth(1);
229        v.setPath_strokeWidth(2);
230        v.setTarget_size(100);
231        v.setGrid_size(10);
232        v.setPath_width(2);
233
234        v.update();
235        v.render();
236    }
237
238    // Prueba de visualizaci\'on
239    void testVisualization() {
240        // Configurar pines de direcci\'on como salida
241        for (int i = 0; i < 4; ++i) {
242            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
243            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
244            setMotorSpeed(i, frequency); // Inicializar PWM con
245            // frecuencia inicial
246        }
247
248        // Prueba de movimiento hacia adelante
249        moveForward();
250        std::this_thread::sleep_for(std::chrono::seconds(2));
251        assert(is_manual_mode == true && "Error: el modo manual no est
252            \'a activo durante la prueba de movimiento hacia adelante.
253            ");
254
255        // Prueba de movimiento hacia atr\'as
256        moveBackward();
257        std::this_thread::sleep_for(std::chrono::seconds(2));
258        assert(is_manual_mode == true && "Error: el modo manual no est
259            \'a activo durante la prueba de movimiento hacia atr\'as.");
260
261        // Prueba de giro
262        turnLeft();
263        std::this_thread::sleep_for(std::chrono::seconds(2));
264        turnRight();
265        std::this_thread::sleep_for(std::chrono::seconds(2));
266
267        // Prueba de velocidad constante
268        setMotorSpeed(0, 100);
269        std::this_thread::sleep_for(std::chrono::seconds(5));
270        setMotorSpeed(0, 0);
271        std::this_thread::sleep_for(std::chrono::seconds(2));
272
273        // Prueba de visualizaci\'on
274        visualization::Visualizer v;
275        v.setLidar(laser);
276        v.setMotors(motors);
277        v.setCameras(cameras);
278        v.setObstacles(obstacles);
279        v.setMap(map);
280        v.setGrid(grid);
281        v.setPath(path);
282        v.setTarget(target);
283        v.setObstacle_color("red");
284        v.setTarget_color("blue");
285        v.setGrid_color("gray");
286        v.setPath_color("green");
287        v.setTarget_alpha(0.5);
288        v.setGrid_alpha(0.2);
289        v.setPath_alpha(0.8);
290        v.setTarget_strokeWidth(2);
291        v.setGrid_strokeWidth(1);
292        v.setPath_strokeWidth(2);
293        v.setTarget_size(100);
294        v.setGrid_size(10);
295        v.setPath_width(2);
296        v.setTarget_alpha(0.5);
297        v.setGrid_alpha(0.2);
298        v.setPath_alpha(0.8);
299        v.setTarget_strokeWidth(2);
300        v.setGrid_strokeWidth(1);
301        v.setPath_strokeWidth(2);
302        v.setTarget_size(100);
303        v.setGrid_size(10);
304        v.setPath_width(2);
305
306        v.update();
307        v.render();
308    }
309
310    // Prueba de control de motores
311    void testMotors() {
312        // Inicializar pigpio
313        assert(gpioInitialise() >= 0 && "Error al inicializar pigpio."
314            );
315
316        // Configurar pines de direcci\'on como salida
317        for (int i = 0; i < 4; ++i) {
318            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
319            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
320            setMotorSpeed(i, frequency); // Inicializar PWM con
321            // frecuencia inicial
322        }
323
324        // Prueba de movimiento hacia adelante
325        moveForward();
326        std::this_thread::sleep_for(std::chrono::seconds(2));
327        assert(is_manual_mode == true && "Error: el modo manual no est
328            \'a activo durante la prueba de movimiento hacia adelante.
