UNIVERSIDAD DEL AZUAY

FACULTAD DE CIENCIA Y TECNOLOGÍA

ESCUELA DE INGENIERÍA ELECTRÓNICA

Diseño e implementación de un sistema de generación de trayectoria para el control de un robot móvil, utilizando Inteligencia Artificial

Trabajo de graduación previo a la obtención del título de:
INGENIERA ELECTRONICA

Autor:
Andrea Estefania Rodas Córdova

Director:
MSc. Francisco Salgado

CUENCA-ECUADOR
2021

“Diseño e implementación de un sistema de generación de trayectoria para el control de un robot móvil, utilizando Inteligencia Artificial”

RESUMEN

En este trabajo se presenta un sistema autónomo basado en ROS para un robot móvil tipo Skid Steer capaz de generar mapas 2D de un ambiente desconocido y luego navegar dentro de él de forma autónoma. El sistema se implementó con una computadora y una Raspberry Pi 3 que controla el robot, ambas máquinas se comunican de manera inalámbrica a través de la red creada por ROS que permitía la comunicación de nodos entre diferentes máquinas y que proporciona varias bibliotecas y librerías. Además, se utilizó un sensor LIDAR para las lecturas de datos que interfieren en cada parte del sistema. Se implementó el algoritmo SLAM para la creación de mapas y se propusieron dos métodos para la navegación. El robot fue capaz de navegar desde una posición inicial hasta una posicion de meta establecida de forma autónoma y evitando obstáculos en el camino Por motivos de experimentación, también se probó una técnica de Aprendizaje Reforzado.

Palabras Clave: LIDAR, Robot Movil, Raspberry Pi, RL, ROS, Skid Steer.

MSc. Francisco Salgado
Director de trabajo de titulación

Phd. Daniel Iturralde
Coordinador de escuela de
Ingeniería Electrónica

Andrea Estefania Rodas Córdova

Autora

“Design and implementation of path planning system for the control of a mobile robot, using Artificial Intelligence”

ABSTRACT

This paper introduced a ROS based autonomous system for a Skid Steer mobile robot capable of generating 2D maps of an unknown environment and then navigate within it autonomously. The system was implemented with a computer and a Raspberry Pi 3 that controls the robot, both machines communicate wirelessly through the network created by ROS that allowed the communication of nodes between different machines and that provides several tools and libraries. Also, a LIDAR sensor was used for all data readings for each part of the system. The SLAM algorithm was implemented for the creation of the map and two stages were proposed for navigation. The robot was capable to navigate from an initial pose to a settled goal autonomously and avoiding obstacles on the way. For experimentation, a Reinforcement Learning technique was tested too.

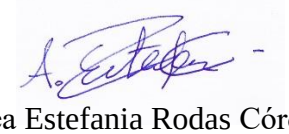
Keywords: LIDAR, Mobile Robot, Raspberry Pi, RL, ROS. Skid Steer.

MSc. Francisco Salgado
Director de trabajo de titulación

Phd. Daniel Iturralde
Coordinador de escuela de
Ingeniería Electrónica

Andrea Estefania Rodas Córdova
Autora

Translated by:



Andrea Estefania Rodas Córdova

Diseño e Implementación de un Sistema de Generación de Trayectoria para el Control de un Robot Móvil, Utilizando Inteligencia Artificial

Andrea Estefania Rodas Córdova
Universidad del Azuay
Cuenca, Ecuador
estrc@es.uazuay.edu.ec

***Abstract*-- This paper introduced a ROS based autonomous system for a Skid Steer mobile robot capable of generating 2D maps of an unknown environment and then navigate within it autonomously. The system was implemented with a computer and a Raspberry Pi 3 that controls the robot, both machines communicate wirelessly through the network created by ROS that allowed the communication of nodes between different machines and that provides several tools and libraries. Also, a LIDAR sensor was used for all data readings for each part of the system. The SLAM algorithm was implemented for the creation of the map and two stages were proposed for navigation. The robot was capable to navigate from an initial pose to a settled goal autonomously and avoiding obstacles on the way. For experimentation, a Reinforcement Learning technique was tested too.**

Keywords— LIDAR, Mobile Robot, Raspberry Pi, RL, ROS. Skid Steer.

