UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Aplicación de Herramientas de Aprendizaje Reforzado y Aprendizaje Profundo en Simulaciones de Robótica de Enjambre con Restricciones Físicas

Trabajo de graduación presentado por Marco Antonio Izeppi Rosales
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2022

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

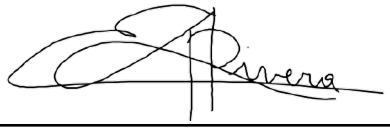
Aplicación de Herramientas de Aprendizaje Reforzado y Aprendizaje Profundo en Simulaciones de Robótica de Enjambre con Restricciones Físicas

Trabajo de graduación presentado por Marco Antonio Izeppi Rosales
para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

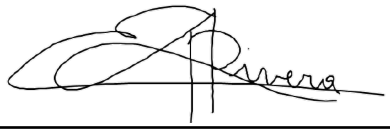
2022

Vo.Bo.:

(f) 

Dr. Luis Alberto Rivera Estrada

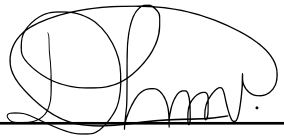
Tribunal Examinador:

(f) 

Dr. Luis Alberto Rivera Estrada

(f) 

MSc. Miguel Enrique Zea Arenales

(f) 

Ing. Diego Alberto Morales Ibáñez

Fecha de aprobación: Guatemala, 5 de Enero de 2022.

Prefacio

La idea de este trabajo que lleva como título "Aplicación de Herramientas de Aprendizaje Reforzado y Aprendizaje Profundo en Simulaciones de Robótica de Enjambre con Restricciones Físicas" es establecer las bases para llegar a desarrollar proyectos cada vez mas complicados dentro del entorno de robótica. Sentando las bases para futuras experimentaciones dentro de la Universidad del Valle de Guatemala.

Le agradezco a todos los miembros que conforman la comunidad universitaria los cuales me apoyaron para desarrollar este proyecto. Cada persona compartió un poco de su conocimiento sobre distintas áreas para lograr que este proyecto fuera posible. Mis más grandes agradecimientos al Dr. Luis Alberto Rivera por acompañarme a lo largo del desarrollo de esta tesis y a MSc. Miguel Zea por introducirme al mundo de la robótica. A mis compañeros que siempre estuvieron presentes para apoyarme y darme ánimos de seguir adelante y lograr alcanzar esta meta.

Finalmente, le agradezco a mi madre quien me brindó el privilegio de estudiar en la Universidad del Valle de Guatemala y a mi familia por siempre apoyarme en las decisiones que tomé para llegar a este punto.

Índice

Lista de figuras

Lista de cuadros

Resumen

El algoritmo de optimización por enjambre de partículas (*Particle Swarm Optimization - PSO*) hace referencia a un método heurístico para resolver un tipo de problema computacional que evoca el comportamiento de las partículas en la naturaleza. En un principio fue concebido para elaborar modelos de conductas sociales, como el movimiento descrito por los organismos vivos en una bandada de aves, enjambres de insectos o un banco de peces. Posteriormente el algoritmo se simplificó y se comprobó que era adecuado para problemas de optimización utilizando partículas que simulan un ser individual dentro del enjambre.

Tomando esto en consideración, en las fases previas a este proyecto se desarrolló un algoritmo que utiliza métodos de Aprendizaje automático como lo es el aprendizaje profundo y el aprendizaje reforzado con la finalidad de crear un *Toolbox* en Matlab que optimice la selección de los parámetros que ingresan al sistema para mejorar la velocidad de convergencia de las partículas hacia el mínimo global.

En la siguiente tesis se describe implementación de este algoritmo, previamente diseñado, en un entorno de simulación con restricciones físicas, tomando en cuenta las dimensiones del robot y que es capaz de evadir obstáculos colocados dentro del entorno de simulación de forma aleatoria. Este algoritmo ya implementado en un sistema físico busca una rápida convergencia hacia el mínimo global al igual que evitar colisiones entre los robots o con los obstáculos. Por otro lado, se evalúa la viabilidad de la realización de simulaciones de sistemas multi-robot en el entorno de *Robotic Operating System* (ROS) debido a que es un *software* popular en el área de robótica, contiene diversas funcionalidades y presenta buenas simulaciones de condiciones físicas.

Abstract

The Particle Swarm Optimization algorithm refers to a heuristic method for solving a type of computational problem that evokes the behavior of particles in nature. They were originally conceived to model social behavior, such as the movement described by living organisms in a flock of birds, swarms of insects, or a bank of fish. Later the algorithm was simplified and it was found to be suitable for optimization problems using particles that simulate an individual being within the swarm.

The PSO algorithm has several variants applied to various objectives such as multi-objective optimization, binary PSO, discrete PSO, combinatorial PSO, etc. Taking this into consideration, the Universidad del Valle de Guatemala began with a variant applied to mobile robots with the Robotat mega project, where the algorithm was part of the navigation system that differential robots would use to converge to a specific point.

Taking this into consideration, in the phases prior to this project, an algorithm was developed that uses machine learning methods such as deep learning and reinforced learning in order to create a Toolbox in Matlab that optimizes the selection of the input parameters of the system to improve the speed at which the particles converge towards the global minimum.

In this thesis, a graduation work is proposed that seeks an implementation of this algorithm, previously designed, in a simulation environment with physical restrictions, which takes into account the dimensions of the robot and is capable of avoiding obstacles placed within the simulation environment. This algorithm already implemented in a physical system seeks a rapid convergence towards the global minimum as well as avoiding collisions between robots or with obstacles. On the other hand, it seeks to evaluate the feasibility of conducting simulations of multi-robot systems in the Robotic Operating System (ROS) environment because it is popular software in the robotics area and presents good simulations of physical conditions.

CAPÍTULO 1

Introducción

El algoritmo de Optimización por Enjambres de Partículas fue por primera vez introducido por Kennedy y Eberhart en su trabajo *Particle swarm optimization (PSO)* en 1995 [1]. Desde entonces, este algoritmo se ha utilizado en diversas ramas de la ingeniería, tales como la informática, civil, informática, etc. Este algoritmo se utiliza para resolver una gran cantidad de problemas relacionados con la optimización de funciones para encontrar mínimos y máximos, dependiendo de la aplicación.

En el caso de la Universidad del Valle de Guatemala, se empezó con el proyecto de Robotat en el cual se propuso la utilización del algoritmo de PSO como base para la navegación del conjunto de robots diferenciales. A partir de esto, se logró desarrollar un algoritmo capaz de tomar en cuenta las dimensión físicas de los robots diferenciales utilizados en el algoritmo PSO. Tomando este algoritmo como base, Eduardo Santizo [2] desarrolló un *Toolbox* capaz de calcular los hiper-parámetros del sistema para obtener un comportamiento deseable de los robots diferenciales y lograr su convergencia.

En el presente trabajo se busca la utilización del *Toolbox* diseñado por Eduardo Santizo para su implementación en simulaciones que no presenten condiciones ideales, tomando como parámetros base los valores dados por el *Swarm Robotics Toolbox* y *PSO Tuner*. También se tomó en consideración que el terreno que los robots exploran es un terreno desconocido con obstáculos colocados aleatoriamente dentro de el entorno. Las simulaciones de estos comportamientos se realizaran dentro de el entorno de Webots para tomar en consideración restricciones físicas tales como las dimensiones de los robots diferenciales, la gravedad, el rango de muestreo de los sensores, etc.

Se desea evaluar la viabilidad de la implementación del algoritmo PSO dentro del entorno de ROS ya que es un entorno que es ampliamente utilizado en la industria y posee diversas funcionalidades que nos permiten obtener mas información sobre el entorno y el funcionamiento del sistema. El entorno posee diversos programas para visualización, simulación, discretización de espacios, etc.

CAPÍTULO 2

Antecedentes

El algoritmo PSO posee diversas variantes aplicadas a diversos objetivos tales como optimización multiobjetivo, PSO binaria, PSO discreta, PSO combinatoria, etc. Tomando esto en consideración, dentro de la Universidad del Valle de Guatemala se inició con una variante aplicada a robots móviles con el mega proyecto Robotat en donde el algoritmo formaba parte del sistema de navegación que utilizarían los robots diferenciales para converger a un punto específico [3].

2.1. Optimización de trayectorias de robots móviles en ambientes desconocidos

Safa Ziadi, Mohamed Njah y Mohamed Chtourou [4] utilizaron el enfoque de optimización de enjambre de partículas multiobjetivo (PSO) para optimizar los parámetros del Método Canónico de Campo de Fuerza. El cálculo de los parámetros óptimos se reinicia en cada nueva posición del robot y el PSO se utiliza para minimizar la distancia entre la posición y la meta a la cual se desea llegar y también para maximizar la distancia segura entre la posición actual del robot y los obstáculos cercanos. La eficacia del método se demuestra mediante simulaciones en el entorno de Webots. Las simulaciones se llevan a cabo en varios entornos conocidos y desconocidos. En los entornos conocidos, el robot reconoce la posición del obstáculo al comienzo de la navegación y la planificación de la ruta es global. Pero en los entornos desconocidos, la localización del robot se basa en las lecturas de los sensores y la planificación de la ruta es individual para cada robot, dependiendo de las lecturas de los sensores.

2.2. Megaproyecto Robotat

El Megaproyecto Robotat es un proyecto que tenía como objetivo el diseño de robots con aplicaciones de seguimiento de trayectorias y comportamientos de enjambre. A partir de esta idea se realizó el diseño electrónico y mecánico de los "Bitbots". Inicialmente se enfocaron en el diseño de *hardware* de los "Bitbots" y en un algoritmo de visión por computadora para obtener la posición y orientación de los robots en un plano bidimensional. Más adelante, se empezaron a refinar detalles de *software* tales como el protocolo de comunicación y el algoritmo computacional que sería utilizado por los "Bitbots" [3]. Para el desarrollo del algoritmo de control para determinar el comportamiento de los robots se fueron desarrollando diversas tesis descritas a continuación:

2.3. Algoritmo Modificado de Optimización de Enjambre de Partículas

Aldo Aguilar [5] buscó la implementación de un algoritmo modificado de optimización de enjambre de partículas con robots diferenciales reales. En el desarrollo de esta tesis se buscaba definir un algoritmo para establecer el comportamiento de los robots para que se asemeje al de un enjambre con capacidad de búsqueda de metas. Uno de los principales problemas que se presentó era el acople directo del movimiento de las partículas del PSO debido a que el movimiento era irregular generando la posibilidad de que se genere una saturación en los actuadores del robot.

Para solucionar este problema se planteó que cada robot no iría directamente a la posición calculada por el PSO, sino que se utilizaría la posición calculada por el PSO como una sugerencia del punto al cual debía desplazarse el robot. Durante este proceso se realizaron pruebas con ocho controladores diferentes, los cuales son: Transformación de uniciclo (TUC), Transformación de uniciclo con PID (TUCPID), Controlador simple de pose (SPC), Controlador de pose Lyapunov-estable (LSPC), Controlador de direccionamiento de lazo cerrado (CLSC), Transformación de uniciclo con LQR (TUC-LQR), y Transformación de uniciclo con LQI (TUC-LQI). Al finalizar las pruebas con estos ocho controladores se determinó que los mejores resultados en cuanto a la generación de trayectorias y velocidades continuas reguladas para los robots diferenciales era el controlador TUC-LQI y el TUC-LQR.

2.4. Algoritmo de optimización de enjambre y Artificial Potential Fields

Juan Cahueque [6] realizó una implementación de la teoría de enjambre en el área de búsqueda y rescate de personas. Sabiendo que el movimiento de partículas bidimensional se realiza en una superficie tridimensional, siendo la tercera componente el resultado de la función de costo que nos indica cual es el mínimo global, podemos determinar hacia qué punto convergerán las partículas.

Juan Cahueque definió funciones artificiales de potencia de Choset y Kim, Wang y Shin

con comportamiento multiplicativo y aditivo. Lo cual nos brinda la posibilidad de modelar una función de costo "personalizada" para establecer a que punto convergerán las partículas. Se realizaron tres modelos, cada uno contaba con una meta y diversos obstáculos, los cuales tena que evitar el robot para llegar a la meta.

Para lograr que estos modelos fueran implementables en robots diferenciales físicos, se emplearon los controladores propuestos por Aldo Aguilar [5] con algunas modificaciones para hacerlos compatibles con esta nueva metodología. Realizando diversas simulaciones se llegó a determinar los mejores parámetros para los controladores en los robots diferenciales físicos. Cabe destacar que se tuvieron que realizar diversas pruebas con Webots, el cual es un *software* enfocado en simulaciones de robótica con restricciones físicas. Estas pruebas mostraron que los parámetros establecidos en Matlab eran idealizados y al ser implementados en Webots no brindaban el comportamiento esperado para el comportamiento del robot.

2.5. Algoritmo PSO modificado con aprendizaje reforzado y aprendizaje profundo

Eduardo Santizo [2] empleó la metodología de aprendizaje reforzado y aprendizaje profundo para desarrollar un *PSO tuner* y un generador de trayectorias. Para el desarrollo del *PSO tuner*, se empleó una red neuronal recurrente que toma diversas métricas propias de las partículas PSO y realiza una predicción de los valores óptimos para los parámetros relevantes del algoritmo. Entrenando las redes neuronales con 7,700 simulaciones de un algoritmo estándar de PSO se generan predicciones de carácter dinámico, logrando que el algoritmo final redujera el tiempo de convergencia y susceptibilidad a mínimos locales del PSO original.

Para el generador de trayectorias se utilizó aprendizaje profundo. En las pruebas iniciales se utilizó una cuadrícula simulando el *gridworld* de Webots en el cual el robot solo tenia la posibilidad de moverse en 4 direcciones: arriba, abajo, izquierda o derecha. Al robot se le daban penalizaciones si colisionaba contra un obstáculo y se le daba un premio si era capaz de esquivar el obstáculo. Si el robot llegaba a la meta rápidamente recibía un premio aún mayor. Usando este sistema de premios y penalizaciones, se entrenó al algoritmo para predecir de mejor manera la trayectoria. Luego se modificó el algoritmo para permitir el movimiento en diagonales a 45°, Por lo que se tienen ocho posibles movimientos y se siguió la misma metodología anteriormente descrita. Cabe destacar que la propuesta de Eduardo Santizo se enfocaba directamente en los algoritmos de aprendizaje reforzado y aprendizaje profundo por lo cual no se llegaron a realizar simulaciones con restricciones físicas.

CAPÍTULO 3

Justificación

El algoritmo de PSO nos permite optimizar un problema (función de costo) a partir de una población de soluciones candidatas o "partículas" que se mueven por todo el espacio de búsqueda según diversas reglas matemáticas que tienen en cuenta la posición y la velocidad en ese instante de las partículas. El movimiento de cada partícula se ve influido por la mejor posición que ha encontrado hasta el momento, así como por las mejores posiciones globales encontradas por las demás partículas a medida que recorren el espacio de búsqueda.

En la fase previa, se desarrolló un algoritmo que se "adapta" a la función de costo para obtener los parámetros óptimos y brindar un mejor comportamiento de las partículas para lograr una convergencia rápida y efectiva hacia el mínimo o máximo del a función. Sin embargo, no se han realizado pruebas en entornos físicos tomando en consideración los controladores a utilizar y la validación del comportamiento esperado por parte de los robots con condiciones físicas.

Debido a esto, este proyecto buscó la integración de simuladores de condiciones físicas junto con el algoritmo adaptado con aprendizaje profundo y aprendizaje reforzado para la optimización de funciones de costo para la obtención de los parámetros óptimos en un entorno estático. Se verificaron y validaron los resultados obtenidos en entornos físicos simulados con restricciones físicas, con los resultados obtenidos a partir de simulaciones ideales del comportamiento de los robots. Esto estableció las bases para futuras experimentaciones que resuelvan cada vez problemas más complejos brindando resultados satisfactorios.

Se evaluó la viabilidad de la simulación de dicho algoritmo en el entorno de ROS debido a que es un *software* popular en el área de robótica por sus buenas simulaciones de condiciones físicas como lo son la fricción, la interacción de las fuerzas, etc. Además, cuenta con diversos modelos de robots, actuadores y sensores los cuales son actualizados por las mismas empresas o algunos diseñadores para brindar un simulador muy completo.

Por lo tanto, se puede considerar el entorno de ROS como un *software* de gran utilidad para la evaluación del comportamiento de robots en condiciones físicas y definir controladores apropiados que nos brinden el comportamiento esperado.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Integrar las herramientas de *software PSO Tuner* y *Swarm Robotics Toolbox*, desarrolladas en la fase anterior, a plataformas de simulación con restricciones físicas.

4.2. Objetivos específicos

- Evaluar entornos de simulación con motores de física que permitan simulaciones de robótica de enjambre más realistas.
- Realizar simulaciones en el entorno seleccionado y obtener datos de entrenamiento para el *PSO Tuner* y *Swarm Robotics Toolbox*.
- Utilizar los parámetros obtenidos por el *PSO Tuner* y el *Swarm Robotics Toolbox*, luego del entrenamiento, en nuevos escenarios, y verificar el rendimiento del sistema.
- Evaluar la viabilidad del uso de ROS para aplicaciones de robótica de enjambre en combinación con las herramientas *PSO Tuner* y *Swarm Robotics Toolbox*.

CAPÍTULO 5

Alcance

Este proyecto tiene como propósito implementar el algoritmo PSO para robots diferenciales en entornos con restricciones físicas. Se busca lograr que el sistema sea capaz de evadir obstáculos en entornos desconocidos siendo capaz de lograr converger al punto mínimo del sistema planteado, tomando en consideración restricciones físicas tales como la gravedad, las dimensiones del sistema, etc.

Por otro lado, se busca la implementación del algoritmo dentro del entorno de *Robotic Operating System* (ROS) debido a que es un entorno de simulación creado para simular condiciones físicas que se asemejen a una implementación real dentro de cualquier entorno desconocido. El algoritmo debe de ser capaz de simular comportamientos similares a una implementación física real.

El motivo de esta investigación es lograr determinar un correcto funcionamiento del sistema en condiciones no ideales. El sistema debe se ser capaz de lograr converger al mínimo global del sistema. Por otro lado, se desea lograr crear ejemplos de implementaciones dentro del sistema de ROS para futuras implementaciones dentro del sistema en cursos universitarios.

Estas simulaciones buscan un sistema capaz de converger al mínimo global tomando en consideración que existen obstáculos dentro del entorno. Deben ser capaces de evadir dichos obstáculos de forma segura evitando cualquier tipo de contacto entre el robot diferencial y el obstáculo colocado aleatoriamente dentro del entorno.

CAPÍTULO 6

Marco teórico

6.1. Optimización por Enjambres de Partículas (PSO)

La Optimización por Enjambres de Partículas (conocida como PSO, por sus siglas en inglés, Particle Swarm Optimization) es una técnica de optimización/búsqueda. Este método fue descrito alrededor de 1995 por James Kennedy y Russell C. Eberhart [1], y se inspira en el comportamiento de los enjambres de insectos en la naturaleza. Formalmente hablando, se supone que tenemos una función desconocida, $f(x, y)$, que podemos evaluar en los puntos que queramos, pero a modo de caja negra, por lo que no podemos conocer su expresión. El PSO es fácil de implementar y ha sido ampliamente usado en variadas aplicaciones con resultados excelentes resolviendo problemas reales de optimización. Este algoritmo puede ser computacionalmente ineficiente por que puede quedar atrapado fácilmente en óptimos locales cuando resuelve problemas cuyo espacio de solución es multimodal; estas debilidades han hecho que el campo de aplicación de la metodología esté un poco restringido [7].

El objetivo es el habitual en optimización, encontrar valores de x y y para los que la función $f(x, y)$ sea máxima (o mínima, o bien verifica alguna relación extremal respecto a alguna otra función). La idea que vamos a seguir en PSO comienza de forma similar, situando partículas al azar en el espacio de búsqueda, pero dándoles la posibilidad de que se muevan a través de él de acuerdo con unas reglas que tienen en cuenta el conocimiento personal de cada partícula y el conocimiento global del enjambre [7].

Cada partícula (individuo) tiene una posición, p (que en 2 dimensiones vendrá determinado por un vector de la forma $[x, y]$), en el espacio de búsqueda y una velocidad, v (que en 2 dimensiones vendrá determinado por un vector de la forma $[v_x, v_y]$), que determina su movimiento a través del espacio. Además, como partículas de un mundo real físico, tienen una cantidad de inercia, que los mantiene en la misma dirección en la que se movían, así como una aceleración (cambio de velocidad), que depende principalmente de dos características:

- Cada partícula es atraída hacia la mejor localización que ella, personalmente, ha encontrado en su historia (mejor personal).

- Cada partícula es atraída hacia la mejor localización que ha sido encontrada por el conjunto de partículas en el espacio de búsqueda (mejor global).

Formalmente, lo podemos escribir como:

$$v_i(t+1) = v_i(t) + c_1 r_1(p_i^{mejor} - p_i(t)) + c_2 r_2(p_g^{mejor} - p_i(t)) \quad (1)$$

En donde:

- c_1, c_2 = son las constantes de atracción al mejor personal y el mejor global
- r_1, r_2 = son números aleatorios entre 0 y 1
- p_i^{mejor} = es la mejor posición por la cual ha pasado una partícula
- p_g^{mejor} = es la mejor posición global de todo el sistema de enjambre

Una vez actualizadas las velocidades de todas las partículas, sus posiciones se actualizan siguiendo una ley simple:

$$p_i(t+1) = v_i(t) + p_i(t) \quad (2)$$

un conjunto de partículas representa potenciales soluciones, donde cada partícula i está asociada a dos vectores, el vector de velocidades y a la posición del vector. La velocidad y la posición de cada partícula son inicializadas por vectores aleatorios con sus correspondientes rangos [8].

6.2. Aprendizaje profundo

En el enfoque Aprendizaje Profundo se usan estructuras lógicas que se asemejan en mayor medida a la organización del sistema nervioso de los mamíferos, teniendo capas de unidades de proceso (neuronas artificiales) que se especializan en detectar determinadas características existentes en los objetos percibidos. La visión artificial es una de las áreas donde el Aprendizaje Profundo proporciona una mejora considerable en comparación con algoritmos más tradicionales. Existen varios entornos y bibliotecas de código de Aprendizaje Profundo que se ejecutan en las potentes GPUs modernas tipo CUDA, como por ejemplo NVIDIA cuDNN.

El Aprendizaje Profundo representa un acercamiento más íntimo al modo de funcionamiento del sistema nervioso humano. Los modelos computacionales de Aprendizaje Profundo imitan estas características arquitecturales del sistema nervioso, permitiendo que dentro del sistema global haya redes de unidades de proceso que se especialicen en la detección de determinadas características ocultas en los datos. El Aprendizaje Profundo lleva a cabo el proceso de *Machine Learning* usando una red neuronal artificial que se compone de un número de niveles jerárquicos. En el nivel inicial de la jerarquía la red aprende algo simple y luego envía esta información al siguiente nivel. El siguiente nivel toma esta información sencilla, la combina, compone una información algo un poco más compleja, y se lo pasa al

tercer nivel, y así sucesivamente. Destaca porque no requiere de reglas programadas previamente, sino que el propio sistema es capaz de "aprender" por sí mismo para efectuar una tarea a través de una fase previa de entrenamiento [9].