329            ");
330
331        // Prueba de movimiento hacia atr\'as
332        moveBackward();
333        std::this_thread::sleep_for(std::chrono::seconds(2));
334        assert(is_manual_mode == true && "Error: el modo manual no est
335            \'a activo durante la prueba de movimiento hacia atr\'as.");
336
337        // Prueba de giro
338        turnLeft();
339        std::this_thread::sleep_for(std::chrono::seconds(2));
340        turnRight();
341        std::this_thread::sleep_for(std::chrono::seconds(2));
342
343        // Prueba de velocidad constante
344        setMotorSpeed(0, 100);
345        std::this_thread::sleep_for(std::chrono::seconds(5));
346        setMotorSpeed(0, 0);
347        std::this_thread::sleep_for(std::chrono::seconds(2));
348
349        // Prueba de control de motores
350        std::vector<float> data = laser.read();
351        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
352
353        // Prueba de visualizaci\'on
354        visualization::Visualizer v;
355        v.setLidar(laser);
356        v.setMotors(motors);
357        v.setCameras(cameras);
358        v.setObstacles(obstacles);
359        v.setMap(map);
360        v.setGrid(grid);
361        v.setPath(path);
362        v.setTarget(target);
363        v.setObstacle_color("red");
364        v.setTarget_color("blue");
365        v.setGrid_color("gray");
366        v.setPath_color("green");
367        v.setTarget_alpha(0.5);
368        v.setGrid_alpha(0.2);
369        v.setPath_alpha(0.8);
370        v.setTarget_strokeWidth(2);
371        v.setGrid_strokeWidth(1);
372        v.setPath_strokeWidth(2);
373        v.setTarget_size(100);
374        v.setGrid_size(10);
375        v.setPath_width(2);
376        v.setTarget_alpha(0.5);
377        v.setGrid_alpha(0.2);
378        v.setPath_alpha(0.8);
379        v.setTarget_strokeWidth(2);
380        v.setGrid_strokeWidth(1);
381        v.setPath_strokeWidth(2);
382        v.setTarget_size(100);
383        v.setGrid_size(10);
384        v.setPath_width(2);
385
386        v.update();
387        v.render();
388    }
389
390    // Prueba de detecci\'on de obst\'aculos
391    void testLidar() {
392        // Configurar pines de direcci\'on como salida
393        for (int i = 0; i < 4; ++i) {
394            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
395            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
396            setMotorSpeed(i, frequency); // Inicializar PWM con
397            // frecuencia inicial
398        }
399
400        // Prueba de movimiento hacia adelante
401        moveForward();
402        std::this_thread::sleep_for(std::chrono::seconds(2));
403        assert(is_manual_mode == true && "Error: el modo manual no est
404            \'a activo durante la prueba de movimiento hacia adelante.
405            ");
406
407        // Prueba de movimiento hacia atr\'as
408        moveBackward();
409        std::this_thread::sleep_for(std::chrono::seconds(2));
410        assert(is_manual_mode == true && "Error: el modo manual no est
411            \'a activo durante la prueba de movimiento hacia atr\'as.");
412
413        // Prueba de giro
414        turnLeft();
415        std::this_thread::sleep_for(std::chrono::seconds(2));
416        turnRight();
417        std::this_thread::sleep_for(std::chrono::seconds(2));
418
419        // Prueba de velocidad constante
420        setMotorSpeed(0, 100);
421        std::this_thread::sleep_for(std::chrono::seconds(5));
422        setMotorSpeed(0, 0);
423        std::this_thread::sleep_for(std::chrono::seconds(2));
424
425        // Prueba de detecci\'on de obst\'aculos
426        std::vector<float> data = laser.read();
427        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
428
429        // Prueba de visualizaci\'on
430        visualization::Visualizer v;
431        v.setLidar(laser);
432        v.setMotors(motors);
433        v.setCameras(cameras);
434        v.setObstacles(obstacles);
435        v.setMap(map);
436        v.setGrid(grid);
437        v.setPath(path);
438        v.setTarget(target);
439        v.setObstacle_color("red");
440        v.setTarget_color("blue");
441        v.setGrid_color("gray");
442        v.setPath_color("green");
443        v.setTarget_alpha(0.5);
444        v.setGrid_alpha(0.2);
445        v.setPath_alpha(0.8);
446        v.setTarget_strokeWidth(2);
447        v.setGrid_strokeWidth(1);
448        v.setPath_strokeWidth(2);
449        v.setTarget_size(100);
450        v.setGrid_size(10);
451        v.setPath_width(2);
452        v.setTarget_alpha(0.5);
453        v.setGrid_alpha(0.2);
454        v.setPath_alpha(0.8);
455        v.setTarget_strokeWidth(2);
456        v.setGrid_strokeWidth(1);
457        v.setPath_strokeWidth(2);
458        v.setTarget_size(100);
459        v.setGrid_size(10);
460        v.setPath_width(2);
461
462        v.update();
463        v.render();
464    }
465
466    // Prueba de lectura de datos
467    void testLidarObstacleDetection() {
468        // Configurar pines de direcci\'on como salida
469        for (int i = 0; i < 4; ++i) {
470            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
471            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
472            setMotorSpeed(i, frequency); // Inicializar PWM con
473            // frecuencia inicial
474        }
475
476        // Prueba de movimiento hacia adelante
477        moveForward();
478        std::this_thread::sleep_for(std::chrono::seconds(2));
479        assert(is_manual_mode == true && "Error: el modo manual no est
480            \'a activo durante la prueba de movimiento hacia adelante.