I. INTRODUCCIÓN

El área de investigación de la robótica móvil ha tenido grandes avances en distintas áreas como la industria, transporte, salud, entretenimiento o como asistentes para tareas diarias [1]. Siguiendo esta tendencia, este trabajo presenta la implementación de un robot móvil autónomo basado en el *framework* ROS (*Robot Operating System*) para proyectos de robótica. ROS contiene herramientas, librerías y paquetes de código abierto disponibles para varios modelos de robots y sensores que soporta varios lenguajes de programación e incluye entornos de simulación totalmente personalizables para las características del trabajo. Un ejemplo de su aplicación está en [2] que presenta un robot de servicio manipulador basado en ROS.

Un proyecto de simulación en el área de la agricultura industrial se presenta en [3] que emplea un sensor LIDAR (*Light Detection and Ranging*) junto con ROS y el simulador de entornos Gazebo para un robot móvil con la labor de medir el volumen de las plantas y la altura de los árboles.

Para este trabajo el sensor principal que interviene en cada parte de la implementación es el LDS-01, un sensor de tipo LIDAR a 360° con el que se puede detectar objetos hasta una distancia de 350 cm, interpretar el entorno en dos dimensiones (2D) y obtener datos odométricos.

El robot utilizado es el Adeept AWR tipo *Skid Steer* de cuatro ruedas que tiene una Rasberry Pi 3 como controlador principal, encargada de procesar toda la información y dar órdenes a los motores. Un computador con sistema operativo Linux es el master del sistema y la comunicación inalámbrica es por medio de la red *Computation Graph Level* [4] por parte de ROS, que permite la comunicación de nodos y mensajes entre diferentes máquinas con una infraestructura de *Publisher-Subscriber*.

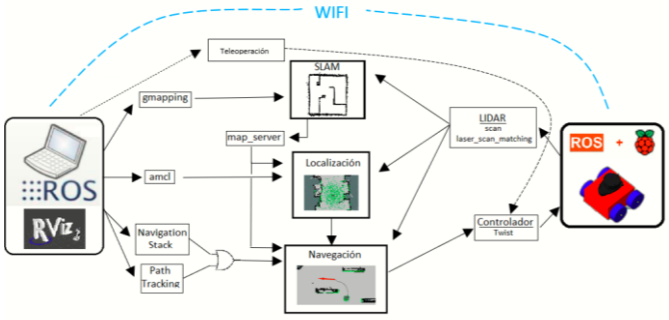


Fig. 1 Resumen del sistema para un robot móvil autónomo.

La fig. 1 presenta un resumen de todo el sistema y el trabajo se divide de la siguiente manera: La primera parte es sobre el controlador del robot, la recepción de mensajes de velocidad y la conversión de estos a valores de *Duty Cycle* (Ciclo de trabajo), la segunda parte explica lo que es SLAM (*Simultaneous Localization and Mapping*) y su implementación con el algoritmo *Gmapping* , la tercera parte trata sobre la localización con el algoritmo AMCL, la cuarta parte presenta dos métodos de navegación autónoma, uno que utiliza el paquete de *Navigation Stack* con los algoritmos Dijkstra y A*[1], por otro lado se utiliza un seguimiento de trayectoria como el segundo método de implementación. La última parte implementa *Reinforcement Learning (*Aprendizaje Reforzado*)*, una técnica de Inteligencia Artificial avanzada que utiliza el algoritmo *Deep Q-Learning* para la navegación autónoma del robot en un entorno simulado.