Los algoritmos que componen un sistema de aprendizaje profundo se encuentra en diferentes capas neuronales compuestas por pesos (números). El sistema está dividido principalmente en tres capas:

- **Capa de entrada (Input Layer):** Está compuesto por las neuronas que asimilan los datos de entrada, como por ejemplo imagen o una tabla de datos.
- **Capa oculta (Hidden Layer):** Es la red que realiza el procesamiento de información y hacen los cálculos intermedios. Cada más neuronas en esta capa haya, más complejos son los cálculos que se efectúan.
- **Salida (Output Layer):** Es el último eslabón de la cadena, y es la red que toma la decisión o realiza alguna conclusión aportando datos de salida.

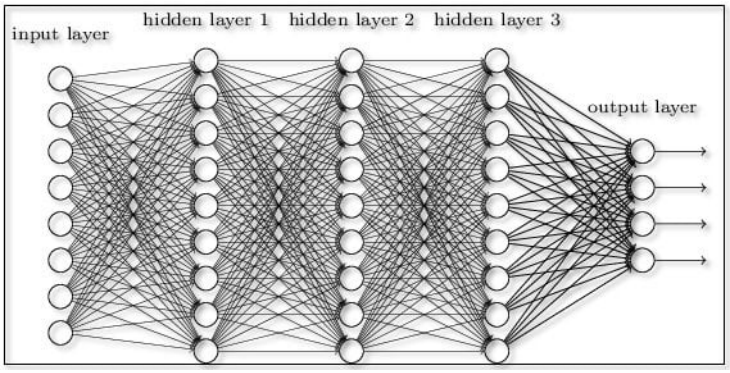


Figura 1: Arquitectura de las redes neuronales [10].

6.3. Aprendizaje reforzado

El Aprendizaje por refuerzo es un área del aprendizaje automático inspirada en la psicología conductista, cuya ocupación es determinar qué acciones debe escoger un agente de software en un entorno dado con el fin de maximizar alguna noción de "recompensa" o premio acumulado. Intenta conseguir que una inteligencia artificial aprenda a decidir mediante su propia experiencia. Es decir, que ante una situación determinada, sea capaz de seleccionar por sí misma la mejor acción a ejecutar en ese momento mediante un proceso interactivo de prueba y error a base de reforzar positivamente cada vez que se aproxima o logra objetivo. Por eso, con el Aprendizaje reforzado una máquina puede tomar decisiones aunque no almacene un conocimiento a priori del entorno o de las variables que se están dando, y realizar de manera satisfactoria cuestiones abstractas más avanzadas. [11]

La aplicación de ese aprendizaje les permite ya reconocer caras, clasificar secuencias de ADN, conducir vehículos, o hacer diagnósticos médicos. En la actualidad compañías tecnológicas punteras como Google, Apple o IBM, están invirtiendo en investigación para entrenar

robots que realicen sencillas tareas mediante esta técnica. El aprendizaje por refuerzo es especialmente adecuado para los problemas que incluyen un razonamiento a largo plazo frente a uno a corto plazo. Se ha aplicado con éxito a diversos problemas, entre ellos el control de robots, telecomunicaciones, backgammon y damas. [11]

Dos componentes hacen aprendizaje por refuerzo de gran alcance: el uso de muestras para optimizar el rendimiento y el uso de la función de aproximación para hacer frente a entornos de gran tamaño. El problema de aprendizaje reforzado requiere mecanismos de exploración inteligente. Seleccionar al azar acciones, sin hacer referencia a una distribución de probabilidad estimada, que se conoce para dar lugar a un rendimiento muy pobre. [11]

6.3.1. Formulación del aprendizaje reforzado

El Aprendizaje Reforzado se plantea como una herramienta de resolución de problemas de toma de decisiones secuenciales, del cual se supone clásicamente que se rige mediante un proceso de decisión de Markov. En este caso, el proceso queda conformado por los elementos presentes en la Figura 2, y puede describirse mediante la tupla MDP $= (S, A, R, P)$ [12]. Adicionalmente, la propiedad de Markov en este caso queda descrita como se presenta en la ecuación 3.

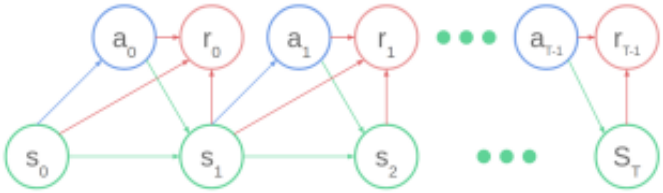


Figura 2: Proceso de decisión de Markov [12].

$$\forall t \qquad P(s_t \mid s_{t-1}, a_{t-1}) = P(s_t \mid s_{t-1}, \ldots, s_0, a_{t-1}, \ldots, a_0) \tag{3}$$

De la Figura 2, S y A representan al espacio de estados y acciones respectivamente. La formulación general no tiene restricciones con respecto a la naturaleza de dichos espacios, pero los trabajos clásicos y las principales garantías de convergencia existen solo para el caso de espacios discretos, en donde no existe necesidad de utilizar aproximadores funcionales. Por otro lado, R representa a la recompensa escalar, pilar fundamental del Aprendizaje Reforzado. Existen diferentes definiciones de la misma, ya sea $R : S \to R$, $R : S \times A \to R$ o incluso $R : S \times A \times S \to R$. Finalmente P representa a la función de transición, que usualmente es considerada estocástica, es decir: $P : S \times A \times S \to [0, 1]$. Una de las ventajas del Aprendizaje Reforzado con respecto a otras metodologías como el control óptimo, es que precisamente no se asume P como conocida (y muchos casos nunca se aprende directamente) [12].

Finalmente, el producto del proceso de aprendizaje, es además el objeto matemático que directa o indirectamente se modifica durante el entrenamiento corresponde a la política π. La política corresponde a un mapeo entre el espacio de estados y el espacio de acciones, el

cual puede ser determinista $\pi : S \to A$ o estocástico $\pi : S \times A \to [0, 1]$. En la literatura se suele recomendar la utilización de políticas estocásticas en aquellos problemas en los que se presente observabilidad parcial. Sin embargo estudios recientes, algoritmos que aprenden políticas determinísticas han sido capaces de abordar problemas complejos con observabilidad parcial [13].

6.4. Algoritmo D* Lite para evasión de obstáculos

El algoritmo D* Lite es un algoritmo de búsqueda que resuelve los mismos problemas de planificación de ruta basada en la suposición, incluyendo la planificación con la suposición de espacio libre, donde un robot tiene que navegar hasta las coordenadas de un objetivo dado en un terreno desconocido. Hace suposiciones sobre la parte desconocida del terreno y encuentra un camino más corto desde sus coordenadas actuales hasta la meta bajo estas suposiciones. El robot entonces sigue el camino. Cuando se observa nueva información del mapa, se añade la información a su mapa y, si es necesario, planea el nuevo camino más corto a partir de sus coordenadas actuales a las coordenadas del objetivo determinado. Se repite el proceso hasta que llega a las coordenadas del objetivo o determina que no se puede llegar a las coordenadas del objetivo [14].

El algoritmo se basa en LPA* (*Lifelong Planning A**) el cual es una versión incremental del algoritmo A*. Este algoritmo se aplica a un espacio finito problemas de búsqueda en espacios conocidos cuyos costos aumentan o disminuyen con el tiempo. S denota el numero finito de vértices en el espacio. $Succ(s) \subseteq S$ denota el numero de sucesores del vértice $s \in S$ y $Pred(s) \subseteq S$ denota el numero de predecesores del vértice $s \in S$. El algoritmo LPA* siempre determina el camino mas corto desde un vértice inicial $s_{inicio} \in S$ a el vértice final $s_{meta} \in S$. La función $g(s)$ denota la distancia mas corta entre el vértice inicial y el vértice final. Al igual que A*, el algoritmo LPA* utiliza la heurística para aproximar la distancia desde cada vértice hasta el vértice final [14].

El algoritmo LPA* mantiene una estimación $g(s)$ de la distancia inicial y una estimación $g(s)$ para cada vértice dentro del espacio. También mantiene un segundo tipo de valor estimado de la distancias iniciales. Estos valores corresponden a $rhs(s)$ el cual es un valor adelante del valor de $g(s)$ y corresponde a lo siguiente:

$$rhs(s) = \begin{cases} 0, & \text{if } s = s_{inicio} \\ min_{s' \in Pred(s)}(g(s') + c(s', s)), & \text{otherwise} \end{cases} \tag{4}$$

Si el valor de $rhs(s)$ es igual al valor de $g(s)$ entonces el vértice es localmente consistente, sin embargo, si el valor de $rhs(s)$ es diferente de $g(s)$ el vértice es localmente inconsistente. Si todos los vértices son localmente consistentes, entonces los valores de $g(s)$ son iguales a sus distancias iniciales. En este caso, se puede calcular el camino mas corto desde el vértice s_{inicio} a cualquier vértice que sea predecesor (s') que minimice $g(s') + c(s', s)$ hasta que se llegue hasta el vértice final [14].

En el caso de que existan vértices que sean localmente inconsistentes, se utiliza la heurística para centrar la búsqueda y actualizar los valores de $g(s)$ que son relevantes para encontrar el camino mas corto. Para realizar esto, el algoritmo utiliza una cola de prioridad,

la cual solo contiene los vértices que son localmente inconsistentes. La prioridad de un vertice se determina utilizando un vector con dos componentes: $k(s) = [k_1(s); k_2(s)]$, en donde $k_1(s) = min(g(s), rhs(s)) + h(s, s_{meta})$ y $k_2(s) = min(g(s), rhs(s))$. El algoritmo LPA* siempre expande el vértice en la cola de prioridad que contiene los valores más pequeños [14].

Hasta ahora, hemos descrito el funcionamiento del algoritmo LPA*, el cual calcula siempre el camino más corto entre el vértice inicial y el vértice final conforme el costo de cada vértice va cambiando. Se procede a explicar el algoritmo D* lite utilizando este algoritmo como base. con el algoritmo D* lite se puede determinar el camino más corto tomando en consideración el cambio del costo de cada vértice a medida que el robot se va moviendo desde el vértice inicial hacia el vértice final. Para esto, se realizan solo dos cambios al algoritmo LPA* los cuales son:

- Se cambia la dirección de la búsqueda ya que el algoritmo LPA* busca el camino mas corto desde la posición s_{meta} hasta s_{inicio}. Debido a que se quiere actualizar los nodos tomando en consideración la posición actual del robot, es necesario invertir la forma en la cual se calcula el camino más corto generando que se calcule desde s_{inicio} hasta s_{meta} [14].
- Se introduce un modificador a la cola de prioridad para tomar en consideración cuando el robot se mueve desde s_{inicio} a s'_{inicio}. este modificador causa que al momento de agregar de nuevo los nodos a la cola de prioridad incremente el valor de los nodos mas alejados y evite tomar en consideración los nodos predecesores como vértices a los cuales se puede mover [14].

Finalmente, nuestra nueva función para la cola de prioridad quedaría de la siguiente forma:

$$k(s) = [min(g(s), rhs(s)) + h(s, s_{meta}) + k_m; min(g(s), rhs(s))] \tag{5}$$

En donde:

$$k_m = k_m + h(s, s') \tag{6}$$

6.5. Algoritmo de Dijkstra

El algoritmo de Dijkstra fue creado y publicado por el Dr. Edsger W. Dijkstra, un ingeniero en sistemas y ciencias de la computacion. El algoritmo de Dijkstra permite encontrar el camino más corto entre dos vértices de un grafo. Para describir el algoritmo partimos de la definición de un grafo, como la representación gráfica de un conjunto de nodos o vértices unidos por enlaces llamados aristas o arcos [15]. Existen dos tipos de grafos:

- **Indirectos:** Por cada par de nodos en el grafo, se puede ir de un nodo a otro en ambas direcciones.
- **Directos:** Por cada par de nodos en el grafo, solo se puede ir de un nodo a otro en una dirección especifica.

El algoritmo de Dijkstra utiliza grafos ponderados. Un grafo ponderado es un grafo cuyos vértices tienen un peso o costo. El peso o costo de estos vértices puede representar tiempo, distancia o cualquier tipo de dato que modele una conexión entre los pares de nodos [15].

Asumiendo que se tiene un grafo $G = (V, E)$ como un conjunto de vértices V donde $V = [v_0, v_1, \ldots, v_{n-1}]$ y un conjunto de aristas E que unen los vértices, a los cuales se les asocia un peso. El orden de complejidad del algoritmo de Dijkstra es de $O(n2)$ [16]. A continuación se describe el pseudocódigo del algoritmo en los siguientes pasos:

1. Definir un vector D, para guardar las distancias e inicializarlo con un valor infinito relativo, ya que las distancias son desconocidas en un principio, excepto para el vértice inicial v_0, el cual tiene una distancia de cero [16].

2. Sea α, el vértice actual. Se recorren todos los vértices adyacentes, marcando los ya visitados y los no visitados v_j [16].

3. Si la distancia desde v_0 hasta v_j guardada en el vector D, es mayor que la distancia desde v_0 hasta a sumada a la distancia desde α hasta v_j, se sustituye el valor de la distancia D_j por el valor de distancia $D_\alpha + d(\alpha, v_j)$ es decir: Si $(D_j > D_\alpha + d(\alpha, v_j))$ entonces $(D_j = D_\alpha + d(\alpha, v_j))$ [16].

4. Se marca el vértice a como vértice ya visitado y se toma como próximo vértice a visitar al de menor valor en el vector D, repitiendo el tercer paso mientras existan vértices no marcados [16].

6.6. Algoritmo de Braitenberg

El Vehículo de Braitenberg tiene dos ruedas actuadas por motores independientes, y varios sensores de luz o distancia y que a mayor intensidad de luz o menor distancia, mayor par aplicado a las ruedas, y en consecuencia se genera un movimiento de las ruedas a mayor velocidad angular. Este tipo de vehículos se guían por instintos, entendiendo que un instinto se corresponde con un sistema de valores de intensidades variables, es decir un sistema de motivaciones y gustos [17].

Suponiendo un robot que funciona con sensores de luz, este puede realizar un movimiento circular alrededor de una fuente de luz externa si la función de activación tiene un máximo en algún punto, es decir si funciona con una polaridad positiva para valores de intensidad de la luz bajas y con polaridad negativa para intensidades elevadas. Si la conexión es contra lateral, se acercará a la fuente de luz cuando esté lejos de ella, y se alejará de ella cuando esté más próximo [17].

Si el anterior robot dispusiese además de otros sensores con funciones de activación diversas, el conjunto de comportamientos que podrían realizar sería muy variado, hasta el punto de que resultaría difícil, si no imposible, para un observador externo deducir esta variedad de comportamientos a partir del análisis de los elementos del robot. La dificultad de segmentar los comportamientos mediante técnicas analíticas es una de las características principales de los vehículos de Braitenberg [17].

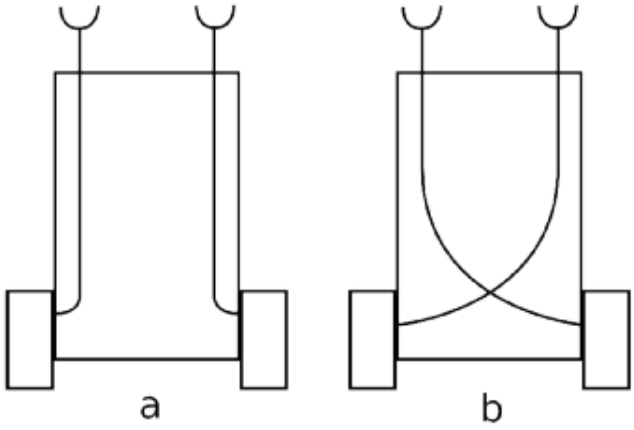


Figura 3: Diagrama de braitenberg [17].

6.7. Controlador PID

Un control PID se basa en un modelo matemático riguroso de algún proceso lineal o lineal dentro de un rango. Estos modelos se desarrollan ya sea por Root-Locus, ubicación de polos, o algún otro método, y si la planta se puede representar por un sistema de primer orden con retardo. El diseño del controlador puede ser realizado de manera empírica (sin conocimiento del modelo), mediante el método de Ziegler-Nichols o algún derivado de este, y se pueden asignar los coeficientes proporcional, integral y derivativo del controlador del sistema [18].

Un control PID lee un valor de entrada o del sensor de entrada, aplica los algoritmos de control, definidos y produce una salida específica como señal actuante o como entrada a un actuador. La salida del sistema es medida por el sensor de entrada, y el proceso se repite indefinidamente.El controlador PID clásico tiene la forma:

$$u = K_p \left(e + \frac{1}{T_i} \int_{x=0}^{t} e dt + T_d \frac{de}{dt} \right) \tag{7}$$

En donde "e" es el error entre el valor de referencia y la salida del sistema,"u" es la salida del controlador, "K_p" la ganancia proporcional, "T_i" el tiempo integral, y "T_d" el tiempo derivativo, existiendo diversos métodos para ajustar los tres parámetros de interés, siendo el de Ziegler-Nichols el origen de todos ellos [18].

En este tipo de controlador, la ganancia K_p que afecta el componente proporcional, reduce el tiempo de crecimiento y elimina "parte" del error estacionario, "$K_i = \frac{1}{Ti}$" que afecta el componente integral, elimina el error en estado estacionario pero puede tener como efecto secundario una afectación de la respuesta transitoria, y "$K_d = T_d$" que afecta el componente derivativo, reduce el sobre pico y mejora la respuesta transitoria, esto se puede observar en el Cuadro 1.

	T_r	M_p	T_s	e_{ee}
K_p	Disminuye	Aumenta	Poco Efecto	Disminuye
K_i	Disminuye	Aumenta	Aumenta	Elimina
K_d	Poco Efecto	Disminuye	Disminuye	Poco Efecto

Cuadro 1: Efecto de las constantes en la salida del sistema [18].

En controladores digitales, la ecuación del controlador PID puede ser reemplazada por:

$$u = K_p \left(e(k) + K_i \sum_{j=1}^{n} e(j) T_s + K_d \frac{e(k) - e(k-1)}{T_s} \right) \tag{8}$$

Cambiando la derivada por una diferencia hacia atrás, la integral por una suma, y un tiempo de muestreo pequeño, siendo "u" el instante de tiempo.

6.8. Regulador cuadrático lineal

6.8.1. Espacio de estados

En ingeniería de control, el control clásico se analiza por conveniencia en el dominio de la frecuencia, y está limitado a sistemas lineales con una única entrada y una única salida. La representación de espacios de estado proporciona un modo compacto y conveniente de modelar y analizar sistemas con múltiples entradas y salidas, tanto para sistemas lineales como no lineales [19].

Una representación de espacios de estados es un modelo matemático de un sistema físico descrito mediante un conjunto de entradas, salidas y variables de estado relacionadas por ecuaciones diferenciales de cualquier orden en el dominio del tiempo, que se combinan en una ecuación diferencial matricial de primer orden. Las variables de entradas, salidas y estados son convenientemente expresadas como vectores: un vector de entrada, un vector de salida y un vector de estados; y si el sistema dinámico es lineal e invariante en el tiempo, las ecuaciones algebraicas se escriben en forma matricial [19].

En el caso de sistemas lineales continuos se puede suponer que el comportamiento dinámico de un sistema se puede expresar como una ecuación diferencial de orden n que establece la relación entre la entrada u y la salida del sistema:

$$y^{(n)}(t) + a_{n-1} y^{(n-1)}(t) + \cdots + a_1 y^{(1)}(t) + a_0 y(t) = b_0 u(t) + \cdots + b_m u^{(m)}(t) \tag{9}$$

Las técnicas de la representación en el espacio de estado o representación interna se basan en la propiedad que cumplen los sistemas y procesos según la cual siempre es posible utilizar un sistema de ecuaciones diferenciales de primer orden para representar un sistema de orden mayor. De esta forma, en vez de tener una ecuación diferencial de $n-$ésimo orden, se tendrán n ecuaciones diferenciales de primer orden [19]. Cada una de estas n ecuaciones

tendrá la forma general siguiente:

$$\frac{dx_i(t)}{dt} = \sum_{j=1}^{n} \alpha_{ij} x_j(t) + \beta_i u(t) \tag{10}$$

con:

$$y(t) = \sum_{j=1}^{n} c_j x_j(t) \tag{11}$$

Estas n ecuaciones diferenciales se pueden expresar con una única ecuación matricial:

$$\begin{bmatrix} \dot{x}_1(t) \\ \dot{x}_2(t) \\ \vdots \\ \dot{x}_n(t) \end{bmatrix} = \begin{bmatrix} \alpha_{11} & \alpha_{12} & \alpha_{13} & \cdots & \alpha_{1n} \\ \alpha_{21} & \alpha_{22} & \alpha_{23} & \cdots & \alpha_{2n} \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \alpha_{n1} & \alpha_{n2} & \alpha_{n3} & \cdots & \alpha_{nn} \end{bmatrix} * \begin{bmatrix} x_1(t) \\ x_2(t) \\ \vdots \\ x_n(t) \end{bmatrix} + \begin{bmatrix} \beta_1(t) \\ \beta_2(t) \\ \vdots \\ \beta_n(t) \end{bmatrix} u(t) \tag{12}$$

A la ecuación anterior se le suele definir como ecuación de estado del sistema la cual fija la evolución del estado en función del valor actual de dicho estado y de la aportación de la entrada [19]. Esta ecuación se puede presentar en forma matricial y compacta, obteniendo finalmente las siguientes expresiones:

$$\dot{x}(t) = Ax(t) + Bu(t) \tag{13}$$

$$y(t) = Cx(t) + Du(t) \tag{14}$$

En donde:

- A = matriz $(n \times n)$ de estado
- B = matriz $(n \times 1)$ de entrada
- C = matriz $(1 \times n)$ de salida
- D = matriz $(n \times 1)$ de acoplamiento (usualmente es un matriz nula)

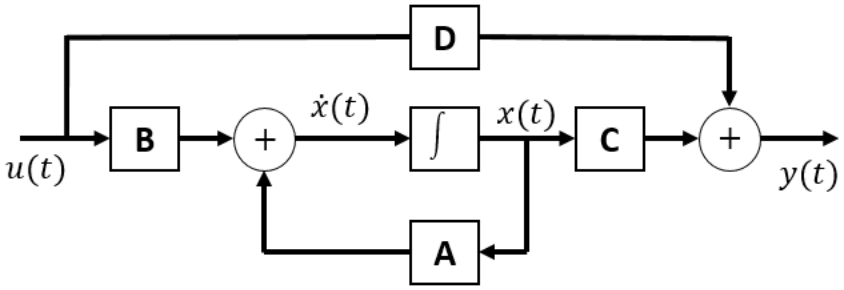


Figura 4: Modelo del espacio de estados [20]

6.8.2. Regulador cuadrático lineal (LQR)

El regulador cuadrático lineal es un controlador por retroalimentación que busca llevar al sistema a un estado determinado de forma rápida y reduciendo la cantidad de control utilizado. El controlador LQR debe utilizar un proceso de optimización que determine el desempeño del mismo [21]. la función que se utiliza es el índice de desempeño dada por:

$$J(t_0) = \frac{1}{2}x^T(T)S(T)x(T) + \frac{1}{2}\int_{t_0}^{T}(x^TQx + u^TRu)dt \tag{15}$$