481            ");
482
483        // Prueba de movimiento hacia atr\'as
484        moveBackward();
485        std::this_thread::sleep_for(std::chrono::seconds(2));
486        assert(is_manual_mode == true && "Error: el modo manual no est
487            \'a activo durante la prueba de movimiento hacia atr\'as.");
488
489        // Prueba de giro
490        turnLeft();
491        std::this_thread::sleep_for(std::chrono::seconds(2));
492        turnRight();
493        std::this_thread::sleep_for(std::chrono::seconds(2));
494
495        // Prueba de velocidad constante
496        setMotorSpeed(0, 100);
497        std::this_thread::sleep_for(std::chrono::seconds(5));
498        setMotorSpeed(0, 0);
499        std::this_thread::sleep_for(std::chrono::seconds(2));
500
501        // Prueba de lectura de datos
502        std::vector<float> data = laser.read();
503        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
504
505        // Prueba de visualizaci\'on
506        visualization::Visualizer v;
507        v.setLidar(laser);
508        v.setMotors(motors);
509        v.setCameras(cameras);
510        v.setObstacles(obstacles);
511        v.setMap(map);
512        v.setGrid(grid);
513        v.setPath(path);
514        v.setTarget(target);
515        v.setObstacle_color("red");
516        v.setTarget_color("blue");
517        v.setGrid_color("gray");
518        v.setPath_color("green");
519        v.setTarget_alpha(0.5);
520        v.setGrid_alpha(0.2);
521        v.setPath_alpha(0.8);
522        v.setTarget_strokeWidth(2);
523        v.setGrid_strokeWidth(1);
524        v.setPath_strokeWidth(2);
525        v.setTarget_size(100);
526        v.setGrid_size(10);
527        v.setPath_width(2);
528        v.setTarget_alpha(0.5);
529        v.setGrid_alpha(0.2);
530        v.setPath_alpha(0.8);
531        v.setTarget_strokeWidth(2);
532        v.setGrid_strokeWidth(1);
533        v.setPath_strokeWidth(2);
534        v.setTarget_size(100);
535        v.setGrid_size(10);
536        v.setPath_width(2);
537
538        v.update();
539        v.render();
540    }
541
542    // Prueba de visualizaci\'on
543    void testVisualization() {
544        // Configurar pines de direcci\'on como salida
545        for (int i = 0; i < 4; ++i) {
546            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
547            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
548            setMotorSpeed(i, frequency); // Inicializar PWM con
549            // frecuencia inicial
550        }
551
552        // Prueba de movimiento hacia adelante
553        moveForward();
554        std::this_thread::sleep_for(std::chrono::seconds(2));
555        assert(is_manual_mode == true && "Error: el modo manual no est
556            \'a activo durante la prueba de movimiento hacia adelante.