II. METODOLOGÍA

A. Controlador

1) Cálculo de la velocidad de las ruedas

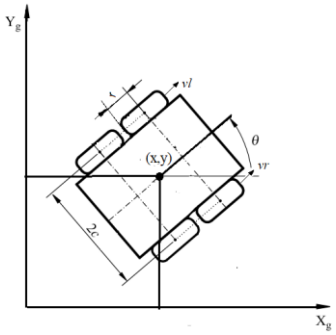


Fig. 2 Sistema de coordenadas y marco de referencia del robot.

[1] Algoritmos de búsqueda de trayectoria.

El robot utilizado cumple con las características de un modelo cinemático *Skid Steer* de direccionamiento diferencial [5] cuyo modelo se puede expresar con el conjunto de ecuaciones de transición (1):

$$\begin{cases} \dot{x} = \frac{r}{2}(vr + vl)cos\theta \\ \dot{y} = \frac{r}{2}(vr + vl)sin\theta \\ \dot{\theta} = \frac{r}{2c}(vr - vl) \end{cases} \quad (1)$$

Como se representa en Fig. 2 donde r corresponde al radio de las ruedas, 2c es la distancia entre las ruedas y vr, vl son las velocidades angulares del lado derecho e izquierdo respectivamente, el controlador interpreta a las dos ruedas de cada lado como una sola y recibirán el mismo valor de velocidad.

Los datos recibidos por el controlador son mensajes del tipo Twist parte del paquete geometry_msgs [4] de ROS cuya función es describir la geometría y cinemática del robot. El mensaje Twist puede codificar el movimiento en seis grados de libertad, pero para un robot diferencial solo se utilizan dos, *linear.x* que corresponde a la velocidad lineal (v) en metros por segundo y *angular. z* que corresponde a la velocidad angular (w) en radianes por segundo. Para simplificar los cálculos se utilizan estas variables como entradas del sistema y el modelo cinemático de un monociclo simple [5] que tiene el conjunto de ecuaciones de transición (2):

$$\begin{cases} \dot{x} = vcos\theta \\ \dot{y} = vsin\theta \\ \dot{\theta} = w \end{cases} \quad (2)$$

Igualando ambos modelos se obtiene las ecuaciones (3) y (4) de velocidad para las ruedas derecha e izquierda [6], [7]:

$$vr = \frac{2v + w(2c)}{2r} \quad (3)$$

$$vl = \frac{2v - w(2c)}{2r} \quad (4)$$

Si $vr = vl$, el robot se desplazará en una sola dirección y a la velocidad comandada. Para la rotación, las ruedas de la derecha tienen que girar en sentido contrario a las ruedas del lado izquierdo, siguiendo la premisa de que una velocidad angular positiva representa una rotación en contra de las manecillas del reloj y una velocidad angular negativa representa una rotación en el sentido de las manecillas del reloj.

2) Conversión entre velocidad y Duty Cycle

Fig. 3 Robot Adeept AWR.

En la parte del hardware el robot Adeept AWR funciona con una placa Raspberry Pi 3B y un *Motor Hat* que incorpora un chip L298N para el control de los puertos GPIO de la placa a los que se conectan los cuatro motorreductores de las ruedas tal como se indica en la Fig. 3.

Este controlador también se encarga de distribuir la alimentación proveniente de las dos baterías 18650 de 3.7v cada una para energizar la placa y los motores según el nivel de la señal PWM (Pulse Width Modulation).

En la parte del software el controlador se encarga de convertir las velocidades *vr* y *vl* que están en radianes por segundo a un valor de *Duty Cycle entre* 0 y 100. Dado que el robot no tiene implementado ningún sensor de velocidad, no se puede obtener una retroalimentación y el controlador es de malla abierta tomando ciertas consideraciones respecto a la funcionalidad de los motores para mantener siempre una velocidad adecuada. Para su funcionamiento óptimo el valor de PWM mínimo es de 20% y el máximo en 80%; el controlador estará encargado de recibir los valores de velocidad y establecer un nivel correspondiente de PWM entre estos valores para que el robot se desplace correctamente.

B. Mapeo

1) Sensor LIDAR

Fig. 4 Sensor LIDAR LDS-0.