En el controlador LQR, las matrices de peso Q y R se convierten en los parámetros de diseño. La matriz Q es la matriz de peso para los estados intermedios, la matriz R es la matriz de peso para la acción de control del sistema y la matriz $S(T)$ representa el peso del estado final $x(T)$ [21]. La dinámica de la planta es:

$$\dot{x}(t) = Ax(t) + Bu(t) \qquad t \geq t_0 \tag{16}$$

$$u(t) = -Kx(t) \tag{17}$$

Para lograr el objetivo de control se necesita determinar una ley de control $u(t)$ que minimice $J(t_0)$. Si se considera que la ecuación diferencial de Riccati tiene una solución en estado estacionario, el problema ahora se convierte en un problema de horizonte en infinito en donde la ecuación diferencial de Riccati (DRE) se torna en una ecuación algebraica de Riccati (ARE) de la forma:

$$-\dot{S} = A^TS + SA - SBR^{-1}B^TS + Q \qquad \Rightarrow \qquad 0 = A^TS + SA - SBR^{-1}B^TS + Q \tag{18}$$

Con índice de desempeño:

$$J = \int_{t_0}^{\infty}(x^TQx + u^TRu)dt \tag{19}$$

En la ecuación anterior ya no es necesario pesar el estado final ya que, cuando el sistema es asintóticamente estable, $x(t) \to 0$ según $t \to \infty$ [21]. Resolviendo la ecuación 18 se obtiene el valor de la ganancia de Kalman mediante:

$$K_\infty = R^{-1}B^TS_\infty \tag{20}$$

finalmente, la acción de control para el controlador LQR con horizonte en infinito en sistema a lazo cerrado es:

$$\dot{x}(t) = (A - BK_\infty)x(t) \tag{21}$$

6.9. Integrador cuadrático lineal (LQI)

Básicamente es similar al LQR, sin embargo, en este controlador, la ley de control se expresa en la ecuación 22 donde x_i es la salida del integrador. Esta ley de control asegura que la salida "y" siga el comando de referencia o punto de consigna, el número de integradores es igual a la dimensión de la salida "y" [22].

$$u = -K[x; x_i] \tag{22}$$

Esta ley de control adicionalmente minimiza las funciones de coste que se muestran en la siguiente ecuación:

$$J(u) = \sum_{n=0}^{\infty} \left\{ z^T Q_z + u^T R_u + 2 z^T N_u \right\} \tag{23}$$

En tiempo discreto, el algoritmo LQI calcula la salida del integrador x_i usando la fórmula de avance de Euler como se indica en la ecuación 24, en donde T_s es el tiempo de muestreo del sistema [22].

$$x_i[n+1] = x_i[n] + T_s(r[n] - y[n]) \tag{24}$$

Tomando esta disctetizacion del algoritmo se procede a encontrar los valores de las matrices del sistema LTI con la ecuación 25 y se plantea el error del sistema calculando la diferencia entre la referencia y la salida del sistema como se observa en la ecuación 26.

$$\begin{bmatrix} \dot{x(t)} \\ \dot{x_i(t)} \end{bmatrix} = \begin{bmatrix} A_{n\times n} & O_{n\times p} \\ -C_{p\times n} & O_{p\times p} \end{bmatrix} \begin{bmatrix} x(t) \\ x_i(t) \end{bmatrix} + \begin{bmatrix} B_{n\times n} \\ O_{p\times n} \end{bmatrix} u(t) + \begin{bmatrix} O_{n\times p} \\ I_{p\times p} \end{bmatrix} r(t) \tag{25}$$

$$e(t) = r(t) - y(t) = r(t) - Cx(t) \tag{26}$$

Finalmente, se puede utilizar el mismo cálculo del regulador lineal cuadrático (LQR) para obtener el valor de la matriz K tomando los nuevos valores de las matrices A y B del sistema. Obteniendo como resultado la siguiente ecuación:

$$u = -Kx = \begin{bmatrix} K_{m\times n} \\ K_{Im\times p} \end{bmatrix} \begin{bmatrix} x_{n\times 1} \\ x_{Ip\times 1} \end{bmatrix} = -Kx - K_i x_i \tag{27}$$

6.10. Robot diferencial E-Puck

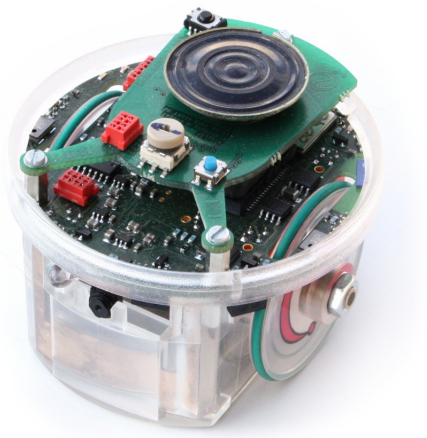


Figura 5: robot e-puck[23].

El E-Puck es un robot diferencial móvil desarrollado por la Escuela Politécnica Federal de Lausana (EPFL). Este robot está incluido en la librería de Webots para realizar simulaciones. Este robot cuenta con una gran variedad de sensores que lo hacen una buena opción para su implementación con el algoritmo PSO. Es un robot pequeño y su programación es amigable con el usuario, por lo cual ha sido utilizado como herramienta de aprendizaje [24].

El e-puck cuenta con las siguientes características:

Características	**Valor**
Diámetro	70 mm
Altura	55 mm
Radio de la rueda	20.5 mm
Largo del eje	52 mm
Peso	0.15 Kg
Velocidad máxima de rotación	0.25 m/s
Velocidad máxima de giro	6.28 rad/s

Cuadro 2: Características físicas del robot e-puck [25].

6.11. Modelado de robots diferenciales

Uno de los sistemas más usuales se basa en el uso de ruedas de tracción, el cual es un sistema poco complejo y adecuado para la navegación en algunos entornos de desarrollo típicos de actividades humanas, por ejemplo oficinas, bodegas y otros. Las configuraciones de tracción diferencial, son muy populares y permiten calcular la posición del robot a partir de las ecuaciones geométricas, que surgen de la relación entre los componentes del sistema de propulsión y de la información de los codificadores rotativos que usualmente llevan acoplados a sus ruedas.

Típicamente una plataforma móvil de tracción diferencial cuenta con dos pares de ruedas: dos ruedas de tracción que tienen acoplados dos motores DC, dos ruedas de estabilización que mantienen el balance del vehículo. La traslación y rotación de este tipo de plataformas diferenciales se determinan por el movimiento independiente de cada una de las ruedas de tracción [26].

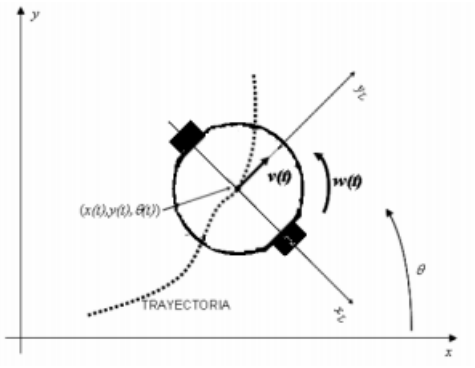


Figura 6: Localización del robot en el plano cartesiano [26]

Las ecuaciones cinemáticas del punto medio del eje entre las ruedas activas, referidas a un sistema de referencia global como se muestra en la Figura 6 son :

$$\dot{x} = v(t)cos(\theta(t)) \tag{28}$$

$$\dot{y} = v(t)sen(\theta(t)) \tag{29}$$

$$\dot{\theta} = \omega(t) \tag{30}$$

La posición y orientación del robot móvil se obtiene integrando las velocidades de este en un periodo de tiempo Δt como se observa en las siguientes ecuaciones [26]:

$$x(t) = x(t_0) + \int_{\Delta t} v(t)cos(\theta(t))dt \tag{31}$$

$$y(t) = y(t_0) + \int_{\Delta t} v(t)sen(\theta(t))dt \tag{32}$$

$$x(t) = \theta(t_0) + \int_{\Delta t} \omega(t)dt \tag{33}$$

Si el periodo de observación Δt, tiende a 0, entonces las integrales anteriores pueden ser remplazadas por los desplazamientos diferenciales Δx, Δy, $\Delta \theta$. Si se mantiene una frecuencia de muestreo constante y elevada sobre la odometría de un móvil, se puede estimar la posición y orientación del mismo mediante las ecuaciones de diferencias (34), (35) y (36). Además se puede considerar que la velocidad angular en cada una de las ruedas se mantiene constante durante el periodo de muestreo [26].

$$x_k = x_{k-1} + \Delta x_k \tag{34}$$

$$y_k = y_{k-1} + \Delta y_k \tag{35}$$

$$\theta_k = \theta_{k-1} + \Delta \theta_k \tag{36}$$

6.12. Funciones de costo

En matemáticas aplicadas, las funciones de prueba, conocidas como paisajes artificiales, son útiles para evaluar características de algoritmos de optimización, tales como:

- Tasa de convergencia
- Precisión
- Robustez
- Rendimiento general

Las siguientes funciones de prueba tienen el objetivo de dar una idea de las diferentes situaciones a las que se enfrentan los algoritmos de optimización a la hora de afrontar este tipo de problemas y verificar que son capaces de converger al mínimo global de la función de costo.

6.12.1. Función de esfera

$$f(x) = \sum_{i=1}^{n} x_i^2 \qquad f(x_1, \ldots, x_n) = 0 \tag{37}$$

La función de esfera se representa en la ecuación (37), es adecuada para la optimización de objetivo único, lo que significa que presenta una función de objetivo único. Además, la función de la esfera es unimodal. Esto significa que presenta un "modo" y tiene un único óptimo global [27]. Esta función se resuelve con los métodos clásicos de optimización mucho más rápido y se puede observar en la Figura 7.

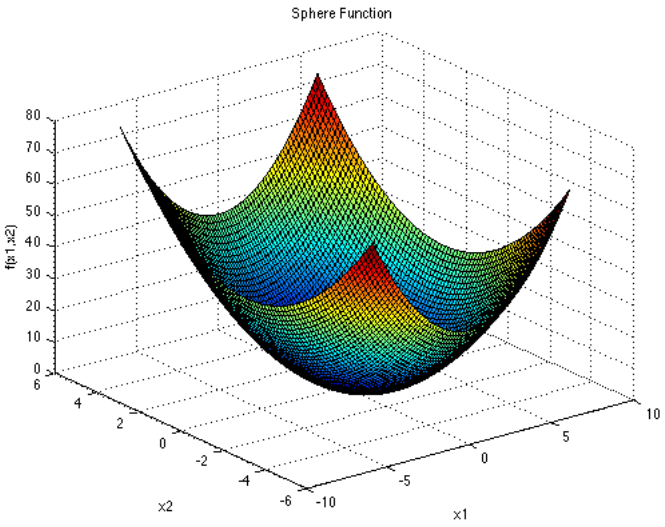


Figura 7: Comportamiento de la función de esfera [27].

6.12.2. Función de Rosenbrock

$$f(x) = \sum_{i=1}^{n-1} \left[100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2\right] \qquad Min = \begin{cases} n = 2, & \text{f}(1,1) = 0, \\ n = 3, & \text{f}(1,1,1) = 0, \\ n > 3, & \text{f}(1,\ldots,1) = 0 \end{cases} \tag{38}$$

En optimización matemática, la función de Rosenbrock es una función no convexa, introducida por Howard H. Rosenbrock en 1960, que se utiliza como un problema de prueba de rendimiento para algoritmos de optimización. También se conoce como el valle de Rosenbrock o la función del plátano de Rosenbrock. El mínimo global está dentro de un valle plano largo, estrecho y de forma parabólica como se observa en la Figura 8. Encontrar el valle es trivial y converger al mínimo global es difícil [27].

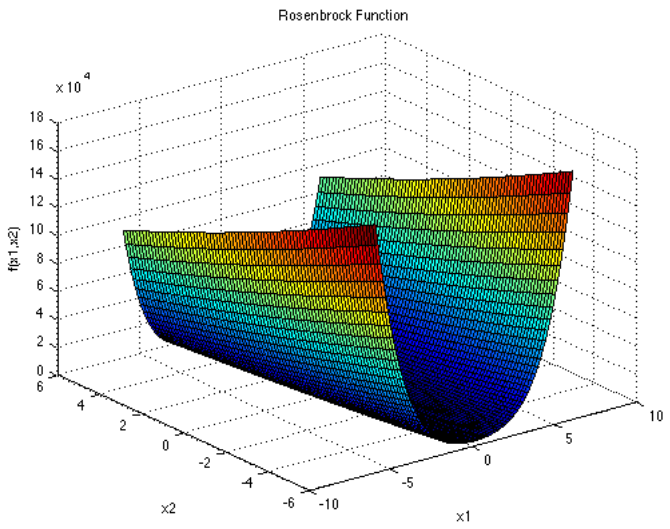


Figura 8: Comportamiento de la función de Rosenbrock [27].

6.12.3. Función de Booth

$$f(x) = (x + 2y - 7)^2 + (2x + y - 5)^2 \qquad f(1,3) = 0 \tag{39}$$

La función de booth se representa en la ecuación (39), esta función es usualmente evaluada en el cuadro $x_i \in -10, 10$ para todos los valores de $i = 1, 2, \ldots, n$. Como se puede observar, posee el mínimo absoluto descentrado del origen [27].

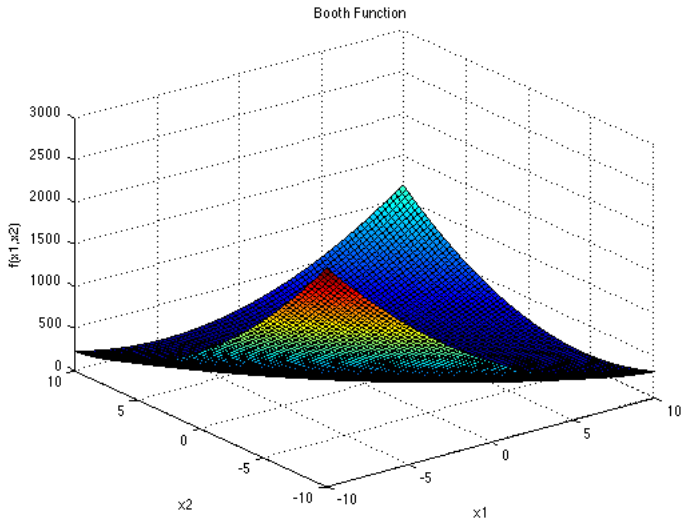


Figura 9: Comportamiento de la función de booth [27].

6.12.4. Función de Himmelblau

$$f(x) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \qquad Min = \begin{cases} f(3,2) = 0, \\ f(-2.805, 3.1313) = 0, \\ f(-3.779, -3.283) = 0, \\ f(3.584, -1.848) = 0 \end{cases} \tag{40}$$

Esta función de costos posee múltiples mínimos que poseen el mismo valor. Por lo tanto, todos son mínimos absolutos. Es útil para determinar influencia de posición inicial de partículas sobre decisión de punto de convergencia final. esta función es usualmente evaluada en $-5 \leq x, y \leq 5$ [27].

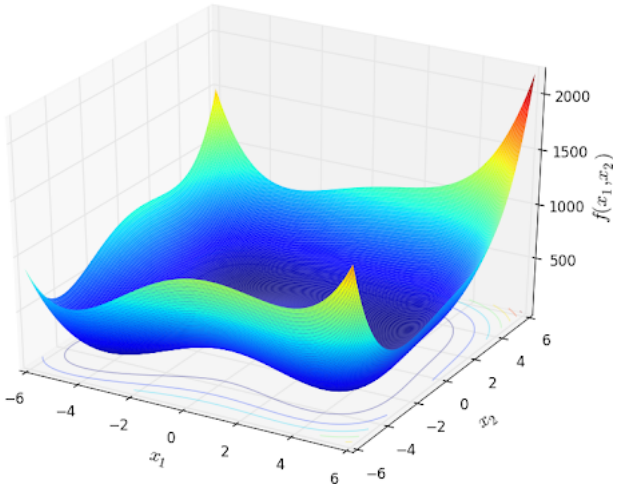


Figura 10: Comportamiento de la función de Himmelblau [27].

6.13. Webots

Figura 11: Logo de Webots [28].

Webots es una Multi-plataforma Open-Source (Código abierto) utilizada para realizar simulaciones de robots en entornos más cercanos a la realidad. Provee un entorno completo de desarrollo para modelar, programar y simular robots. Esta plataforma fue desarrollada por el Dr. Oliver Michel del Instituto Federal Suizo de Tecnología para fines educativos. Utiliza el Open Dynamics Engine que Permite la simulación de colisiones y la dinámica de cuerpos rígidos creando así un entorno mas real. El programa permite utilizar controladores escritos en C, C++, Java, Python, MATLAB y ROS [28].

Webots utiliza nodos para representar cada una de sus funciones dentro del entorno. Los nodos en webots se basan en la sintaxis del estándar VRML97. Webots utiliza un subconjunto de nodos y campos definidos por en estandar VRML97, pero también es posible especificar un nodo dentro de las definiciones de un robot en especifico[29].

El VRML o *Virtual Reality Modeling Language* es un formato de archivos enfocado en describir la interacción de objetos 3D y mundos virtuales. VRML fue diseñado para ser utilizado en la internet, intranet o sistemas locales. VRML busca ser un formato de intercambio universal para gráficas 3D integradas y multimedia [30].

VRML es capaz de representar objetos 3D animados y estáticos con hipervínculos dirigidos a media como textos, sonidos, películas o imágenes. VRML admite un modelo de extensibilidad que permite definir nuevos objetos 3D dinámicos, lo que permite que las comunidades de aplicaciones desarrollen extensiones interoperables para el estándar base. Las asignaciones entre los objetos VRML y las características de la interfaz del programador de aplicaciones (*Application Programmer Interface* -API) 3D de uso común [30].

6.14. Robot Operating System (ROS)

Robot Operating System (ROS) es una colección de marcos para el desarrollo de *software* de robots. ROS se desarrolló originariamente en 2007 bajo el nombre de switchyard por el Laboratorio de Inteligencia Artificial de Stanford para dar soporte al proyecto del Robot con Inteligencia Artificial de Stanford (STAIR2). Desde 2008, el desarrollo continuó principalmente en Willow Garage, un instituto de investigación robótico con más de veinte instituciones colaborando en un modelo de desarrollo federado [31].

Los procesos ROS se representan como nodos (*ROS nodes*) en una estructura gráfica, conectados por bordes llamados temas (*ROS topics*). Los ROS nodos pueden pasar mensajes entre sí a través de los temas, realizar llamadas de servicio a otros nodos, proporcionar un servicio para otros nodos o establecer o recuperar datos compartidos de una base de datos común llamada servidor de parámetros. Un proceso llamado maestro (*ROS Master*) hace que todo esto sea posible al registrar nodos para sí mismo, configurar la comunicación de nodo a nodo para los temas y controlar las actualizaciones del servidor de parámetros [32].

A pesar de no ser un sistema operativo, ROS provee los servicios estándar de uno de estos tales como la abstracción del *hardware*, el control de dispositivos de bajo nivel, la implementación de funcionalidad de uso común, el paso de mensajes entre procesos y el mantenimiento de paquetes. Se basa en una arquitectura de grafos donde el procesamiento toma lugar en los nodos que pueden recibir, mandar y multiplexar mensajes de sensores,

control, estados, planificaciones y actuadores, entre otros [32].

La librería está orientada para un sistema UNIX (Ubuntu -Linux-). ROS tiene dos partes básicas: la parte del sistema operativo (ROS) y ros-pkg. El ros-pkg consiste en un conjunto de paquetes aportados por la contribución de usuarios (organizados en conjuntos llamados pilas o *stacks*) que implementan las funcionalidades tales como localización y mapeo simultáneo, planificación, percepción, simulación, etc. Debido a esto es un sistema bastante completo para la realización de diversas simulaciones las cuales nos brindan resultados muy confiables y similares a lo que se vería en la vida real [31].

Al igual que Webots, ROS utiliza nodos para realizar la comunicación entre sus diferentes componentes. Un "nodo" es un proceso que realiza cálculos. Los nodos se combinan en un gráfico y se comunican entre sí mediante la transmisión de temas (*topics*), Servicios RPC (*RPC services*) y el Servidor de parámetros (*Parameter Server*). Estos nodos están destinados a operar a una escala detallada; un sistema de control de robot generalmente comprenderá muchos nodos. Por ejemplo, un nodo controla un telémetro láser, un nodo controla los motores de la rueda del robot, un nodo realiza la localización, un nodo realiza la planificación de la ruta, un nodo proporciona una vista gráfica del sistema, etc [33].

El uso de nodos en ROS proporciona varios beneficios al sistema en general. Existe una "tolerancia a fallos" adicional, ya que los bloqueos se aíslan en nodos individuales. La "complejidad del código" se reduce en comparación con los sistemas monolíticos. Los detalles de implementación también están bien ocultos, ya que los nodos exponen una API mínima al resto del gráfico y las implementaciones alternativas, incluso en otros lenguajes de programación, pueden sustituirse fácilmente. Un nodo de ROS se escribe con el uso de una librería de cliente de ROS (*ROS client library*) como roscpp o rospy [33].

6.15. Gazebo

Gazebo es un simulador 3D mientras que ros sirve como una interfaz para los robots. Combinando ambos se obtiene un simulador muy completo y versátil. Con Gazebo se pueden crear escenarios en 3 dimensiones con robots, obstáculos y muchos otros objetos. Gazebo utiliza múltiples motores físicos tales como ODE y Bullet para simular los efectos de iluminación, gravedad, inercia, etc. Se puede evaluar y probar el rendimiento de los robots en situaciones peligrosas o difíciles sin dañar físicamente el robot [34].

Originalmente Gazebo fue diseñado para evaluar algoritmos de robots. Para muchas aplicaciones es esencial probar la aplicación del robot para resolver errores, vida de la batería, localización, navegación, etc. Gazebo puede modelar sensores que son capaces de "ver" el entorno, tales como sensores láser, cámaras, sensores cinéticos, etc [34].

Gazebo es de código abierto y fue un componente en *The player project* del 2004 al 2011. Gazebo se volvió independiente cuando Willow Garage apoyo el proyecto. En 2012, *Open Source Robotics Foundation* (OSRF) se convirtió en el administrador del proyecto Gazebo. En 2018 OSRF cambio su nombre a *Open Robotics* que es el actual dueño del proyecto Gazebo [34].

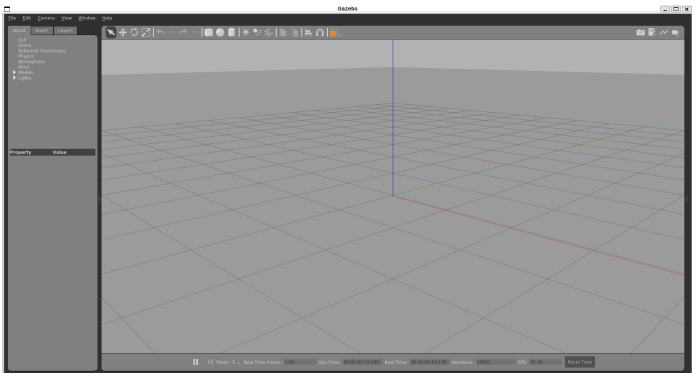


Figura 12: Entorno de Gazebo

6.16. Rviz

Rviz, abreviatura de visualización ROS, es una poderosa herramienta de visualización 3D para ROS. Permite al usuario ver el modelo de robot simulado, registrar la información de los sensores del robot y reproducir la información del sensor registrada. Al visualizar lo que el robot ve, piensa y hace, el usuario puede depurar una aplicación de robot desde las entradas del sensor hasta las acciones planificadas (o no planificadas)[35].

Rviz muestra datos de sensores 3D de cámaras estéreo, láseres, *Kinects* y otros dispositivos 3D en forma de nubes de puntos o imágenes de profundidad. Los datos de sensores 2D de cámaras web, cámaras RGB y telémetros láser 2D se pueden ver en rviz como datos de imagen [35].