557            ");
558
559        // Prueba de movimiento hacia atr\'as
560        moveBackward();
561        std::this_thread::sleep_for(std::chrono::seconds(2));
562        assert(is_manual_mode == true && "Error: el modo manual no est
563            \'a activo durante la prueba de movimiento hacia atr\'as.");
564
565        // Prueba de giro
566        turnLeft();
567        std::this_thread::sleep_for(std::chrono::seconds(2));
568        turnRight();
569        std::this_thread::sleep_for(std::chrono::seconds(2));
570
571        // Prueba de velocidad constante
572        setMotorSpeed(0, 100);
573        std::this_thread::sleep_for(std::chrono::seconds(5));
574        setMotorSpeed(0, 0);
575        std::this_thread::sleep_for(std::chrono::seconds(2));
576
577        // Prueba de visualizaci\'on
578        visualization::Visualizer v;
579        v.setLidar(laser);
580        v.setMotors(motors);
581        v.setCameras(cameras);
582        v.setObstacles(obstacles);
583        v.setMap(map);
584        v.setGrid(grid);
585        v.setPath(path);
586        v.setTarget(target);
587        v.setObstacle_color("red");
588        v.setTarget_color("blue");
589        v.setGrid_color("gray");
590        v.setPath_color("green");
591        v.setTarget_alpha(0.5);
592        v.setGrid_alpha(0.2);
593        v.setPath_alpha(0.8);
594        v.setTarget_strokeWidth(2);
595        v.setGrid_strokeWidth(1);
596        v.setPath_strokeWidth(2);
597        v.setTarget_size(100);
598        v.setGrid_size(10);
599        v.setPath_width(2);
600        v.setTarget_alpha(0.5);
601        v.setGrid_alpha(0.2);
602        v.setPath_alpha(0.8);
603        v.setTarget_strokeWidth(2);
604        v.setGrid_strokeWidth(1);
605        v.setPath_strokeWidth(2);
606        v.setTarget_size(100);
607        v.setGrid_size(10);
608        v.setPath_width(2);
609
610        v.update();
611        v.render();
612    }
613
614    // Prueba de control de motores
615    void testMotors() {
616        // Inicializar pigpio
617        assert(gpioInitialise() >= 0 && "Error al inicializar pigpio."
618            );
619
620        // Configurar pines de direcci\'on como salida
621        for (int i = 0; i < 4; ++i) {
622            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
623            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
624            setMotorSpeed(i, frequency); // Inicializar PWM con
625            // frecuencia inicial
626        }
627
628        // Prueba de movimiento hacia adelante
629        moveForward();
630        std::this_thread::sleep_for(std::chrono::seconds(2));
631        assert(is_manual_mode == true && "Error: el modo manual no est
632            \'a activo durante la prueba de movimiento hacia adelante.
633            ");
634
635        // Prueba de movimiento hacia atr\'as
636        moveBackward();
637        std::this_thread::sleep_for(std::chrono::seconds(2));
638        assert(is_manual_mode == true && "Error: el modo manual no est
639            \'a activo durante la prueba de movimiento hacia atr\'as.");
640
641        // Prueba de giro
642        turnLeft();
643        std::this_thread::sleep_for(std::chrono::seconds(2));
644        turnRight();
645        std::this_thread::sleep_for(std::chrono::seconds(2));
646
647        // Prueba de velocidad constante
648        setMotorSpeed(0, 100);
649        std::this_thread::sleep_for(std::chrono::seconds(5));
650        setMotorSpeed(0, 0);
651        std::this_thread::sleep_for(std::chrono::seconds(2));
652
653        // Prueba de control de motores
654        std::vector<float> data = laser.read();
655        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
656
657        // Prueba de visualizaci\'on
658        visualization::Visualizer v;
659        v.setLidar(laser);
660        v.setMotors(motors);
661        v.setCameras(cameras);
662        v.setObstacles(obstacles);
663        v.setMap(map);
664        v.setGrid(grid);
665        v.setPath(path);
666        v.setTarget(target);
667        v.setObstacle_color("red");
668        v.setTarget_color("blue");
669        v.setGrid_color("gray");
670        v.setPath_color("green");
671        v.setTarget_alpha(0.5);
672        v.setGrid_alpha(0.2);
673        v.setPath_alpha(0.8);
674        v.setTarget_strokeWidth(2);
675        v.setGrid_strokeWidth(1);
676        v.setPath_strokeWidth(2);
677        v.setTarget_size(100);
678        v.setGrid_size(10);
679        v.setPath_width(2);
680        v.setTarget_alpha(0.5);
681        v.setGrid_alpha(0.2);
682        v.setPath_alpha(0.8);
683        v.setTarget_strokeWidth(2);
684        v.setGrid_strokeWidth(1);
685        v.setPath_strokeWidth(2);
686        v.setTarget_size(100);
687        v.setGrid_size(10);
688        v.setPath_width(2);
689
690        v.update();
691        v.render();
692    }
693
694    // Prueba de detecci\'on de obst\'aculos
695    void testLidar() {
696        // Configurar pines de direcci\'on como salida
697        for (int i = 0; i < 4; ++i) {
698            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
699            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
700            setMotorSpeed(i, frequency); // Inicializar PWM con
701            // frecuencia inicial
702        }
703
704        // Prueba de movimiento hacia adelante
705        moveForward();
706        std::this_thread::sleep_for(std::chrono::seconds(2));
707        assert(is_manual_mode == true && "Error: el modo manual no est
708            \'a activo durante la prueba de movimiento hacia adelante.
709            ");
710
711        // Prueba de movimiento hacia atr\'as
712        moveBackward();
713        std::this_thread::sleep_for(std::chrono::seconds(2));
714        assert(is_manual_mode == true && "Error: el modo manual no est
715            \'a activo durante la prueba de movimiento hacia atr\'as.");
716
717        // Prueba de giro
718        turnLeft();
719        std::this_thread::sleep_for(std::chrono::seconds(2));
720        turnRight();
721        std::this_thread::sleep_for(std::chrono::seconds(2));
722
723        // Prueba de velocidad constante
724        setMotorSpeed(0, 100);
725        std::this_thread::sleep_for(std::chrono::seconds(5));
726        setMotorSpeed(0, 0);
727        std::this_thread::sleep_for(std::chrono::seconds(2));
728
729        // Prueba de detecci\'on de obst\'aculos
730        std::vector<float> data = laser.read();
731        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
732
733        // Prueba de visualizaci\'on
734        visualization::Visualizer v;
735        v.setLidar(laser);
736        v.setMotors(motors);
737        v.setCameras(cameras);
738        v.setObstacles(obstacles);
739        v.setMap(map);
740        v.setGrid(grid);
741        v.setPath(path);
742        v.setTarget(target);
743        v.setObstacle_color("red");
744        v.setTarget_color("blue");
745        v.setGrid_color("gray");
746        v.setPath_color("green");
747        v.setTarget_alpha(0.5);
748        v.setGrid_alpha(0.2);
749        v.setPath_alpha(0.8);
750        v.setTarget_strokeWidth(2);
751        v.setGrid_strokeWidth(1);
752        v.setPath_strokeWidth(2);
753        v.setTarget_size(100);
754        v.setGrid_size(10);
755        v.setPath_width(2);
756        v.setTarget_alpha(0.5);
757        v.setGrid_alpha(0.2);
758        v.setPath_alpha(0.8);
759        v.setTarget_strokeWidth(2);
760        v.setGrid_strokeWidth(1);
761        v.setPath_strokeWidth(2);
762        v.setTarget_size(100);
763        v.setGrid_size(10);
764        v.setPath_width(2);
765
766        v.update();
767        v.render();
768    }
769
770    // Prueba de lectura de datos
771    void testLidarObstacleDetection() {
772        // Configurar pines de direcci\'on como salida
773        for (int i = 0; i < 4; ++i) {
774            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
775            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
776            setMotorSpeed(i, frequency); // Inicializar PWM con
777            // frecuencia inicial
778        }
779
780        // Prueba de movimiento hacia adelante
781        moveForward();
782        std::this_thread::sleep_for(std::chrono::seconds(2));
783        assert(is_manual_mode == true && "Error: el modo manual no est
784            \'a activo durante la prueba de movimiento hacia adelante.