Si un robot real se está comunicando con una estación de trabajo que ejecuta rviz, rviz mostrará la configuración actual del robot en el modelo de robot virtual. Los temas(*ROS topics*) de ROS se mostrarán como representaciones en vivo basadas en los datos del sensor publicados por las cámaras, sensores infrarrojos y escáneres láser que forman parte del sistema del robot. Esto puede ser útil para desarrollar y depurar el código base del robot [35].

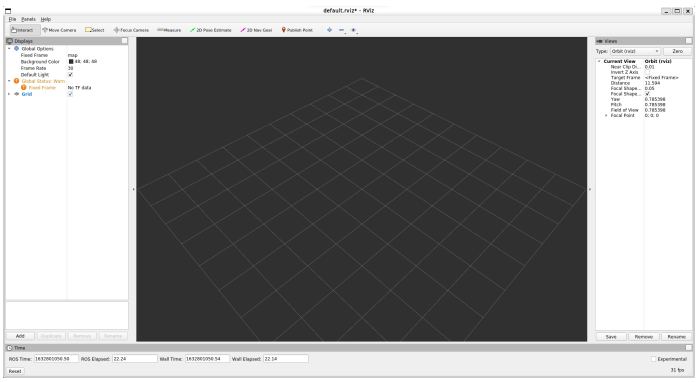


Figura 13: Entorno de Rviz

6.17. Lenguaje de programación XML

XML, siglas en inglés de eXtensible Markup Language (lenguaje de marcas extensible), es un lenguaje de marcas desarrollado por el World Wide Web Consortium (W3C) utilizado para almacenar datos de forma legible. XML permite definir etiquetas personalizadas para la descripción y la organización de datos. Además permite una utilización efectiva en Internet para sus diferentes terminales de una manera segura, fiable y fácil. Se puede usar en bases de datos, editores de texto, hojas de cálculo, etc [36].

Ventajas del XML:

- Es ampliable: Se puede extender fácilmente el código XML con la adición de nuevas etiquetas, de modo que se pueda continuar utilizando sin ninguna complicación [36].
- El analizador es un componente estándar, no es necesario crear un analizador específico para cada versión de lenguaje XML. Esto posibilita el empleo de cualquiera de los analizadores disponibles. De esta manera se evitan bugs y se acelera el desarrollo de aplicaciones [36].
- Al ser de estructura jerárquica, es sencillo entender su estructura y procesarla. Mejora la compatibilidad entre aplicaciones [36].
- Su análisis sintáctico es fácil debido a las estrictas reglas que rigen la composición de un documento [36].

6.17.1. Estructura de un documento XML

La tecnología XML mantiene la información estructurada de la forma más abstracta y reutilizable posible. Esto quiere decir que se compone de partes bien definidas, y que esas partes se componen a su vez de otras partes. Por esta razón, presenta una estructura jerárquica. A esas partes se las llaman elementos, y se las señala mediante etiquetas. Una etiqueta es una marca hecha en el documento, que señala un fragmento de éste como un elemento. Las etiquetas tienen la forma <nombre>, donde nombre es el nombre del elemento que se está señalando [36].

6.17.2. Partes de un documento XML

- **Prólogo**: Los documentos XML pueden empezar con unas líneas que describen la versión XML, el tipo de documento y otras cosas. No es obligatorio ponerlo, pero es recomendable hacerlo para un mejor entendimiento.
  Por ejemplo: *<?xml version="1.0" encoding="UTF-8"?>*
- **Cuerpo**: El cuerpo tiene que contener solo un elemento raíz, característica obligatoria para que el documento esté bien formado. Sin embargo es necesaria la adquisición de datos para su buen funcionamiento.
  Por ejemplo: *<Edit_ Mensaje> (...) </ Edit_ Mensaje>*

- **Elementos**: Los elementos XML pueden tener contenido (más elementos, caracteres o ambos) o bien ser elementos vacíos.
- **Atributos**: Los elementos pueden tener atributos, que son una manera de incorporar características o propiedades a los elementos de un documento. Deben ir entre comillas.
- **Entidades predefinidas**: Entidades para representar caracteres especiales para que no sean interpretados como marcado en el procesador XML.
  Por ejemplo: *entidad predefinida:* & *carácter:* &
- **Comentarios**: Son aclaraciones que pueden ser escritas para informar al lector, y son ignorados por el procesado. Los comentarios en XML mantienen el siguiente formato:
  <!— Esto es un comentario —>

6.18. Formato de descripción para robots - URDF

La *Unified Robot Description Format*(URDF) es una especificación XML que se utiliza para describir la estructura de un robot. Dentro de este sistema de descripción, solo se aceptan estructuras tipo árbol las cuales establecen que existe una dependencia entre las conexiones de las juntas. Esto nos indica que estructuras de robots paralelos no son aceptadas dentro de este tipo de descripción. Por otro lado, la especificación asume eslabónes rígidos conectados por eslabónes, por lo que elementos que presentan comportamientos flexibles no son aceptados [36]. Dentro de esta especificación se incluye:

- Descripción cinemática y dinámica del robot
- Representación visual del robot
- Modelo de colisiones del robot

La descripción de un robot se hace a partir de un conjunto de eslabónes (*link*) y de un conjunto de elementos de unión (*joints*) que son las conexiones entre los eslabónes [36]. Esta conexión realizada entre los eslabónes definen toda la estructura del sistema al igual que la interacción que tendrá con el entorno. A continuación se describe como se definen los eslabónes y juntas dentro del archivo URDF para describir a cualquier tipo de robot.

6.18.1. eslabónes (*links*)

Estos elementos describen un cuerpo rígido con una inercia y características visuales. Los eslabónes simbolizan un elemento importante de la arquitectura del manipulador: un brazo para posición, una muñeca para dar destreza y un efector final que realiza una tarea específica.

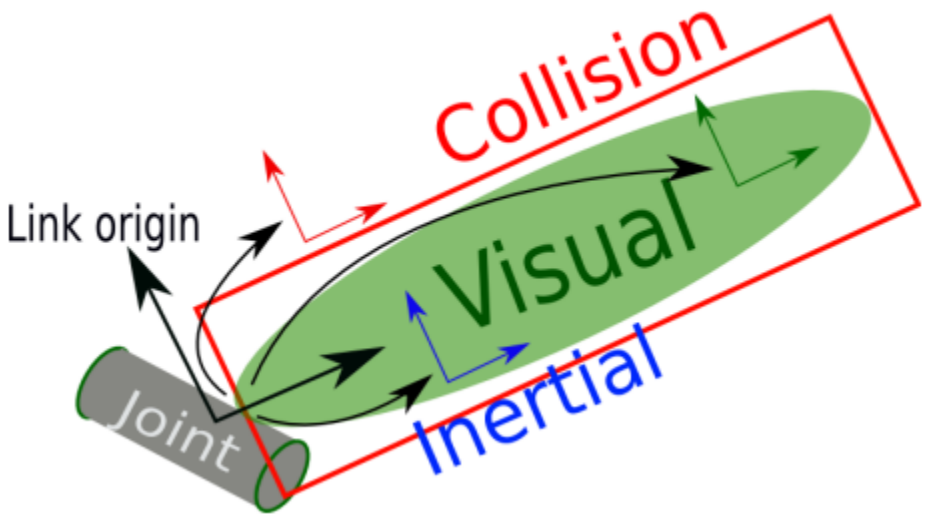


Figura 14: Elemento eslabón [36].

Atributos:

Nombre (necesario) nombre del eslabón.

Elementos:

<**inertia**> **(opcional)** Propiedades inerciales del eslabón.

<**origin**> **(opcional: es la identidad por defecto)** Posición del marco de referencia inercial con respecto al marco de referencia del eslabón. Sus ejes no necesariamente deben alinearse con los ejes principales de inercia.

xyz (opcional: cero por defecto) Representa el desplazamiento del robot con respecto al origen.

rpy (opcional: es la identidad por defecto) Representa los ángulos de giro (*roll, pitch, yaw*)

<**mass**> Valor de la masa del eslabón.

<**inertia**> Indica la matriz rotacional de inercia (3×3), representada en el marco inercial. Solo se especifican 6 atributos ($i_{xx}, i_{xz}, i_{yy}, i_{yz}, i_{zz}$) ya que la matriz rotacional de inercia es simétrica.

<**visual**> **(opcional)** Indica las propiedades visuales del eslabón. Este elemento especifica la forma del objeto. Se puede escribir más de una etiqueta para el mismo eslabón.

name (opcional) Especifica un nombre para una parte de la geometría del eslabón.

<**origin**> **(opcional: es la identidad por defecto)** El marco de referencia del elemento visual con respecto al marco de referencia del eslabón.

xyz (opcional: cero por defecto) Representa el desplazamiento del robot con respecto al origen.

rpy (opcional: es la identidad por defecto) Representa los ángulos de giro (*roll, pitch, yaw*)

<**geometry**> **(necesario)** La forma que tendrá el objeto visualmente.

<**box**> El atributo de tamaño o (*size*) representa la longitud de los lados del cubo. su origen se encuentra en su centro o la mitad del tamaño.

<**cylinder**> Se puede especificar el radio y la altura con los comandos *radius* y *lenght*. Su origen se encuentra en su centro.

<**sphere**> Se puede especificar el radio de la esfera con el comando *radius*. Su origen se encuentra en su centro.

<**mesh**> Se puede utilizar una figura tridimensional especificada por un archivo y se puede escalar su tamaño. Se recomienda archivos .dae para mejor textura y color aunque también acepta archivos .stl.

<**material**> **(opcional)** Es el material del elemento visual. Se puede especificar previamente y solo hacer referencia al mismo.

<**color**> **(opcional)** Se utiliza el sistema rgba lo cual especifica el color utilizando cuatro valores que representan la intensidad de los colores rojo, verde, azul y la opacidad.

<**texture**> **(opcional)** La textura que tendrá el material definida por un archivo.

<**collision**> **(opcional)** Define las propiedades de un eslabón al colisionar. Puede ser diferente de las propiedades visuales. Pueden existir varias propiedades para un mismo eslabón.

name (opcional) Especifica el nombre para una parte específica de la geometría del eslabón.

<**origin**> **(opcional: es la identidad por defecto)** El marco de referencia del elemento visual con respecto al marco de referencia del eslabón.

<**geometry**> La forma que tendrá el objeto visualmente.

6.18.2. Juntas (*joints*)

Estos elementos definen las propiedades cinemáticas y dinámicas. Por otro lado, se definen los límites de seguridad para el movimiento de la articulación del robot.

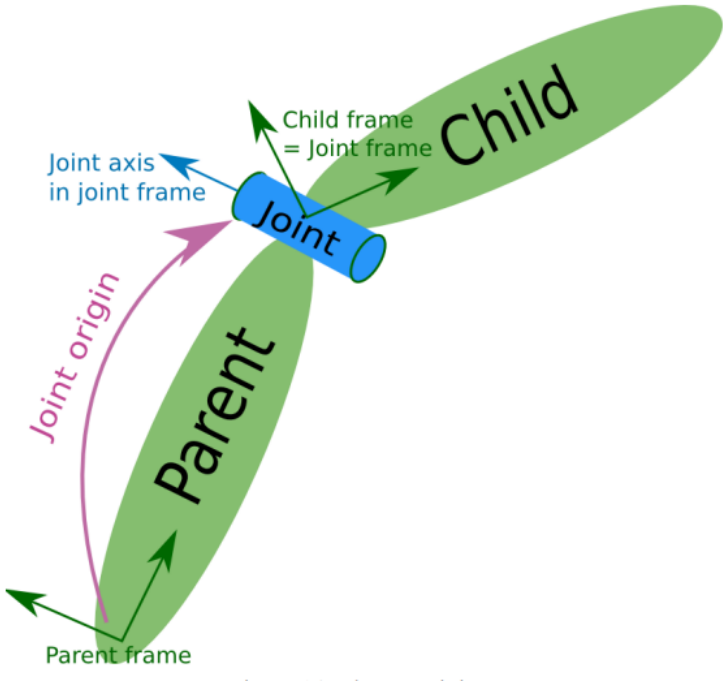


Figura 15: Elemento junta [36].

Atributos:

Name (necesario) Es el nombre de la articulación.

Type (necesario) Establece el tipo de articulación, pueden ser:

- ***revolute*:** una articulación de bisagra que gira a lo largo del eje y tiene un rango limitado especificado por los límites superiores e inferiores.
- ***continuous*:** una articulación en bisagra continua que gira entorno al eje y no tiene límites superiores o inferiores.
- ***prismatic*:** una junta deslizante que se desliza a lo largo del eje y tiene un rango limitado especificado por los límites superior e inferior.
- ***fixed*:** Es una junta que no puede moverse ya que todos los grados de libertad están bloqueados.
- ***floating*:** es una articulación que permite el movimiento de los 6 grados de libertad.
- ***planar*:** es una articulación que permite el movimiento en un plano perpendicular al eje.

Elementos:

<**origin**> **(opcional: es la identidad por defecto)** Es la transformada desde el eslabón padre con el eslabón hijo como se observa en la Figura 15. La articulación se encuentra en el origen del eslabón hijo.

xyz (opcional: cero por defecto) Representa el desplazamiento del robot con respecto al origen.

rpy (opcional: es la identidad por defecto) Representa los ángulos de giro (*roll, pitch, yaw*) en radianes.

<**parent**> **(necesario)** El nombre del eslabón padre.

link El nombre del eslabón padre que es el padre de el eslabón actual según la jerarquía del árbol del robot.

<**child**> **(necesario)** El nombre del eslabón hijo.

link El nombre del eslabón hijo de el eslabón actual según la jerarquía del árbol del robot.

<**axis**> **(opcional: por defecto** $(1, 0, 0)$**)** Este es el eje de rotación de articulaciones de giro, el eje de traslación para juntas prismáticas y la superficie normal para juntas planas. El eje se especifica en el marco común de referencia.

xyz (necesario) Representa la componente de un vector x,y y z. El vector debe estar normalizado

<**calibration**> **(opcional)** la posición de referencia de al articulación, usada para calibrar la posición absoluta de la articulación.

- ***rising*****(opcional)** Cuando la articulación se mueve en una dirección positiva, esta posición de referencia dará lugar a un flanco ascendente.
- ***falling*****(opcional)** Cuando la articulación se mueve en una dirección negativa, esta posición de referencia dará lugar a un flanco descendente.

<**dynamic**> **(opcional)** Un elemento que especifica las propiedades físicas de la articulación. Estos valores se utilizan para definir las propiedades de modelado de la articulación especialmente útil para la simulación.

- ***damping*****(opcional: cero por defecto)** El valor de amortiguación fisica de la articulación.
- ***friction*****(opcional: cero por defecto)** El valor de fricción estática física de la articulación.

<**limit**> **(para prismáticas y revolutas)** Un elemento que puede contener los siguientes atributos:

- ***lower*****(opcional: cero por defecto)** un atributo que especifica el límite inferior de unión (radianes para juntas de revolución y metros para articulaciones prismáticas).

- ***upper*(opcional: cero por defecto)** un atributo que especifica el límite superior conjunta (radianes para juntas de revolución y metros para articulaciones prismáticas).
- ***effort*(requerido)** un atributo para el máximo esfuerzo de la articulación.
- ***velocity*(requerido)** define la velocidad máxima de la articulación.

<**mimic**> **(opcional)** Esta etiqueta se utiliza para especificar que la junta actual imita a otra articulación existente.

- ***joint*(requerido)** Especifica el nombre de la articulación a imitar.
- ***multiplier*(opcional)** Especifica el factor multiplicativo en la fórmula anterior.
- ***offset*(opcional)** Especifica el desplazamiento para agregar en la fórmula anterior.

<**safety_controller**> **(opcional)** Un elemento que contiene algunos de los siguientes atributos:

- ***soft_lower_limit*(opcional: cero por defecto)** atributo que especifica el límite inferior de la articulación donde el controlador de seguridad comienza a limitar la posición de la articulación.
- ***soft_upper_limit*(opcional: cero por defecto)** atributo que especifica el límite superior de la articulación donde el controlador de seguridad comienza a limitar la posición de la articulación.
- ***k_position*(opcional: cero por defecto)** atributo que especifica la relación entre los límites de posición y velocidad.
- ***k_velocity*(requerido)** atributo que especifica la relación entre el esfuerzo y los límites de velocidad.

6.19. Lenguaje Macro XML

Lenguaje Macro XML o Xacro es un lenguaje de macros XML. Con xacro, puede construir archivos XML más cortos y legibles mediante el uso de macros que se expanden a expresiones XML más grandes. El lenguaje de macros XML, Xacro, amplía el formato de descripción de robots universales (URDF) y es parte de un cadena de herramientas críticas que abarca desde representaciones geométricas hasta simulación, visualización y ejecución del sistema. Los comandos de Xacro brindan un poder expresivo en los modelos de robots para lógica condicional, variables y tomar ventaja de simetrías físicas (una simetría podría ser dos brazos robóticos que se reflejan entre sí). El problema es que el poder expresivo de los archivos xacros poseen una curva de aprendizaje más pronunciada y las representaciones geométricas codificadas en Xacros son necesarias para lanzar la mayoría de los sistemas ROS, visualizar datos (Rviz) o ejecutar simulaciones (Gazebo) [37].

CAPÍTULO 7

Diseño experimental en Webots

7.1. Desarrollo de algoritmo PSO

Para el desarrollo del algoritmo PSO se utilizó la posición de los robots diferenciales obtenida por un nodo de GPS que nos brinda la posición y la orientación de los robots en un momento determinado del tiempo. Se tomaron los datos de la posición del robot para comparar la mejor posición que ha tenido el robot con la posición actual del mismo ($p_i^{mejor} - p_i(t)$). Este cálculo se utiliza dentro de la ecuación 41.

$$v_i(t+1) = v_i(t) + c_1 r_1 (p_i^{mejor} - p_i(t)) + c_2 r_2 (p_g^{mejor} - p_i(t)) \tag{41}$$

En el caso de la mejor posición global se utiliza un nodo transmisor y un nodo receptor dentro del entorno de webots. El nodo emisor se utiliza para modelar emisores de radio, serie o infrarrojos. Este tipo de nodos solo es capaz de enviar información pero no de recibirla. Por otro lado, un nodo receptor también se utiliza para modelar receptores de radio, serie o infrarrojos, pero a diferencia del emisor, este solo es capaz de recibir información. Si se desea tener una comunicación unidireccional entre dos robots, uno de ellos debe de poseer un nodo emisor mientras el otro debe de tener un nodo receptor. En el caso de querer tener una comunicación bidireccional, ambos robots deben de contar con un nodo emisor y un nodo receptor.

Para poder compartir los datos de la mejor posición global entre los robots se debe tener una comunicación bidireccional entre todos los robots, esto significa que todos los robots deben de tener un nodo receptor y un nodo emisor. Dentro de las conFiguraciones que pueden tener estos dos nodos, nos encontramos con el tamaño de los bytes, el canal por el cual se transmitirá la información, el tamaño del *buffer*, el rango, el tipo de señal, etc. En este caso, nos interesa el canal por el cual se realiza la comunicación, este es un

numero de identificación para un emisor de “infrarrojos” o una frecuencia para un emisor de “radio”. Normalmente un receptor debe utilizar el mismo canal que un emisor para recibir los datos emitidos. El canal 0 (el predeterminado) está reservado para comunicarse con un complemento de física. Debido a esto, se decidió seleccionar el canal de comunicación No.1.

Ya teniendo claro cuál será el canal que se utilizará para transmitir los datos, se procede a determinar cuáles son los datos relevantes que se desean transmitir. En este caso, nos interesa transmitir la mejor posición del robot y verificar si los otros robots han tenido una mejor posición. Para esto, se determina que se necesita un *buffer* capaz de contener 3 datos, siendo estos el valor de la posición del robot y el valor obtenido por el algoritmo PSO en esa posición. Si el robot con el cual estamos comparando la posición posee un mejor valor, este valor pasa a tener el lugar de la mejor posición global de todo el arreglo de robots. A partir de estos datos se comparó la mejor posición actual del robot con la mejor posición global del arreglo. Lo cual corresponde a la segunda parte de la ecuación (41) en donde se determina la diferencia entre estos valores $(p_g^{mejor} - p_i(t))$.

Con esta información ya es posible calcular las posiciones futuras de cada uno de los robots del PSO. Cabe destacar que cada robot calcula su propia función del PSO y la mejor posición del robot y la posición actual son calculadas internamente, mientras que la mejor posición global si es compartida entre todos los robots utilizando los nodos emisores y receptores dentro de cada robot.

Las constantes del algoritmo fueron obtenidas del *Toolbox* creado por Eduardo Santizo. Dentro de este *Toolbox* se calculan los valores de c_1 y c_2 óptimos para el algoritmo PSO los cuales son ingresados dentro de la definición en el mundo de Webots. Estos parámetros buscan una convergencia mas rápida hacia el mínimo de la función por parte de todos los robots diferenciales.

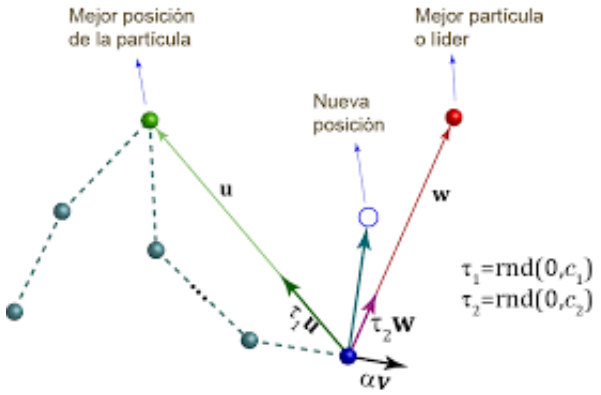
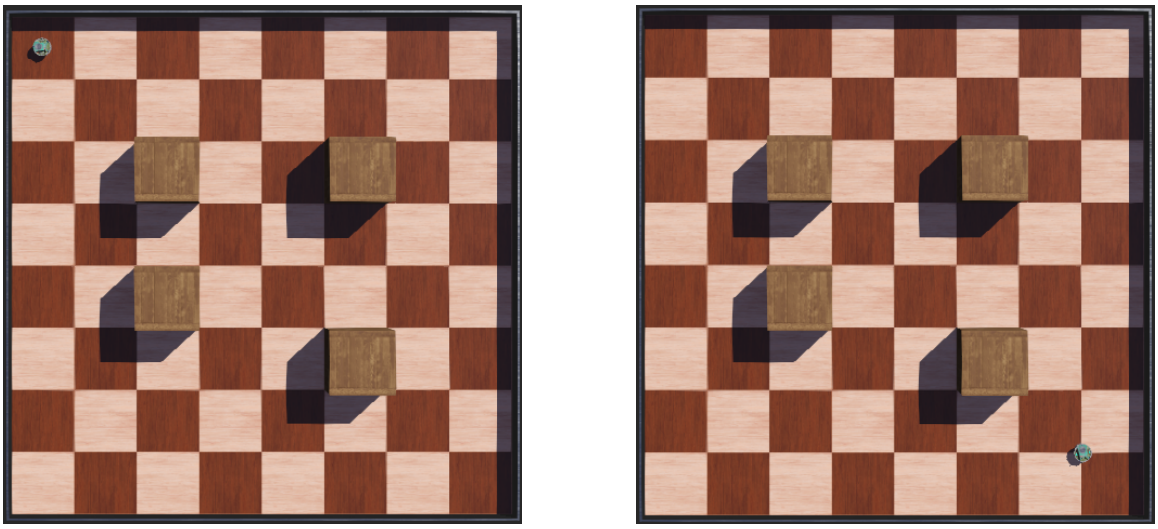


Figura 16: Representación gráfica del comportamiento del algoritmo PSO [38]

7.2. Algoritmo D* Lite para evasión de obstáculos

Se logró implementar un el algoritmo D* Lite el cual era capaz de navegar por un terreno inicialmente vacío y al momento de encontrar un obstáculo cerca del robot, el algoritmo recalculaba la nueva trayectoria que debía seguir el robot para evitar colisionar con el obstáculo. Para esto, se tuvo que discretizar el espacio creando segmentos con distancia δ tanto en el eje x como en el eje z creando así una cuadrícula con vértices hacia los cuales el robot era capaz de moverse. Se estableció $s_{meta} = (0.75, 0.75)$ y $s_{inicio} = (0.125, 0.125)$ en el eje x y z respectivamente. Estos puntos se pueden observar dentro del entorno de Webots en las Figuras 17a y 17b.