785            ");
786
787        // Prueba de movimiento hacia atr\'as
788        moveBackward();
789        std::this_thread::sleep_for(std::chrono::seconds(2));
790        assert(is_manual_mode == true && "Error: el modo manual no est
791            \'a activo durante la prueba de movimiento hacia atr\'as.");
792
793        // Prueba de giro
794        turnLeft();
795        std::this_thread::sleep_for(std::chrono::seconds(2));
796        turnRight();
797        std::this_thread::sleep_for(std::chrono::seconds(2));
798
799        // Prueba de velocidad constante
800        setMotorSpeed(0, 100);
801        std::this_thread::sleep_for(std::chrono::seconds(5));
802        setMotorSpeed(0, 0);
803        std::this_thread::sleep_for(std::chrono::seconds(2));
804
805        // Prueba de lectura de datos
806        std::vector<float> data = laser.read();
807        assert(data.size() > 0 && "Error: no se leyeron datos de LiDAR");
808
809        // Prueba de visualizaci\'on
810        visualization::Visualizer v;
811        v.setLidar(laser);
812        v.setMotors(motors);
813        v.setCameras(cameras);
814        v.setObstacles(obstacles);
815        v.setMap(map);
816        v.setGrid(grid);
817        v.setPath(path);
818        v.setTarget(target);
819        v.setObstacle_color("red");
820        v.setTarget_color("blue");
821        v.setGrid_color("gray");
822        v.setPath_color("green");
823        v.setTarget_alpha(0.5);
824        v.setGrid_alpha(0.2);
825        v.setPath_alpha(0.8);
826        v.setTarget_strokeWidth(2);
827        v.setGrid_strokeWidth(1);
828        v.setPath_strokeWidth(2);
829        v.setTarget_size(100);
830        v.setGrid_size(10);
831        v.setPath_width(2);
832        v.setTarget_alpha(0.5);
833        v.setGrid_alpha(0.2);
834        v.setPath_alpha(0.8);
835        v.setTarget_strokeWidth(2);
836        v.setGrid_strokeWidth(1);
837        v.setPath_strokeWidth(2);
838        v.setTarget_size(100);
839        v.setGrid_size(10);
840        v.setPath_width(2);
841
842        v.update();
843        v.render();
844    }
845
846    // Prueba de visualizaci\'on
847    void testVisualization() {
848        // Configurar pines de direcci\'on como salida
849        for (int i = 0; i < 4; ++i) {
850            gpioSetMode(DIR_PINS[i], PI_OUTPUT);
851            gpioSetMode(PWM_PINS[i], PI_OUTPUT);
852            setMotorSpeed(i, frequency); // Inicializar PWM con
853            // frecuencia inicial
854        }
855
856        // Prueba de movimiento hacia adelante
857        moveForward();
858        std::this_thread::sleep_for(std::chrono::seconds(2));
859        assert(is_manual_mode == true && "Error: el modo manual no est
860            \'a activo durante la prueba de movimiento hacia adelante.
861            ");
862
863        // Prueba de movimiento hacia atr\'as
864        moveBackward();
865        std::this_thread::sleep_for(std::chrono::seconds(2));
866        assert(is_manual_mode == true && "Error: el modo manual no est
867            \'a activo durante la prueba de movimiento hacia atr\'as.");
868
869        // Prueba de giro
870        turnLeft();
871        std::this_thread::sleep_for(std::chrono::seconds(2));
872        turnRight();
873        std::this_thread::sleep_for(std::chrono::seconds(2));
874
875        // Prueba de velocidad constante
876        setMotorSpeed(0, 100);
877        std::this_thread::sleep_for(std::chrono::seconds(5));
878        setMotorSpeed(0, 0);
879        std::this_thread::sleep_for(std::chrono::seconds(2));
880
881        // Prueba de visualizaci\'on
882        visualization::Visualizer v;
883        v.setLidar(laser);
884        v.setMotors(motors);
885        v.setCameras(cameras);
886        v.setObstacles(obstacles);
887        v.setMap(map);
888        v.setGrid(grid);
889        v.setPath(path);
890        v.setTarget(target);
891        v.setObstacle_color("red");
892        v.setTarget_color("blue");
893        v.setGrid_color("gray");
894        v.setPath_color("green");
895        v.setTarget_alpha(0.5);
896        v.setGrid_alpha(0.2);
897        v.setPath_alpha(0.8);
898        v.setTarget_strokeWidth(2);
899        v.setGrid_strokeWidth(1);
900        v.setPath_strokeWidth(2);
901        v.setTarget_size(100);
902        v.setGrid_size(10);