(a) Posición inicial

(b) Posición final

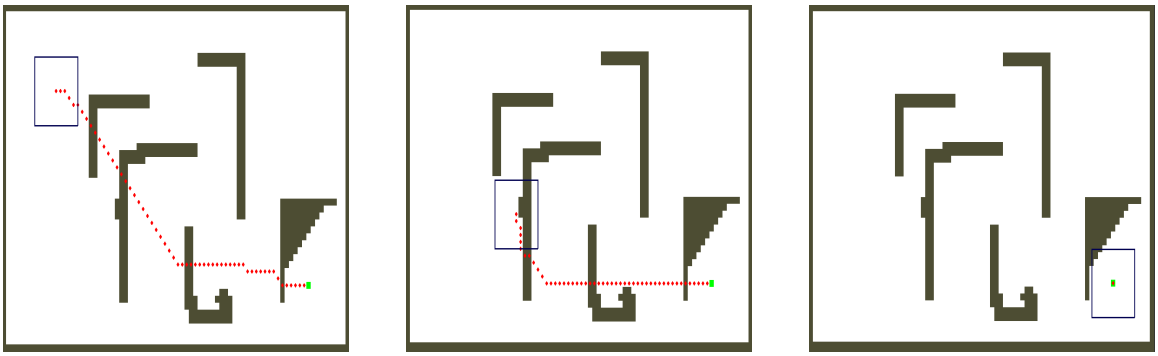
Figura 17: Posición inicial y final del robot en el *Gridworld*

Como se puede observar en las imágenes mostradas en la Figura 17 el *Gridworld* consta de ocho cuadros y la extensión del terreno es de 1 metro $\times$ 1 metro. por lo tanto se estableció un $\delta = 0.0625$ metros para obtener una cuadrícula de 16×16 nodos. Esta cuadrícula de 16×16 nodos fue definida dentro del entorno de Webots en lenguaje C para establecer las posiciones a las cuales el robot es capaz de moverse dentro del entorno.

Tomando estas posiciones a las cuales se podría mover el robot se calculó el valor de $rhs(s)$ para cada nodo. El peso de cada nodo corresponde a la distancia entre cada nodo. Se permitió el movimiento diagonal del robot a lo largo de los nodos definidos y se agregó una penalización de 1.5 para este tipo de movimientos diagonales.

Se definió una lista de prioridad de tamaño 16^2 lo cual nos daba una lista de 256 datos. La razón de hacer la predefinición del tamaño de las variables se debe a que el lenguaje C no se pueden utilizar arreglos dinámicos. Un arreglo dinámico es un arreglo de elementos que crece dinámicamente conforme los elementos se agregan o se eliminan.

Se realizaron simulaciones preliminares dentro del entorno de python para evaluar la velocidad con la cual el algoritmo recalcula las trayectorias entre la posición inicial (s_{start}) y la posición final (s_{final}). Como se pueden observar en la Figura 18 el algoritmo es capaz de recalcular de manera efectiva la trayectoria que debe tomar el robot para llegar a la posición final y la velocidad a la cual lo hace es casi instantáneo. Se puede observar todos los cambios en la trayectoria del algoritmo en las Figuras 50, 51 y 52 del anexo. Dentro de las imágenes mostradas en la Figura 18 se puede observar un cuadrado alrededor de nuestro punto móvil, este cuadrado simula la visión y el alcance de nuestros sensores dentro del entorno.



(a) Posición inicial (b) Posición media (c) Posición final

Figura 18: Trayectorias generadas por el algoritmo D* Lite

El objetivo principal de este algoritmo es lograr que el movimiento del robot, hacia los puntos específicos brindados por algoritmo PSO, sea lo más rápido posible mientras el sistema es capaz de evadir los obstáculos que encuentra en el camino. Al llegar al punto, el sistema calcula de nuevo el algoritmo PSO para encontrar el siguiente punto y así sucesivamente hasta lograr converger hacia el mínimo global.

7.3. Algoritmo de Braitenberg

Para el algoritmo de vehículos de braitenberg se utilizó robótica basada en comportamiento para generar el movimiento de evasión. La robótica basada en comportamiento es un enfoque de la robótica que se centra en aquellos robots capaces de exhibir comportamientos de apariencia compleja a pesar de pequeñas variables del estado interno para modelar su entorno, sobre todo corrigiendo gradualmente sus acciones a través de enlaces sensorio motores. Por lo tanto, mientras el robot no detecte ningún obstáculo cercano a él, seguirá moviéndose en la dirección del punto que sea nuestra meta. En el momento que el robot detecta un obstáculo cercano a él, utiliza el algoritmo de braitenberg para realizar las maniobras necesarias para evitar el obstáculo.

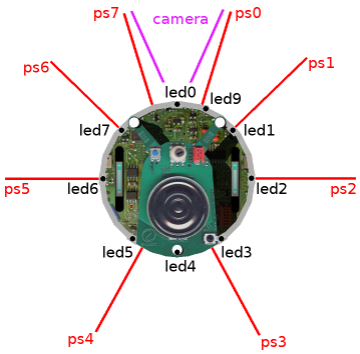


Figura 19: Sensores del robot e-puck [39]

Como se puede observar en la Figura 19, el robot e-puck cuenta con 8 sensores de distancia. Tomando en consideración que el giro del robot depende de la velocidad de la rueda derecha (V_r) y de la rueda izquierda (V_l). Se puede establecer que si $V_r > V_l$ el robot realiza un giro a la derecha, si $V_l > V_r$ el robot realiza un giro a la izquierda y si $V_r = V_l$ el robot se mueve hacia adelante si las velocidades son positivas o hacia atrás si las velocidades son negativas. Bajo esta definición, se puede establecer el comportamiento del robot como:

$$||V_r|| = ||V_r||_0 + \sum_{k=1}^{2} \frac{\alpha_{1k}(||s_k||)}{\rho}$$

$$||V_l|| = ||V_l||_0 + \sum_{k=1}^{2} \frac{\alpha_{2k}(||s_k||)}{\rho}$$

En donde:

- $||V_l||_0$ = velocidad constante.
- ρ = rango de los sensores (por lo general se normaliza obteniendo un valor entre 0 y 1).
- α = peso de cada uno de los sensores sobre cada uno de los motores.

Teniendo esto definido, se puede emplear producto matricial para encontrar la forma compacta del controlador de braitenberg, finalmente la ecuación queda de la siguiente manera:

$$\begin{bmatrix} ||V_r|| \\ ||V_l|| \end{bmatrix} = \begin{bmatrix} ||V_r||_0 \\ ||V_l||_0 \end{bmatrix} + As$$

En donde, la matriz A corresponde a los pesos de cada uno de los sensores y el vector s corresponde a los valores de cada uno de los sensores s_k, en un tiempo específico, dividido por su rango (ρ). Para determinar los valores de la matriz de A se tomó en consideración que los sensores en la parte de atrás del robot ($ps3, ps4$) relativo a la Figura 19 no se utilizarían, por

lo tanto el valor de $\alpha = 0$ para estos sensores. Para los sensores laterales $(ps5, ps2)$ relativo a la Figura 19, solo requerimos que el robot gire lo suficiente para mantener un movimiento en linea recta. Finalmente para los cuatro sensores en la parte frontal $(ps0, ps1, ps7, ps6)$ relativo a la Figura 19, se requiere que el robot gire sobre su eje, por lo que se le asigna un peso mayor a estos sensores. Finalmente, definimos la matriz A y el vector s como:

$$A = \begin{bmatrix} \alpha_{11} & \alpha_{21} & \cdots & \alpha_{k1} \\ \alpha_{12} & \alpha_{22} & \cdots & \alpha_{k2} \end{bmatrix} = \begin{bmatrix} 0.95 & 0.8 & 0.4 & 0 & 0 & -0.2 & -0.35 & -0.45 \\ -0.5 & -0.4 & -0.2 & 0 & 0 & 0.4 & 0.75 & 0.90 \end{bmatrix}$$

$$s = \frac{1}{\rho} \begin{bmatrix} ||s_1|| & ||s_2|| & \cdots & ||s_k|| \end{bmatrix}^T$$

7.4. Selección de controladores

Para la selección de los controladores óptimos se realizaron una serie de simulaciones dentro del entorno de Webots para encontrar el controlador que le tomara una menor cantidad de tiempo converger hacia el mínimo global de la función. Dentro de estas simulaciones se variaron las posiciones de los obstáculos dentro del entorno de webots generando cada vez entornos más difíciles.

Como se mencionó con anterioridad, se utilizó una aproximación de vehículos de braitenberg con robótica basada en comportamientos. Esto se debe a las limitaciones que presentaba el algoritmo D* Lite a la hora de ser implementado dentro del mundo de Webots. Utilizando robótica basada en comportamientos, el robot era capaz de poder determinar, dependiendo de varios factores, si debería moverse hacia el punto o alrededor de un obstáculo.

Entre los controladores PID, LQR y LQI, se tomó en consideración tiempos de convergencia entre todos los controladores en diferentes escenarios y diversas posiciones iniciales. Esto se realizó para tener un aproximado de cuál es la variación de los datos en comparación con el cambio o la dificultad del entorno simulado.

Para los parámetros utilizados para cada uno de los controladores, se tomaron los parámetros establecidos dentro del *Toolbox* realizado por Eduardo Santizo [2] en donde establece que las constantes para el controlador PID son:

$$K_p = 0.5 \qquad K_i = 0.1 \qquad K_d = 0.001 \tag{42}$$

La matriz de constantes para el regulador cuadrático lineal (LQR) es igual a:

$$K_{lqr} = \begin{bmatrix} 1 \\ 1 \end{bmatrix} \tag{43}$$

La matriz de constantes para el Integrador cuadrático lineal (LQI) es igual a:

$$K = \begin{bmatrix} 0.2127 \\ 0.2127 \end{bmatrix} \qquad K_i = \begin{bmatrix} -0.0224 \\ -0.0224 \end{bmatrix} \tag{44}$$

CAPÍTULO 8

Robotic Operating System (ROS)

8.1. Entorno de ROS

8.1.1. Comandos básicos

La comunicación que se establece entre los nodos es gracias al *ROS Master*. El *ROS Master* proporciona servicios de nombres y registros a los nodos en el sistema ROS y realiza un seguimiento de los publicadores y suscriptores de los temas (*topics*). El papel del maestro es permitir que los nodos ROS individuales se ubiquen entre sí, para comunicarse entre sí de igual a igual [40].

Para crear un *ROS Master*, se utiliza la el comando "roscore". roscore es una colección de nodos y programas que son requisitos para un sistema basado en ROS y se debe tener un "roscore" ejecutándose para que se se pueda realizar la comunicación entre los diferentes nodos [40].

Para poder enviar instrucciones al robot y recibir información de sus sensores y odometría se deben crear nodos de comunicación, con distintos temas y mensajes. El comando de "rosnode" es una herramienta por línea de comandos que muestra información sobre los nodos ROS como se observa en la Figura 20. Incluyendo publicadores, suscriptores y conexiones [40].

Por otro lado, El comando "rqt_graph" es una herramienta muy útil para ver lo que sucede en el gráfico ROS. Es un complemento GUI (*Graphical User Interface*) del conjunto de herramientas Rqt. Con el comando "rqt_graph" se obtiene una visión global del sistema ROS. Esta herramienta también permite comprobar la comunicación entre los nodos y detectar posibles problemas [40]. Un ejemplo de como se observa este tipo de conexión de puede encontrar en la Figura 21.

```
turtlebot@turtlebot:~$ rosnode list
/app_manager
/bumper2pointcloud
/capability_server
/capability_server_nodelet_manager
/cmd_vel_mux
/diagnostic_aggregator
/interactions
/master
/mobile_base
/mobile_base_nodelet_manager
/robot_state_publisher
/rosout
/zeroconf/zeroconf
```

Figura 20: Ejemplo de visualización de los nodos de ROS [40].

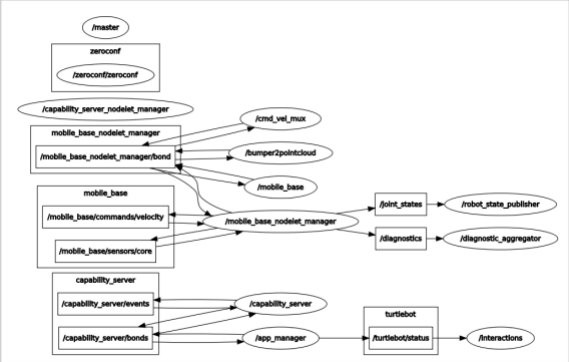


Figura 21: Ejemplo de la gráfica de conexiones entre los nodos [40].

El comando "rostopic" es otra herramienta que muestra información de los temas de ROS, incluidos los publicadores, suscriptores y mensajes ROS. Los temas son la manera que tienen los nodos de comunicarse entre ellos. A través de estos temas los nodos se pasan mensajes y pueden estar varios activos al mismo tiempo, como se observa en la Figura 22. Estos mensajes pueden tener una estructura predeterminada o bien puede ser creada por nosotros [40].

```
turtlebot@turtlebot:~$ rostopic list
/capability_server/bonds
/capability_server/events
/cmd_vel_mux/active
/cmd_vel_mux/input/navi
/cmd_vel_mux/input/safety_controller
/cmd_vel_mux/input/switch
/cmd_vel_mux/input/teleop
/cmd_vel_mux/parameter_descriptions
/cmd_vel_mux/parameter_updates
/diagnostics
/diagnostics_agg
/diagnostics_toplevel_state
/gateway/force_update
/gateway/gateway_info
/info
/interactions/interactive_clients
/interactions/pairing
/joint_states
/mobile_base/commands/controller_info
```

Figura 22: Ejemplo de visualización de los temas activos en ROS [40].

8.1.2. Componentes principales

Un sistema de comunicación es a menudo una de las primeras necesidades que surgen en la implementación de una nueva aplicación robótica. El sistema de mensajería integrado de ROS le permite ahorrar tiempo mediante la gestión de los detalles de la comunicación entre los nodos distribuidos a través de publicaciones anónimas/método de suscripción. Otra ventaja de utilizar un sistema de de mensajes es que obliga a implementar interfaces claras entre los nodos del sistema, mejorando así la encapsulación y la promoción de la reutilización de código . La estructura de estas interfaces de mensaje se define en el mensaje de IDL (*Interface Description Language*) [36].

Años de discusión y desarrollo de la comunidad han dado lugar a un conjunto de formatos de mensaje estándar que cubren la mayor parte de los casos de uso común en la robótica. Hay definiciones de mensajes de conceptos geométricos como poses, transformadas y vectores; para sensores como cámaras, IMU y láser; y para los datos de navegación como odometría, rutas y mapas; entre muchos otros. Estos mensajes también incrementan la interoperabilidad con las herramientas y características existentes en el ecosistema de ROS.

ROS proporciona un conjunto de herramientas para describir y modelar el robot para que pueda ser comprendido por el resto de su sistema de ROS, incluyendo tf, *robot_state_publisher y rviz* [36]. El formato para la descripción de su robot de ROS es URDF (*Unified Robot Descripción Format*), que consiste en un documento XML en el que se describen las propiedades físicas de su robot, desde la longitud de las extremidades y tamaños de ruedas a la ubicación de los sensores y la apariencia visual de cada parte del robot. Una vez que se define de esta manera, el robot se puede utilizar fácilmente con la librería tf, representada en las tres dimensiones con visualizaciones detalladas, y se utiliza con los simuladores y los planificadores de movimiento.

8.2. Algoritmo de dijkstra en ROS

Es un algoritmo para la determinación del camino más corto, dado un vértice origen, hacia el resto de los vértices en un grafo que tiene pesos en cada arista. Su nombre alude a Edsger dijkstra, científico de la computación de los Países Bajos que lo concibió en 1956 y lo publicó por primera vez en 1959.

El algoritmo de dijkstra es un algoritmo globalmente conocido en donde el sistema busca encontrar el camino mas corto entre un posición y otra. Debido a esto, se decidió realizar una simulación de este algoritmo dentro del entorno de ROS, para evaluar la eficiencia de la implementación de este tipo de algoritmos. El algoritmo posee dos listas en donde se incluyen los nodos abiertos o no visitados y los nodos cerrados o nodos ya visitados. Llevando un control de este tipo, el algoritmo es capaz de navegar a través de un terreno relativamente desconocido.

El único inconveniente con el algoritmo de dijkstra esta en que el algoritmo itera sobre todos los nodos del espacio hasta que encuentra el nodo en donde esta la meta. Esto conlleva mucho costo computacional debido a que el algoritmo tiene que realizar el calculo para cada nodo no visitado que este alrededor de un nodo ya visitado, independientemente si la meta

se encuentra en esa dirección o no.

Tomando esto en consideración, el entorno de ROS cuenta con una herramienta visual la cual es capaz de discretizar un espacio para lograr obtener una cuadricula.El nombre de esta herramienta es Rviz y es una herramienta sumamente útil en lo que conlleva a observar información dentro del entorno de ROS. Esta herramienta nos permite elegir un nodo o un punto dentro de la cuadricula al cual se desea mover el robot y nos permite visualizar el avance de la búsqueda que se esta realizando en tiempo real.

La herramienta de Rviz se comunica directamente con el simulador de Gazebo, cuando el sistema encuentra una trayectoria por la cual el robot puede pasar y luego esta información se pasa al simulador que se encarga del movimiento del robot. El espacio de búsqueda, los nodos visitados, los nodos no visitados y la trayectoria generada se puede observar en la Figura 23 y el entorno en Gazebo con el robot en la posición inicial se puede observar en 24.

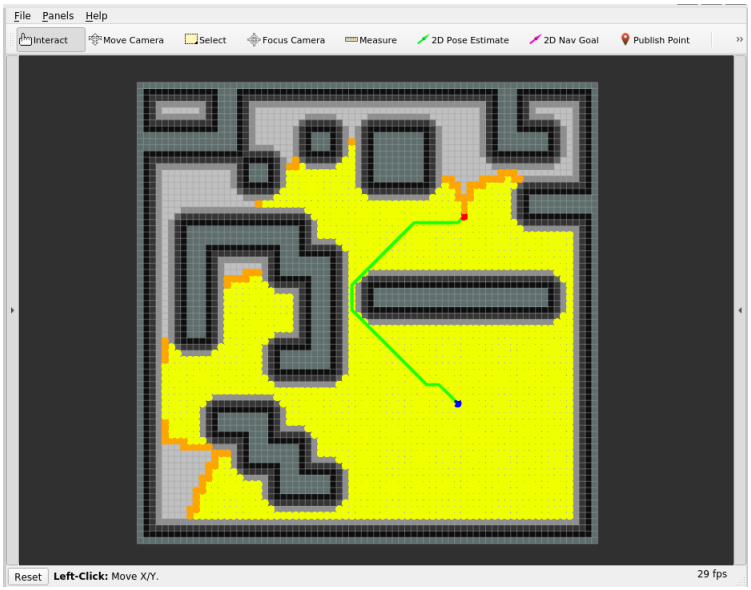


Figura 23: Espacio de búsqueda y la trayectoria generada por algoritmo de Dijkstra

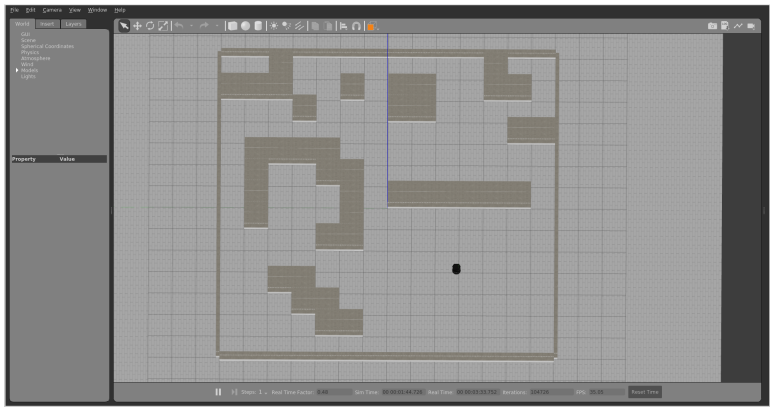
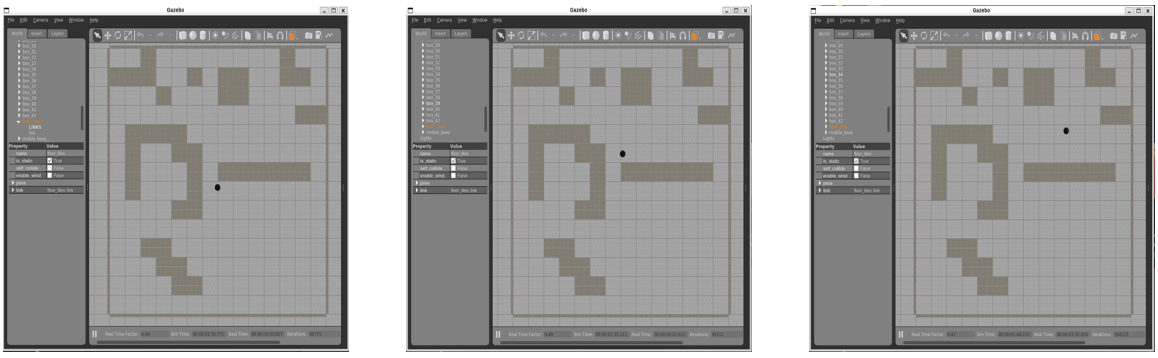


Figura 24: Espacio de simulación de Gazebo en la posición inicial

Para lograr el correcto funcionamiento de este algoritmo, inicialmente se define un archivo en donde se establece un modelo del mapa que se utilizara en formato YAML. YAML es un formato de serialización de datos, La principal ventaja de usar YAML es la legibilidad y la capacidad de escritura. YAML admite varios tipos de datos como casos, matrices, diccionarios, listas y escalares. Tiene un buen soporte para los lenguajes más populares como JavaScript, Python, Ruby, Java, etc.

Dentro de python, se toma este archivo y se genera una cuadricula de ocupación al cual es una discretización del espacio en donde el robot se esta moviendo. A partir de esto, se evalúa las posiciones alrededor de el punto en donde esta ubicado el robot, para ver cuáles son los nodos libres. Se repite este proceso para cada uno de los nodos ya evaluados obteniendo finalmente el camino más corto hacia el punto elegido. Finalmente, se publica el conjunto de puntos obtenidos a el tema de odometria del robot para generar que el robot se mueva siguiendo la trayectoria como se observa en la Figura 25.