903        v.setPath_width(2);
904        v.setTarget_alpha(0.5);
905        v.setGrid_alpha(0.2);
906        v.setPath_alpha(0.8);
907        v.setTarget_strokeWidth(2);
908        v.setGrid_strokeWidth(1);
909        v.setPath_strokeWidth(2);
910        v.setTarget_size(100);
911        v.setGrid_size(10);
912        v.setPath_width(2);
913
914        v.update();
915        v.render();
916    }
917
918    // Prueba de control de motores
```

```

49         std::this_thread::sleep_for(std::chrono::seconds(2));
50
51     // Prueba de giro a la izquierda
52     turnLeft();
53     std::this_thread::sleep_for(std::chrono::seconds(2));
54
55     // Prueba de giro a la derecha
56     turnRight();
57     std::this_thread::sleep_for(std::chrono::seconds(2));
58
59     // Detener los motores
60     stopMotors();
61     std::cout << "Prueba de motores completada correctamente." <<
62         std::endl;
63
64     gpioTerminate();
65 }
66
67 // Funci\on de prueba para el sensor LiDAR
68 void testLidar() {
69     std::string port;
70     ydlidar::os_init();
71
72     // Obtener los puertos disponibles de LiDAR
73     std::map<std::string, std::string> ports = ydlidar::
74         lidarPortList();
75     if (ports.size() > 1) {
76         auto it = ports.begin();
77         std::advance(it, 1); // Selecciona el segundo puerto
78             disponible
79         port = it->second;
80     } else if (ports.size() == 1) {
81         port = ports.begin()->second;
82     } else {
83         std::cerr << "No se detect\o ning\un LiDAR. Verifica la
84             conexi\on." << std::endl;
85         assert(false && "Error: No se detect\o LiDAR.");
86         return;
87     }
88
89     CYdLidar laser;
90     int baudrate = 115200;
91     laser.setlidaropt(LidarPropSerialPort, port.c_str(), port.size
92         ());
93     laser.setlidaropt(LidarPropSerialBaudrate, &baudrate, sizeof(
94         int));
95
96     bool isSingleChannel = true;
97     laser.setlidaropt(LidarPropSingleChannel, &isSingleChannel,
98         sizeof(bool));
99
100    float max_range = 8.0f;
101    float min_range = 0.1f;
102    float max_angle = 180.0f;
103    float min_angle = -180.0f;
104    float frequency = 8.0f;
105
106    laser.setlidaropt(LidarPropMaxRange, &max_range, sizeof(float)
107        );
108    laser.setlidaropt(LidarPropMinRange, &min_range, sizeof(float)

```

```

        );
101    laser.setlidaropt(LidarPropMaxAngle, &max_angle, sizeof(float)
        );
102    laser.setlidaropt(LidarPropMinAngle, &min_angle, sizeof(float)
        );
103    laser.setlidaropt(LidarPropScanFrequency, &frequency, sizeof(
        float));
104
105    // Inicializar LiDAR
106    assert(laser.initialize() && "Error al inicializar el LiDAR.")
        ;
107    assert(laser.turnOn() && "Error al encender el LiDAR.");
108
109    std::cout << "Prueba: LiDAR encendido y funcionando." << std::
        endl;
110
111    // Simular un escaneo
112    LaserScan scan;
113    assert(laser.doProcessSimple(scan) && "Error al procesar el
        escaneo LiDAR.");
114    std::cout << "Prueba de escaneo LiDAR completada correctamente
        ." << std::endl;
115
116    // Apagar LiDAR
117    laser.turnOff();
118    laser.disconnecting();
119}
120
121 // Función de prueba para la detección de obstáculos con
122 // LiDAR
123 void testLidarObstacleDetection(CYdLidar &laser) {
124     // Prueba de detección de obstáculos con el LiDAR
125     LaserScan scan;
126     if (laser.doProcessSimple(scan)) {
127         bool obstacle_detected = false;
128         for (const auto &point : scan.points) {
129             if (point.range > 0 && point.range < 0.30) { // //
130                 Condición de obstáculo
131                 obstacle_detected = true;
132                 break;
133             }
134         }
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 integraci\'on se
152     completaron correctamente." << std::endl;
153
154     return 0;
155 }
```

Las pruebas iniciales de integración mostraron que el robot es capaz de realizar todas las funciones previstas: moverse, detectar obstáculos y visualizar su entorno en tiempo real. Los resultados indican que los módulos están correctamente integrados y listos para pruebas más avanzadas en entornos complejos.

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.