(a) Movimiento # 1 (b) Movimiento # 2 (c) Movimiento # 3

Figura 25: Seguimiento de trayectoria de Dijkstra dentro del entorno de Gazebo

El objetivo de la implementación de este algoritmo dentro del entorno de ROS es poder establecer las bases para la implementación del algoritmo D* Lite y posteriormente utilizarlo en conjunto con el algoritmo PSO creando un robot capaz de navegar a través de un entorno totalmente desconocido y siendo capaz de converger a la meta de forma rápida y efectiva.

8.3. Algoritmo de D* Lite en ROS

A partir de lo desarrollado con el algoritmo de dijkstra se utilizó la misma idea de la discretización del espacio utilizando la herramienta de rviz la cual se puede ajustar para realizar una discretización más fina. A partir de esto se crearon dos listas, una de nodos abiertos y otra con nodos cerrados. Según lo establecido dentro de la lógica del algoritmo D* Lite se realizo una búsqueda entre los nodos pea obtener la ruta que presentará el menor costo para ser recorrido.

Se debe tomar en cuenta que en este caso, el valor de costo para cada nodo está dado por la suma del costo de llegar a cada nodo más la heurística desde el punto desde donde está el robot hacia el punto el cual está siendo evaluado. tomando esto en consideración se crea un vector el cual contiene los valores del costo del nodo y se priorizan los nodos según

cual sea el valor mínimo siguiente en la sucesión de nodos dentro de la lista definida. este comportamiento se puede observar en la Figura 26.

(a) Posición inicial (b) Búsqueda finalizada

Figura 26: Búsqueda de trayectoria utilizando algoritmo D* Lite

En la Figura 26 se pueden observar los nodos visitados de color amarillo, los nodos no visitados en color naranja y la trayectoria desde la posición inicial del robot hacia la posición indicada de color verde. Al obtener una trayectoria brindada se analiza cual es el nodo al cual pertenece el menor valor y se busca cual de los nodos padre tiene el menor valor y así sucesivamente hasta que se encuentra el nodo en donde se encuentra el robot. Este tipo de análisis nos brinda un conjunto de puntos los cuales están conectados entre sí para formar la trayectoria brindada. Este conjunto de puntos es enviado hacia el nodo de odometría del robot que se encarga de mover el robot hacia los puntos indicados dentro del conjunto para que el robot sea capaz de recorrer el mapa sin colisionar contra ningún objeto, como se observa en la Figura 27.

(a) Posición inicial (b) desplazamiento #1 (c) desplazamiento #2

(d) desplazamiento #3 (e) desplazamiento #4 (f) Posición final

Figura 27: Movimiento del robot siguiendo la trayectoria generada con Rviz

8.4. Diseño de algoritmo PSO dentro del entorno de ROS

Para el diseño del algoritmo PSO dentro de ROS se tuvo que generar un archivo .launch que fuera capaz de generar más de un robot dentro del entorno de simulación gazebo. Para esto se tomó el modelo urdf del robot Turtlebot el cual viene dentro de la instalación de ROS, en este archivo se describen todas las propiedades físicas del robot al igual que los nombres de los parámetros. Luego, se creó otro archivo .launch en el cual se crearon diversos grupos renombrando a cada grupo para indicar la denominación que tendría cada uno de los robots e indicando la posición inicia y el giro de cada uno de los robots. Ya teniendo el archivo este archivo con subgrupos se creo un código de python que se utilizaría como código general para cada uno de los robots.

Tomando esto en consideración se procedió a inicializar este código dentro del archivo .launch que describe las generalidades del robot. Cabe destacar que el modelo del Turtlebot que se utilizó no es un robot diferencial, mas bien utiliza el modelo de *roll*, *pitch*, *yaw*. En este caso, se modificaba la dimensión de "*yaw*" para realizar el giro en el eje z hasta que la parte frontal del robot estuviera apuntando hacia el punto deseado, luego de esto se le daba una velocidad de avance. Finalmente cuando el robot llega al punto deseado, se actualiza la función del PSO para poder obtener una nueva meta y repetir el proceso.

El código realizado obtiene la posición actual del robot y calcula el algoritmo PSO, luego verifica que el valor obtenido sea menor que el valor global, si lo es, crea un nuevo tema (*topic*) propio dentro del robot y procede a publicar en este tema la nueva mejor posición encontrada. Por otro lado, si el mejor global es mejor que el mejor local, publica en el mismo tema generado la mejor posición global que existía con anterioridad. El tema propio creado dentro del robot se llama "mejor global p" el cual es un mensaje personalizado que contiene el mejor valor global personal de cada robot y la mejor posición en x y en y que acompaña a dicho valor. Los temas que se utilizan se pueden observar en 28.

```
/robot1/camera/depth/camera_info
/robot1/camera/depth/image_raw
/robot1/camera/depth/points
/robot1/camera/parameter_descriptions
/robot1/camera/parameter_updates
/robot1/camera/rgb/camera_info
/robot1/camera/rgb/image_raw
/robot1/camera/rgb/image_raw/compressed
/robot1/camera/rgb/image_raw/compressed/parameter_descriptions
/robot1/camera/rgb/image_raw/compressed/parameter_updates
/robot1/camera/rgb/image_raw/compressedDepth
/robot1/camera/rgb/image_raw/compressedDepth/parameter_descriptions
/robot1/camera/rgb/image_raw/compressedDepth/parameter_updates
/robot1/camera/rgb/image_raw/theora
/robot1/camera/rgb/image_raw/theora/parameter_descriptions
/robot1/camera/rgb/image_raw/theora/parameter_updates
/robot1/cmd_vel
/robot1/joint_states
/robot1/mejor_global_p
/robot1/odom
```

Figura 28: Temas publicados por cada robot generado.

Por otro lado, se creó un código de python que crea un nodo global que toma todos los parámetros independientes de cada robot y calcula cuál de todos estos parámetros es el que posee el menor valor del mejor global. Ya teniendo el valor que representa la verdadera mejor posición global se procede a publicar este valor a un nodo global el cual es leído por todos los robots para sincronizar la mejor posición global de todo el enjambre de robots. Este tema se puede observar en la Figura 29.

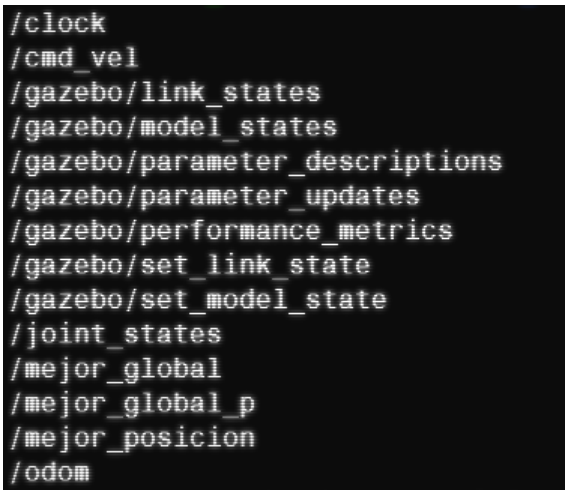


Figura 29: Temas generales publicados para la simulación de PSO en ROS.

8.4.1. Archivos .launch

Los archivos .launch están escritos en lenguaje XML, dentro del cual se pueden inicializar varios nodos, definir parámetros, nombrar diversos nodos, establecer parámetros de simulación, etc. La utilidad de crear archivos .launch es evitar inicializar cada nodo por separado y definir los parámetros individualmente. Además, al momento de inicializar un archivo .launch con el comando "roslaunch" se inicializa el roscore si no esta en funcionamiento.

Como podemos apreciar en 30, no deja de ser código basado en XML en el que utilizamos etiquetas .launch entre las cuales añadimos los nodos que deseamos ejecutar. Dentro de la etiqueta disponemos de tres opciones que son fundamentales para definir los nodos:

- ***Pkg:*** es el nombre del paquete en el cual se encuentra nuestro archivo o en donde estara el nodo.

- ***Name:*** es el nombre que le queremos dar al nodo.

- ***Type:*** define el nombre del archivo que se ejecutara.

```
<arg name="urdf_file" default="$(find xacro)/xacro '$(find turtlebot_description)/robots/$(arg base)_$(arg stacks)_$(arg 3d_sensor).urdf.xacro'" />
<param name="robot_description" command="$(arg urdf_file)" />

<group ns="robot1">
  <param name="tf_prefix" value="robot1_tf" />
  <include file="$(find uvg)/launch/one_robot.launch" >
    <arg name="x_pos" default="2"/>
    <arg name="y_pos" default="-2"/>
    <arg name="z_pos" default="0.0"/>
    <arg name="yaw" default="2.0"/>
    <arg name="robot_name"  value="Robot1" />
  </include>
</group>

<group ns="robot2">
  <param name="tf_prefix" value="robot2_tf" />
  <include file="$(find uvg)/launch/one_robot.launch" >
    <arg name="x_pos" default="-3"/>
    <arg name="y_pos" default="-3"/>
    <arg name="z_pos" default="0.0"/>
    <arg name="yaw" default="0.0"/>
    <arg name="robot_name"  value="Robot2" />
  </include>
</group>

<group ns="robot3">
  <param name="tf_prefix" value="robot3_tf" />
  <include file="$(find uvg)/launch/one_robot.launch" >
    <arg name="x_pos" default="-3"/>
    <arg name="y_pos" default="3"/>
    <arg name="z_pos" default="0.0"/>
    <arg name="yaw" default="-2.0"/>
    <arg name="robot_name"  value="Robot3" />
  </include>
</group>
```

Figura 30: Descripción del archivo .launch de la simulación del PSO

Como se puede observar en la Figura 30, dentro de esta se definen argumentos tales como la posición en el eje x, y y z. Estos argumentos son parte de la definición de la simulación dentro del entorno de gazebo. Dentro de la definición de cada uno de los grupos o robots se incluye otro archivo, este archivo contiene la definición del modelo urdf al igual que la posición en la cual se colocara cada robot como se observa en 31. Cabe destacar que los nodos no pueden tener el mismo nombre debido a que esto crea confusión dentro del sistema y puede causar resultados inesperados o un mal funcionamiento del código. Por otro lado, se puede observar que se inicializa el nodo del PSO dentro de este archivo, esto es para que cada robot ejecute de forma individual el código que contiene el algoritmo PSO.

```
<launch>
  <arg name= "robot_name"/>
  <arg name= "x_pos"/>
  <arg name= "y_pos"/>
  <arg name= "z_pos"/>
  <arg name= "yaw"/>

  <node name="urdf_robot_model_spawner" pkg="gazebo_ros" type="spawn_model"
    args="-param /robot_description
    -urdf
    -x $(arg x_pos)
    -y $(arg y_pos)
    -z $(arg z_pos)
    -Y $(arg yaw)
    -model $(arg robot_name)"
    respawn="false" output="screen"/>


  <node pkg="uvg" type="PSO.py" name="run_PSO" output="screen" />

  </launch>
```

Figura 31: Descripción general del robot

8.5. Diseño de robot diferencial

Para el diseño del robot diferencial se trato de tener un modelo similar al del robo e-puck utilizado para las simulaciones realzadas en Webots. Para esto se definieron Figuras básicas para la forma del robot como lo son circulos y esferas. Inicialmente se creo un archivo que define diversos colores que podrían utilizarse para diferenciar las partes individuales del robot. Luego, se definió el aspecto que tendría el robot, se inició definiendo la estructura central utilizando la etiqueta de inercia para establecer las propiedades del robot tales como la masa y la posición que tendría. Luego se utilizó una etiqueta de colisión para establecer cual seria la geometría del robot, que en este caso es un cilindro. Se definió una propiedad visual que es lo que nos permite ver al robot en los entornos de simulación como Gazebo o rviz. Luego de tener las bases para la estructura central, se definió la geometría de las ruedas, que en este caso, igualmente son cilindros, pero con rotación en el eje y desplazados hacia cada uno de los lados del robot. Por ultimo, se agrego una esfera que es la que sirve como soporte para la estructura, evitando que el robot gire o se arrastre.

Se definió un cilindro que esta colocado en la parte superior del robot, este cilindro tiene como función ser el sensor del robot. De igual manera de definieron etiquetas de colisión, visuales y de inercia. Finalmente, se definieron juntas para establecer los nodos padres y los nodos hijos para llegar a obtener la estructura deseada. Ya teniendo el robot visual, se creo un macro que define las propiedades del robot dentro del entorno de gazebo, esto se realizó para darle propiedades diferenciales al robot, siendo capaz de ser controlado de forma diferencial. El resultado final en ambos plataformas (gazebo y rviz) se puede observar en la Figura 33 y en la Figura 32 respectivamente.

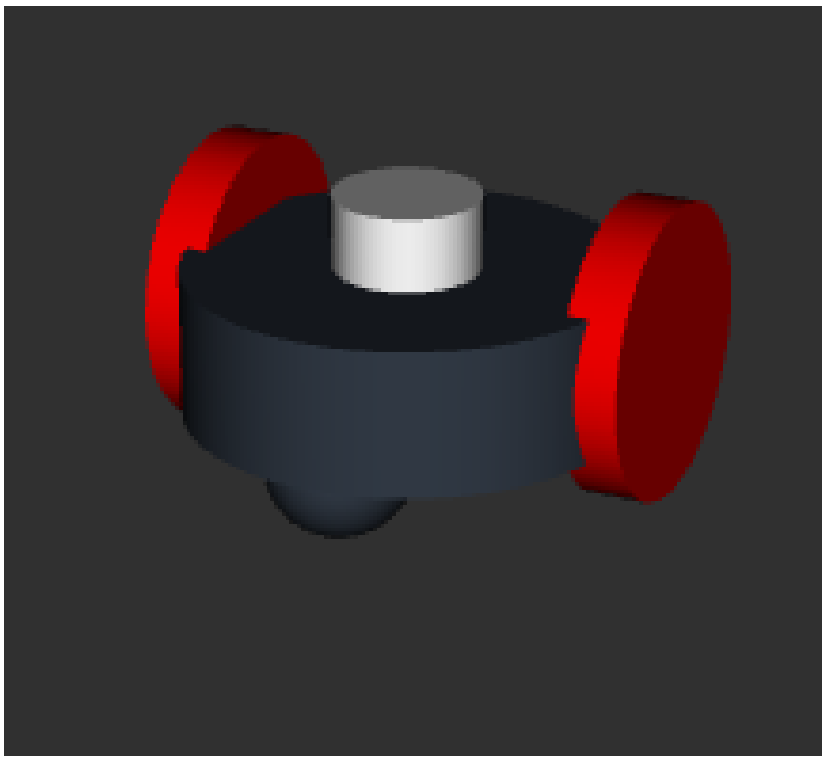


Figura 32: Robot diferencial dentro del entorno de rviz

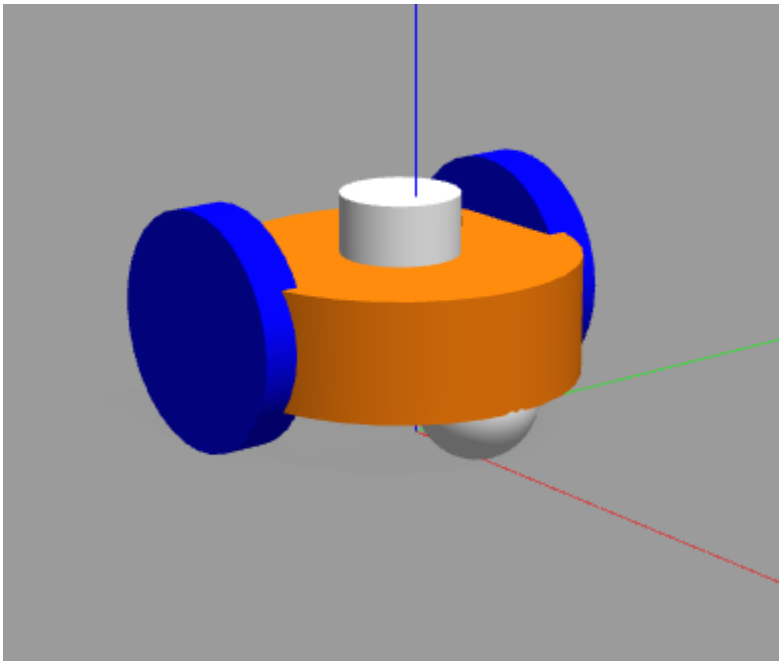
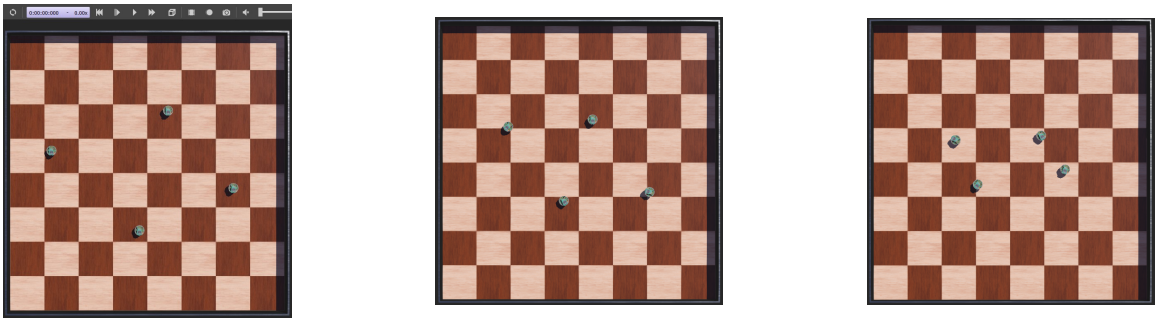


Figura 33: Robot diferencial dentro del entorno de Gazebo

CAPÍTULO 9

Resultados

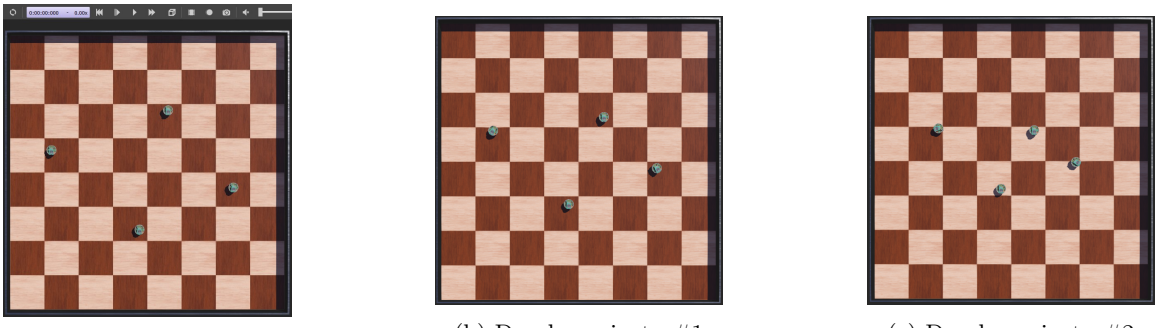
Se realizaron pruebas con algoritmos básicos de PSO y controladores PID, LQI y LQR para establecer el comportamiento que deberían de tener los robots y el tiempo que le toma al algoritmo converger en cada caso. Se pueden observar los tiempos de estas simulaciones en el apartado de anexos, desde la Figura 40 - 49. El algoritmo PSO es calculado individualmente por cada uno de los robots y se mantiene una comparación constante entre la posición actual del robot ($p_i(t)$) y la mejor posición que ha tenido el robot (p_i^{mejor}). Al mismo tiempo se compara la posición actual del robot con la mejor posición global (p_g^{mejor}). Estas comparaciones se realizan para indicarle al robot la dirección hacia donde debe moverse. El robot constantemente calcula el algoritmo de PSO de forma independiente, esto quiere decir que no existe un nodo supervisor que le indique a cada robot hacia donde moverse o que lleve el conteo del numero de partículas dentro del sistema. Esto nos presenta un problema debido a que los robots solo comparten la información de la mejor posición global y no de la posición individual de cada robot, causando así que puedan existir posibles colisiones al momento de llegar al mínimo global del sistema o que dos o más robots se crucen durante su trayectoria.



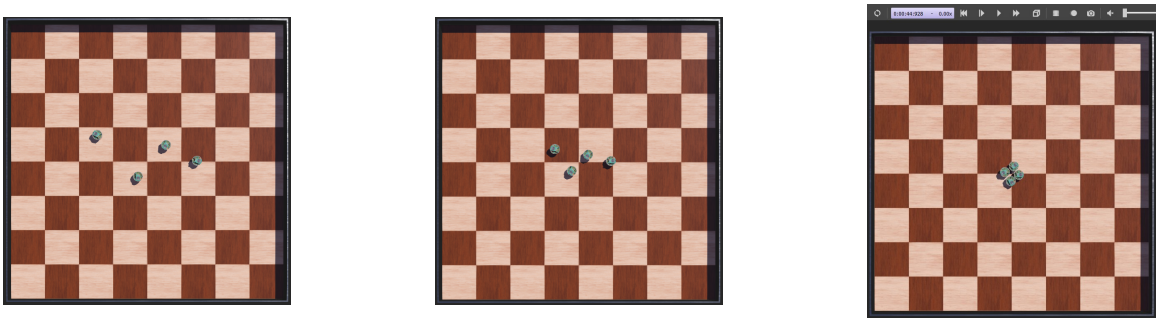
(a) Posición inicial (b) Desplazamiento #1 (c) Desplazamiento #2

(d) Desplazamiento #3 (e) Desplazamiento #4 (f) Posición final

Figura 34: Trayectoria del robot utilizando un controlador PID

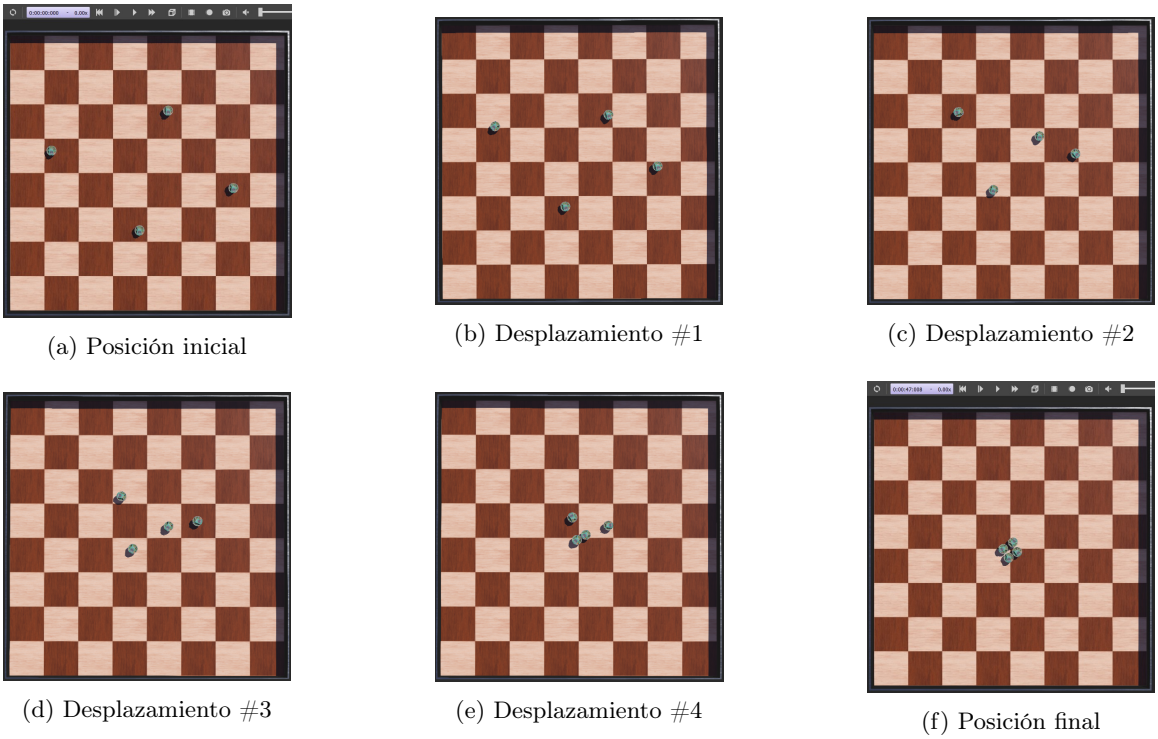


(a) Posición inicial (b) Desplazamiento #1 (c) Desplazamiento #2



(d) Desplazamiento #3 (e) Desplazamiento #4 (f) Posición final

Figura 35: Trayectoria del robot utilizando un controlador LQR



(a) Posición inicial (b) Desplazamiento #1 (c) Desplazamiento #2

(d) Desplazamiento #3 (e) Desplazamiento #4 (f) Posición final

Figura 36: Trayectoria del robot utilizando un controlador LQI

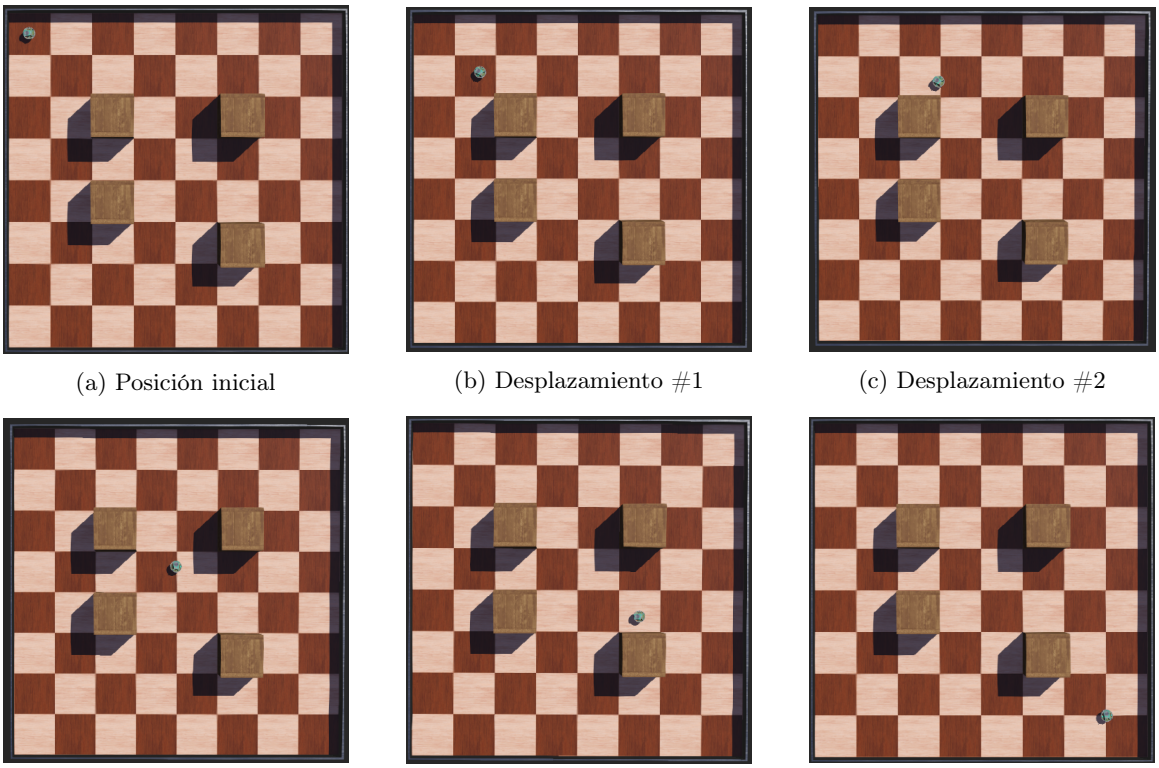
En el caso de las figuras mostradas con anterioridad, la función de costo utilizada fue la función de la esfera y su mínimo global es el punto $(0, 0)$. Como se puede observar en las figuras 34,35 y 36, todos los controladores son capaces de converger hacia el mínimo global de la función de costo. La única diferencia es el tiempo el cual les tomó a los robots en converger hacia este punto. Debido a esto se realizaron varias simulaciones para establecer los tiempos que le toma al sistema converger al mínimo global con cada uno de los controladores y estos valores se graficaron para obtener valores promedio de los tiempos. Obteniendo como resultado los siguientes valores (para mas detalle ver 40):

Controlador	**Tiempo promedio (s)**
PID	29.0632
LQR	10.4672
LQI	34.8332

Tomando estos datos en consideración, se puede observar que el controlador LQR es el controlador con el menor tiempo de convergencia. Por otro lado, se puede observar que los tiempos de el controlador PID y el controlador LQI son similares siendo la diferencia 5 segundos entre uno y el otro. Esto nos brinda los parámetros base para el desarrollo de las simulaciones con el algoritmo de evasión de obstáculos. Por lo tanto, se establece que el mejor tiempo de convergencia es brindado por el controlador LQR, seguido por el controlador PID y finalmente el controlador LQI.

9.1. Simulación de algoritmo D* Lite

En la siguiente secuencia de imágenes se logra observar el movimiento del robot desde una posición inicial en $(0.125, 0.125)$ y una posición final en $(0.75, 0.75)$.



(a) Posición inicial (b) Desplazamiento #1 (c) Desplazamiento #2

(d) Desplazamiento #3 (e) Desplazamiento #4 (f) Posición final

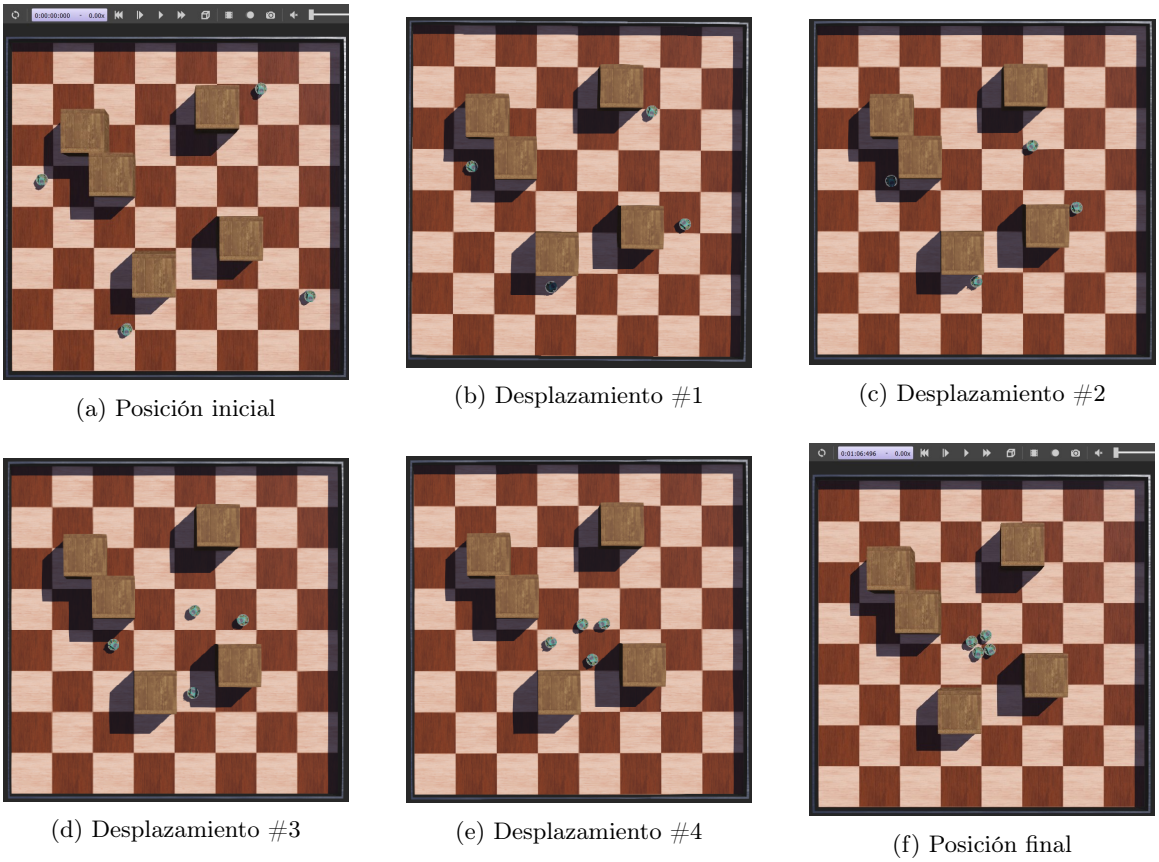
Figura 37: Trayectoria del robot utilizando el algoritmo D* Lite

Como se puede observar en la Figura 37, el robot es capaz de navegar a través de los obstáculos siguiendo una trayectoria definida. Al momento de implementar el algoritmo D* en conjunto con el algoritmo PSO nos encontramos con algunos problemas:

- El algoritmo PSO nos brinda el punto s_{meta} luego de ser calculado por la fórmula y es un punto arbitrario dentro del espacio, por lo cual era necesario discretizar el espacio con un valor de δ muy pequeño, aumentando de manera significativa el tamaño de la matriz que contenía los nodos, lo cual causaba que el calculo fuera más tardado debido a que el algoritmo debe iterar sobre todos los nodos presentes que se vayan acercando a la s_{meta} iniciando en s_{inicio}. Por lo tanto el tiempo el cual le tomaba al algoritmo realizar los cálculos y converger era mayor.

- Durante las pruebas, se pudo observar que el algoritmo D* tomaba a los otros robots como si fueran obstáculos, por lo cual, recalcular las trayectorias al estar cerca del punto de convergencia causaba una gran carga en el sistema y los robots se quedaban estáticos o realizando movimientos mínimos.

9.2. Simulación de algoritmo Braitenberg y PSO



(a) Posición inicial

(b) Desplazamiento #1

(c) Desplazamiento #2

(d) Desplazamiento #3

(e) Desplazamiento #4

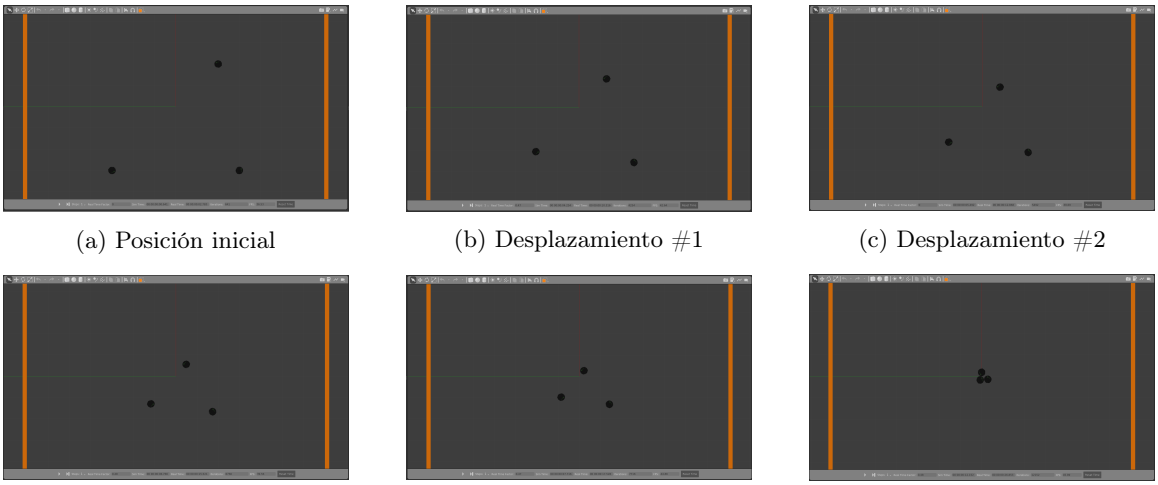
(f) Posición final

Figura 38: Trayectoria de los robots utilizando el algoritmo de braitenberg y algoritmo PSO

Como se puede observar en las Figura 38, el algoritmo PSO es capaz de converger al mínimo global siendo capaz de evadir los obstáculos dentro del entorno. El sistema es capaz de detectar cuando el obstáculo esta cerca y realiza los giros pertinentes para lograr evadir el obstáculo y dirigirse hacia el punto establecido por el algoritmo PSO. En el caso de las imágenes mostradas, se utiliza un controlador LQR debido a que es el controlador que obtuvo mejores resultados en lo que respecta a la velocidad de convergencia del algoritmo al igual que la forma en la cual converge.

9.3. Simulación de PSO en el entorno de ROS

Se realizaron simulaciones utilizando tres robots los cuales fueron colocados de forma aleatoria dentro del entorno con giros también aleatorios. Al momento de correr la simulación los robots eran capaces de converger hacia el mínimo global del sistema, en el caso particular de la Figura 39, la función que de utilizó fue la de la esfera la cual presenta un punto de convergencia en la posición $(0, 0)$.



(a) Posición inicial (b) Desplazamiento #1 (c) Desplazamiento #2

(d) Desplazamiento #3 (e) Desplazamiento #4 (f) Posición final

Figura 39: Trayectoria de los robots con algoritmo PSO en ROS

CAPÍTULO 10

Conclusiones

- Se logró implementar un algoritmo capaz de navegar a través de terrenos desconocidos y siguiendo un comportamiento brindado por el algoritmo de optimización de enjambre causando que el robot converja hacia el mínimo local de la función de costo.
- Se determinó que el sistema se comporta diferente al modelo idealizado dentro de las simulaciones del *Swarm Robotics Toolbox* debido a que las restricciones físicas agregan nuevos parámetros al sistema.
- Los parámetros obtenidos por el *PSO Tuner* generan una convergencia relativamente rápida hacia el mínimo local de la función de costo.
- Los parámetros brindados por el *PSO Tuner* son idealizados por lo cual el resultado puede no ser el esperado en relación a los resultados que se obtienen en las simulaciones.
- Aun variando el entorno, el sistema es capaz de converger hacia el mínimo global de la función de costo.
- El sistema es capaz de navegar dentro de diversos escenarios y ser capaz de adaptarse y evadir los obstáculos dentro del entorno.
- Se determinó la viabilidad de la utilización de ROS para simulaciones de algoritmos de búsqueda de trayectorias como lo es el algoritmo de Dijkstra y D*.
- Se determinó la viabilidad de la implementación de algoritmos PSO dentro del entorno de ROS para simulaciones multiagente.

CAPÍTULO 11

Recomendaciones

- Se recomienda la utilización de un nodo supervisor dentro de Webots, que sea capaz de indicarle a cada robot el punto al cual debe moverse y llevar un control de las trayectorias para evitar colisiones de los robots.
- Se recomienda la investigación de los diversos paquetes que ofrece ROS ya que esto podría brindar diversas mejoras al comportamiento de los robots al igual que una gran versatilidad y funcionalidad.
- Las mediciones realizadas por estos simuladores son estrictamente utilizando su propio sistema de coordenadas, por lo cual para una implementación física es necesario tomar en consideración estos ajustes.
- Se recomienda investigar métodos de lógica difusa para la implementación de un algoritmo de evasión de obstáculos más optimizado.

CAPÍTULO 12

Bibliografía

[1] J. Kennedy y R. Eberhart, «Particle swarm optimization,» en *Proceedings of ICNN'95 - International Conference on Neural Networks*, vol. 4, 1995, págs. 1942-1948. DOI: 10.1109/ICNN.1995.488968.

[2] E. Santizo, «Aprendizaje Reforzado y Aprendizaje Profundo en Aplicaciones de Robótica de Enjambre,» Tesis de licenciatura, Universidad del Valle de Guatemala, 2021.

[3] M. Castillo, «Diseñar e implementar una red de comunicación inalámbrica para la experimentación en robótica de enjambre,» Tesis doct., 2019.

[4] S. Ziadi, M. Njah y M. Chtourou, «PSO optimization of mobile robot trajectoriesin unknown environments,» *International Multi-Conference on Systems, Signals & Devices*, vol. 13, págs. 774-782, 2016.

[5] A. Aguilar, «Algoritmo Modificado de Optimización de Enjambre de Partículas (MPSO),» Tesis de licenciatura, Universidad del Valle de Guatemala, 2019.

[6] J. Cahueque, «Implementación de enjambre de robots en operaciones de búsqueda y rescate,» Tesis de licenciatura, Universidad del Valle de Guatemala, 2019.

[7] D. Álvarez, E. Toro y G. Ramón, «Algoritmo de Optimización cumulo de partículas aplicado en la solución de problemas de empaquetamiento óptimo bidimensional con y sin rotación,» *Scientia et Technica*, págs. 10-16, 2009.

[8] Z. Chi, G. Hai-bing, G. Liang y Z. Wan-guo, «Particle Swarm Optimization(PSO) Algorithm,» Tesis de mtría., Huazhong University of Science & Technology, China, 2010.

[9] A. Gonzáles, «Aplicaciones de técnicas de inteligencia artificial basadas en aprendizaje profundo (deep learning) al análisis y mejora de la eficiencia de procesos industriales,» Tesis de mtría., Universidad de Oviedo, España, feb. de 2018.

[10] The MathWorks, Inc., *¿Qué es una red neuronal?* https://la.mathworks.com/discovery/neural-network.html, 2021.

[11] E. Morales, «Aprendizaje por Refuerzo,» *Instituto Nacional de Astrofísica, Óptica y Electrónica*, págs. 3-12, 2011.

[12] K. Lobos, «Aplicaciones de aprendizaje reforzado en robótica móvil,» Tesis de licenciatura, Universidad de chile, 2018.

[13] N. Heess, J. Hunt, T. Lillicrap y D. Silver, «Memory-based control with recurrent neural networks,» *CoRR*, 2015.

[14] S. Koenig y M. Likhachev, «Fast Replanning for Navigation in Unknown Terrain,» *IEEE Transactions On Robotics And Automation*, vol. 20, págs. 1-10, 2002.

[15] Yujin y G. Xiaoxue, «Optimal Route Planning of Parking Lot Based on Dijkstra Algorithm,» en *2017 International Conference on Robots Intelligent System (ICRIS)*, 2017, págs. 221-224. DOI: 10.1109/ICRIS.2017.62.

[16] D. Fan y P. Shi, «Improvement of Dijkstra's algorithm and its application in route planning,» en *2010 Seventh International Conference on Fuzzy Systems and Knowledge Discovery*, vol. 4, 2010, págs. 1901-1904. DOI: 10.1109/FSKD.2010.5569452.

[17] V. Braitenberg, *Vehicles: Experiments in Synthetic Psychology*. Londres: MIT Press, 1986.

[18] A. Lorandi, G. Hermida, E. Durán y J. Hernández, «Controladores PID y Controladores Difusos,» *Revista de la Ingenieria Industrial*, vol. 5, págs. 2-6, 2011.

[19] Á. Fernández, «Modelado y control en el espacio de estados,» *Universidad politécnica de valencia*, págs. 12-15, 2016.

[20] S. Castaño, «Variables de Estado o Espacio de Estados,» *Control Automático Educación*, 2021.

[21] G. Beauchamp y R. Batista, «Aplicación de Técnicas de Control Óptimo a una plataforma estacionaria cuatrimotor,» *Revista de Ingeniería Electrónica, Automática y comunicaciones*, vol. XXXVII, págs. 34-49, 2016.

[22] V. Chalán, «Desarrollo de in controlador optimo LQR utilizando herramientas IOT para un sistema de presión constante controlado remotamente,» Tesis de mtría., Universidad politécnica salesiana, Quito, Ecuador, 2020.

[23] Stéphane Magnenat, *Photo of the e-puck mobile robot*, https://en.wikibooks.org/wiki/Cyberbotics%27_Robot_Curriculum/E-puck_and_Webots, 2008.

[24] C. Cianci, X. Raemy, J. Pugh y A. Martinoli, «Communication in a swarm of miniature robots: The e-puck as an educational tool for swarm robotics,» *International Workshop on Swarm Robotics*, págs. 103-115, 2006.

[25] A. Cubides, «Diseño e implementación de un sistema de interacción entre robots dirigidos por un usuario y objetos virtuales,» Tesis de licenciatura, Universidad Autónoma de occidente, 2016.

[26] J. Valencia, A. Montoya y L. Rios, «Modelo Cinemático de un robot móvil tipo diferencial y navegación a partir de la estimación odométrica,» *Scientia et Technica*, vol. 41, págs. 191-196, 2009, ISSN: 0122-1701.

[27] S. Surjanovic y D. Bingham, *Virtual Library of Simulation Experiments: Test Functions and Datasets*, Retrieved September 28, 2021, from http://www.sfu.ca/~ssurjano.

[28] Cyberbotics Ltd, *Webots*, https://cyberbotics.com, 2021.

[29] ——, *Webots*, https://cyberbotics.com/doc/reference/nodes-and-functions, 2021.

[30] The VRML Consortium Incorporated., *The Virtual Reality Modeling Language*, http://www.vrml.org/Specifications/VRML97/index.html, 1997.

[31] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler y A. Ng, «ROS: an open-source Robot Operating System,» *Computer Science Department, University of Southern California*, págs. 1-6, 2009.

[32] T. Foote, «tf: The transform library,» en *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*, ép. Open-Source Software workshop, abr. de 2013, págs. 1-6. DOI: 10.1109/TePRA.2013.6556373.

[33] Open Robotics, *Nodes*, http://wiki.ros.org/es/Nodes, 2021.

[34] Open Source, *Gazebo*, http://gazebosim.org, 2014.

[35] C. Fairchild y D. T. L. Harman, *ROS Robotics By Example*. Packt, 2016, ISBN: 9781782175193.

[36] R. Merino, «Creación del modelo URDF del robot Manfred,» Proyecto de fin de carrera, Universidad Carlos III de Madrid, 2020.

[37] N. Albergo, V. Rathi y J.-P. Ore, *Understanding Xacro Misunderstandings*, 2021. arXiv: 2109.09694 [cs.RO].

[38] J. Chavarría y J. Fallas, «El algoritmo PSO aplicado al problema de particionamiento de datos cuantitativos,» *Matemática, Educación e Internet*, vol. 19, n.º 1, págs. 1-13, 2019, ISSN: 1659-0643.

[39] Cyberbotics Ltd, *Webots*, https://cyberbotics.com/doc/guide/epuck, 2021.

[40] S. Sánchez, «Implementación y experimentación de herramientas para el diseño y desarrollo de comportamientos en robots tipo Turtlebot,» Trabajo Fin de Grado, Escuela politécnica superior, 2019.

CAPÍTULO 13

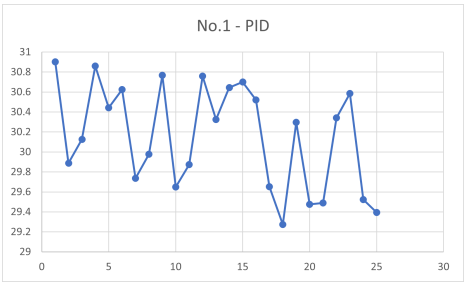
Anexos

13.1. Simulaciones de Webots sin obstáculos

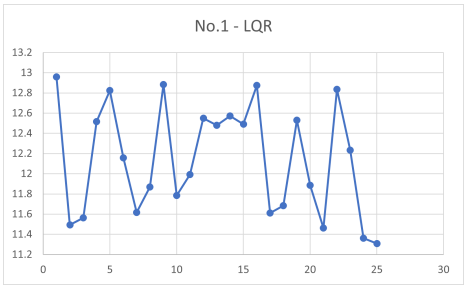
iteracion	**PID (s)**	**LQR (s)**	**LQI (s)**
1	30.24	12.16	34.496
2	29.38	11.488	35.264
3	28.42	9.76	35.712
4	29.1	10.144	34.08
5	30.12	9.76	35.168
6	28.23	10.368	34.816
7	28.61	9.824	33.024
8	28.42	9.92	33.536
9	28.64	10.944	34.72
10	29.472	10.304	37.516

Cuadro 3: Promedio de tiempos de cada controlador por conjunto.

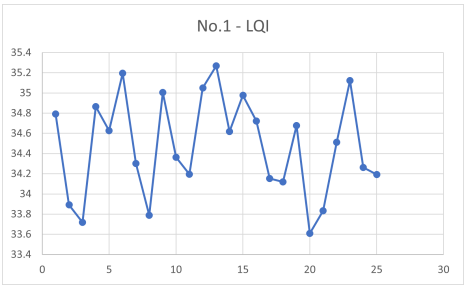
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

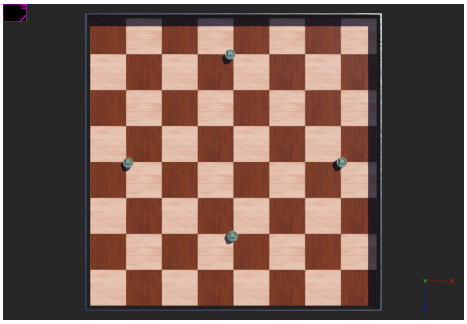


(c) Tiempos obtenidos con controlador LQR

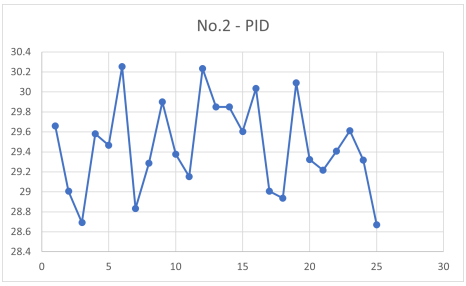


(d) Tiempos obtenidos con controlador LQI

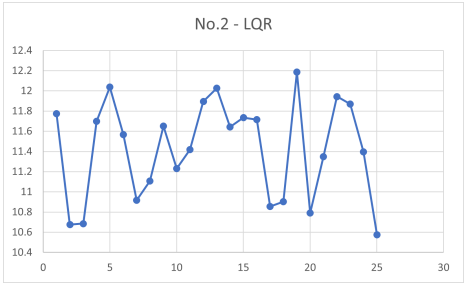
Figura 40: Primer conjunto de simulaciones dentro de Webots



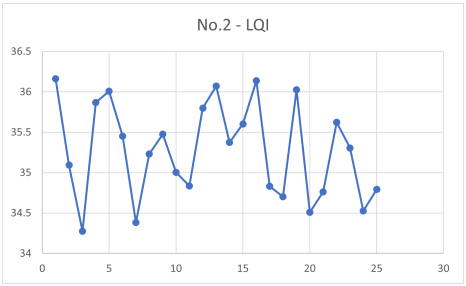
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

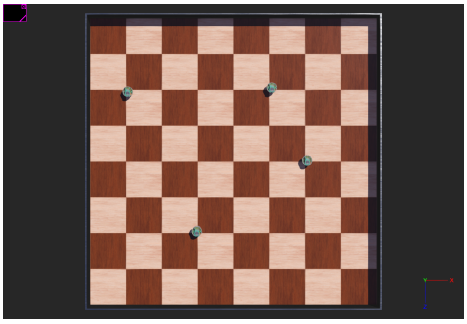


(c) Tiempos obtenidos con controlador LQR

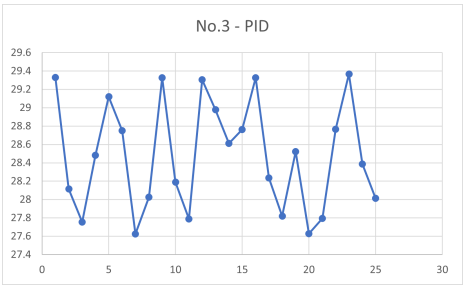


(d) Tiempos obtenidos con controlador LQI

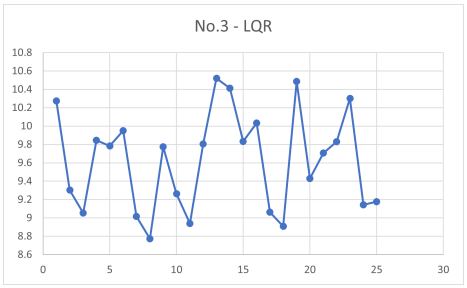
Figura 41: Segundo conjunto de simulaciones dentro de Webots



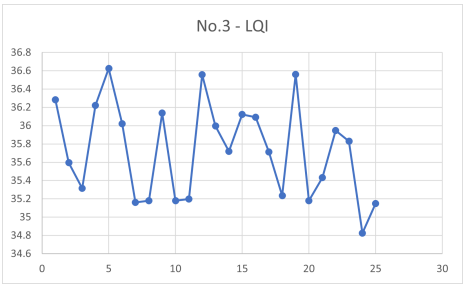
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

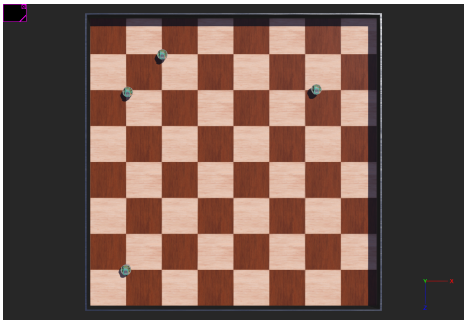


(c) Tiempos obtenidos con controlador LQR

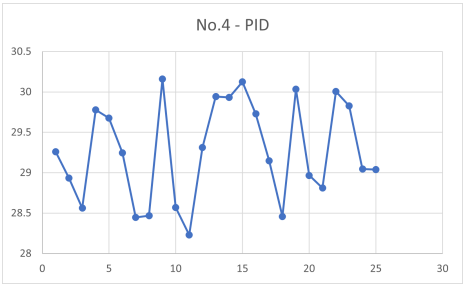


(d) Tiempos obtenidos con controlador LQI

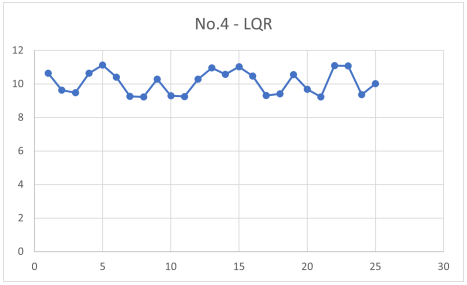
Figura 42: Tercer conjunto de simulaciones dentro de Webots



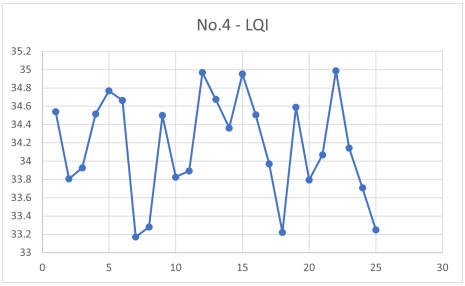
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID



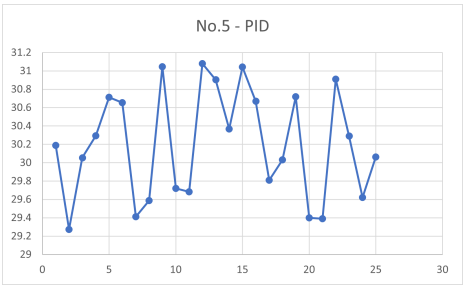
(c) Tiempos obtenidos con controlador LQR



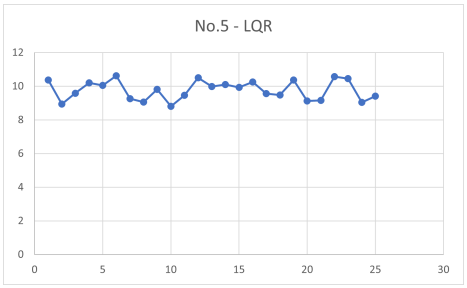
(d) Tiempos obtenidos con controlador LQI

Figura 43: Cuarto conjunto de simulaciones dentro de Webots

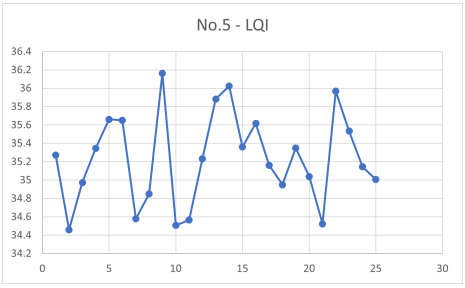
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

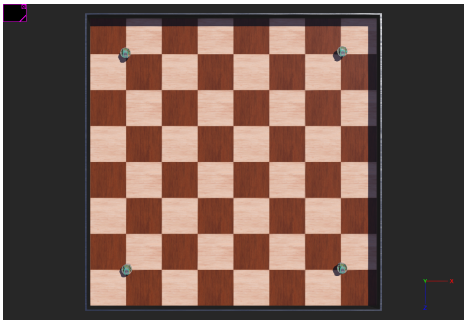


(c) Tiempos obtenidos con controlador LQR

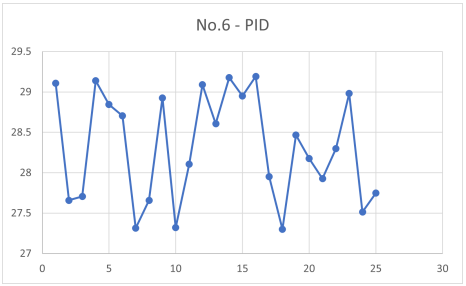


(d) Tiempos obtenidos con controlador LQI

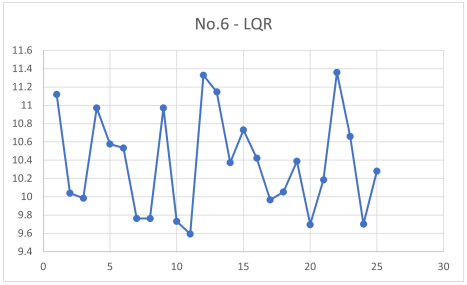
Figura 44: Quinto conjunto de simulaciones dentro de Webots



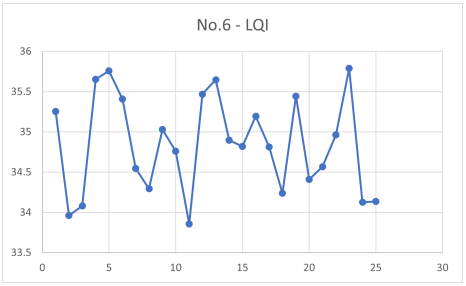
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

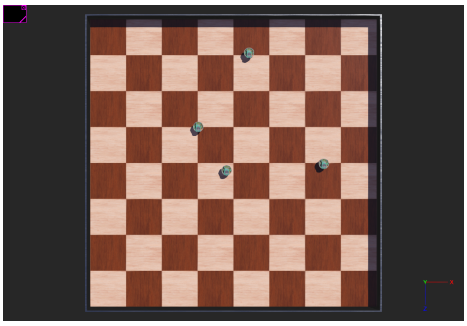


(c) Tiempos obtenidos con controlador LQR

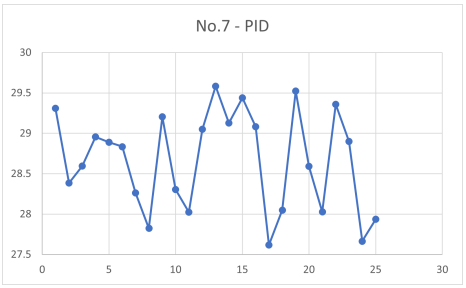


(d) Tiempos obtenidos con controlador LQI

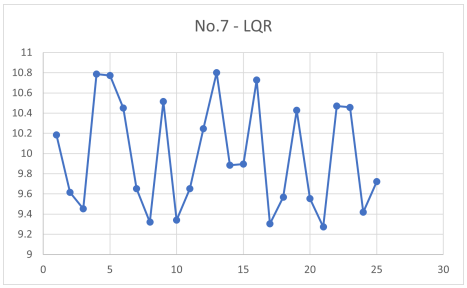
Figura 45: Sexto conjunto de simulaciones dentro de Webots



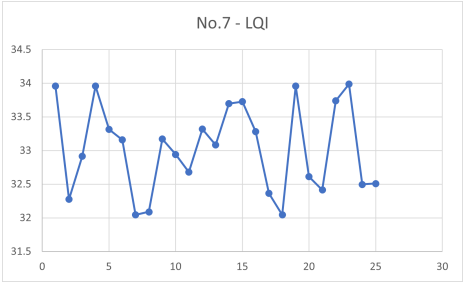
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

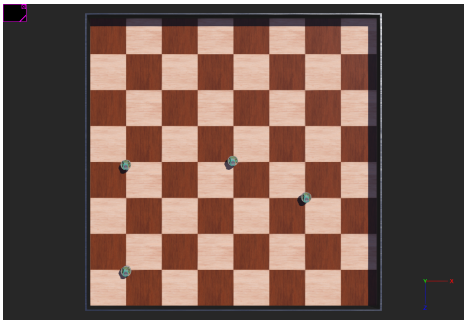


(c) Tiempos obtenidos con controlador LQR

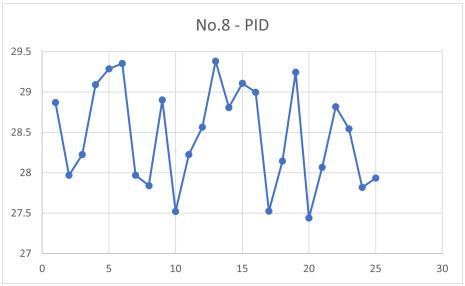


(d) Tiempos obtenidos con controlador LQI

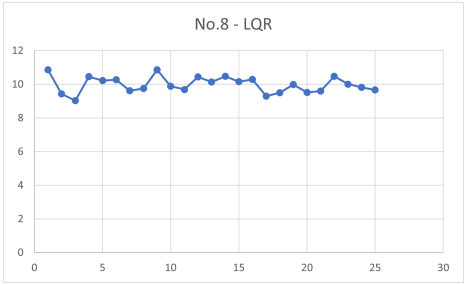
Figura 46: Séptimo conjunto de simulaciones dentro de Webots



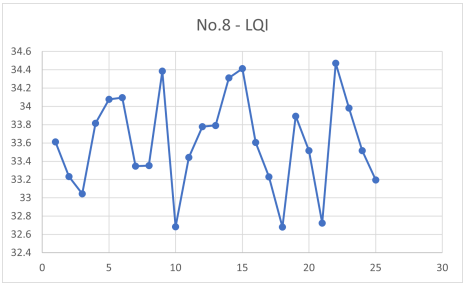
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID

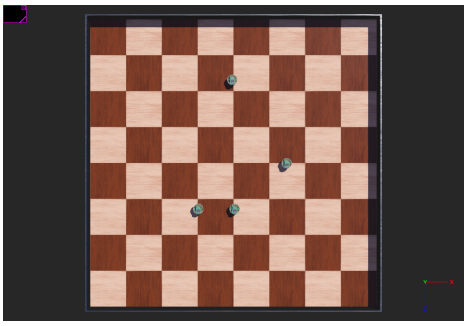


(c) Tiempos obtenidos con controlador LQR

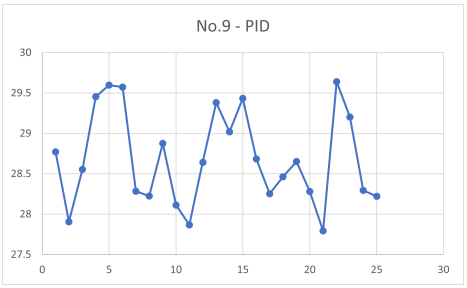


(d) Tiempos obtenidos con controlador LQI

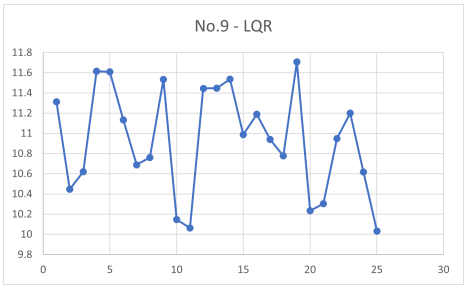
Figura 47: Octavo conjunto de simulaciones dentro de Webots



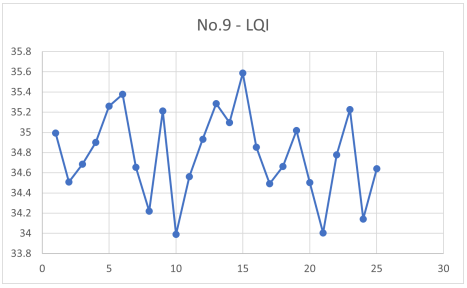
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID



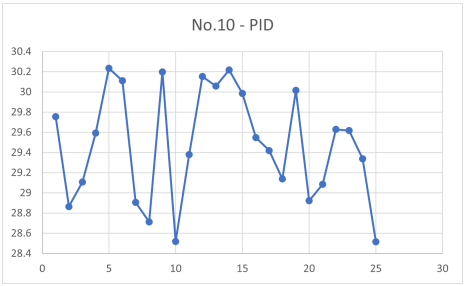
(c) Tiempos obtenidos con controlador LQR



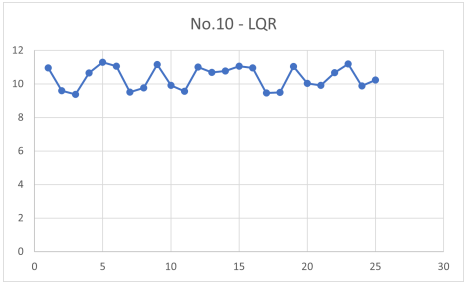
(d) Tiempos obtenidos con controlador LQI

Figura 48: Noveno conjunto de simulaciones dentro de Webots

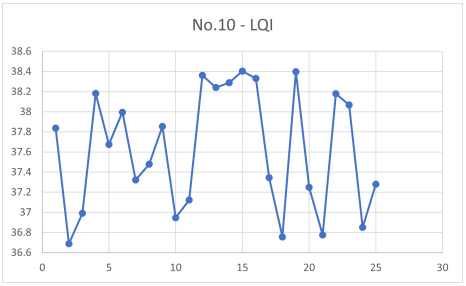
(a) Posición inicial



(b) Tiempos obtenidos con controlador PID



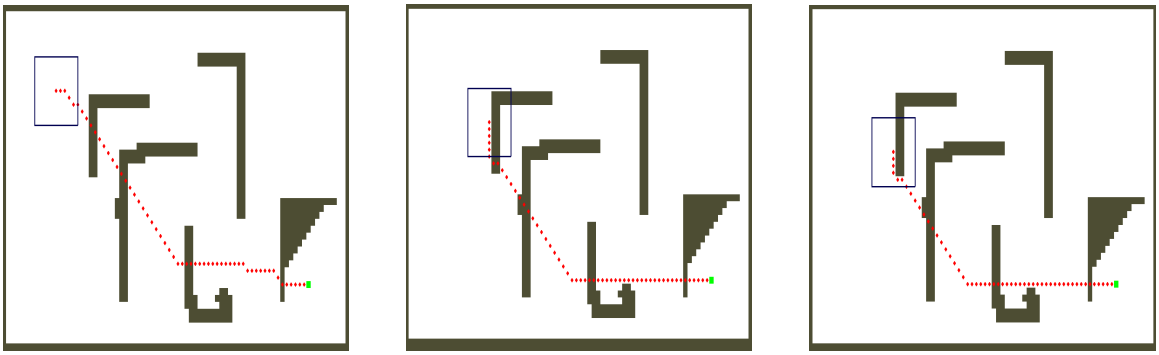
(c) Tiempos obtenidos con controlador LQR



(d) Tiempos obtenidos con controlador LQI

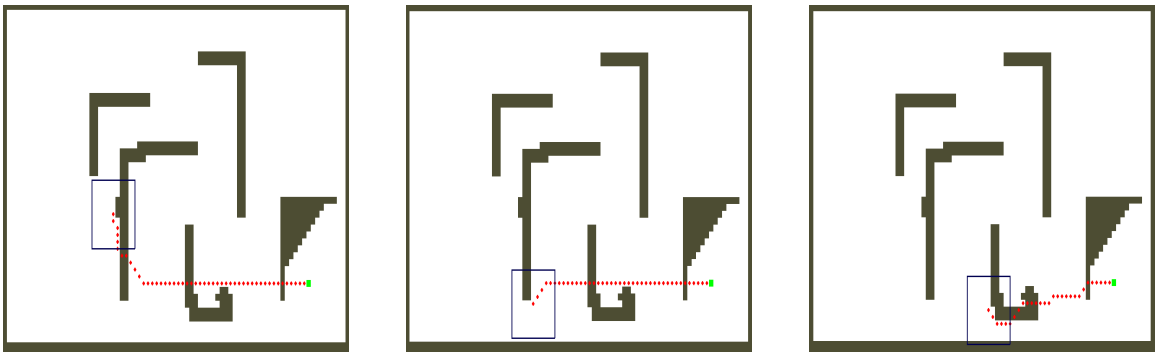
Figura 49: Décimo conjunto de simulaciones dentro de Webots

13.2. Simulaciones con algoritmo D* lite



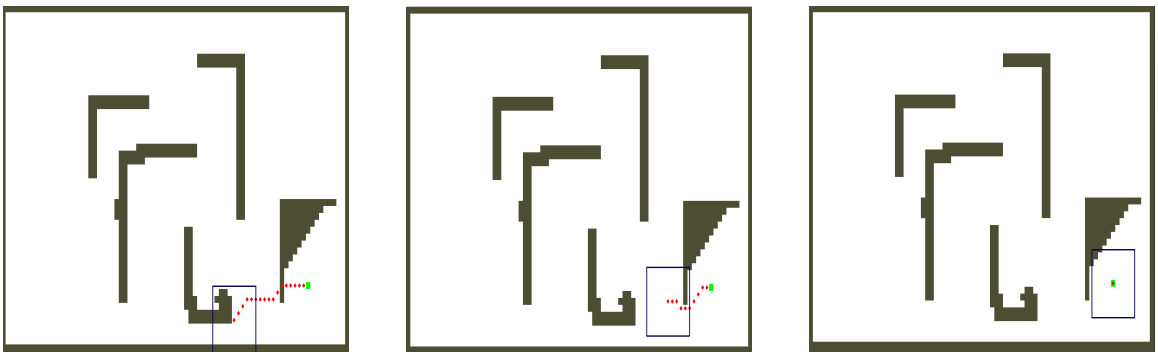
(a) Posición Inicial (b) Primer cambio (c) Segundo cambio

Figura 50: Primer conjunto de trayectorias generadas por el algoritmo D* Lite



(a) Tercer cambio (b) Cuarto cambio (c) Quinto cambio

Figura 51: Segundo conjunto de trayectorias generadas por el algoritmo D* Lite



(a) Sexto cambio (b) Séptimo cambio (c) Punto final

Figura 52: Tercer conjunto de trayectorias generadas por el algoritmo D* Lite