El sensor principal utilizado en este trabajo es el LDS-01 (Fig.4), es un sensor laser de distancia 2D catalogado dentro de los sensores LIDAR, capaz de escanear los 360° de su entorno y alcanzar una distancia de detección de objetos desde 12 cm hasta 350 cm a una taza de muestreo de 1.8 kHz [8]. Funciona emitiendo pulsos de luz laser y calculando la distancia a través del retardo entre la emisión y el rebote en un objeto hacia el sensor. Combinando y procesando la información en un plano 2D se puede generar una nube de puntos del entorno alrededor del robot y luego la creación del mapa [9].

Para utilizar el sensor con ROS, los fabricantes incluyen el paquete HLS-LFCD2 con el software necesario para el funcionamiento.

2) Odometría

Para conseguir datos odométricos se utilizó el sensor LIDAR conectado a la Rasberry Pi junto con el paquete *Laser Scan Matcher* que implementa el algoritmo de Andrea Censi [10] conocido como *Canonical Scan Matcher* o CSM. El algoritmo realiza una variación del punto más cercano iterativo (ICP) utilizando una métrica de punto a línea optimizada con el fin de encontrar puntos comunes entre dos censados del láser consecutivos. El paquete además es parte de ROS y proporciona un *topic* llamado */Pose2D* con las coordenadas x, y, θ

representando la ubicación al mismo tiempo que proporciona el *topic* /*odom* esencial para el funcionamiento necesario de los demás nodos de ROS que se utilizaron en este trabajo.

3) SLAM – Gmapping

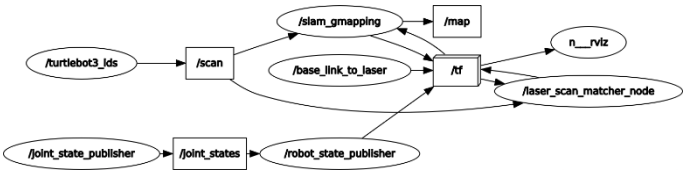


Fig. 5 Estructura de nodos en ejecución para Gmapping.

SLAM (Simultaneous Localization and Mapping) se describe como la solución al problema de mapeo y localización simultánea, mediante un proceso algorítmico que crea y actualiza un mapa de un entorno desconocido mientras guarda información sobre su localización [11].
En el caso de ROS, presenta varios algoritmos y técnicas para crear mapas 2D entre los que se podrían nombrar a *Gmapping, Hector SLAM y Cartographer* como los más conocidos.

Gmapping es un algoritmo perteneciente al proyecto OpenSLAM que implementa el filtro de partículas Rao-Blackwelized [12] para construir mapas a partir de datos provenientes de un sensor LIDAR. Esta técnica minimiza el número de partículas haciendo un muestreo solo cuando existe movimiento del robot y conservando la información sobre la última lectura, logrando reducir la incertidumbre de la posición del robot en las próximas muestras del filtro [13].
El nodo *slam_gmapping* se subscribe a los *topics /scan*, que proviene del sensor, y a /*tf* que representa las transformaciones de los marcos de coordenadas del robot incluyendo la información odométrica que proviene de *laser_scan_matching.*
La Fig.5 es una representación de toda la estructura de nodos y *topics* activos que interfieran con Gmapping.
El mapa se actualiza periódicamente y se publica en el *topic /map*, luego de terminar con el proceso de SLAM se puede guardar como un archivo pgm y yaml.

C. Localización en el mapa

Una vez construido el mapa del entorno, el robot necesita localizarse dentro del mismo antes de comenzar la navegación, para este problema se aplica el algoritmo de Localización de Monte Carlo [11], una técnica probabilística que emplea los filtros de partículas para predecir la posición y orientación del robot dentro del mapa. Cada posible ubicación tiene un peso de mayor o menor probabilidad dependiendo de las lecturas del sensor LIDAR [14].

1) AMCL (Adaptive Monte Carlo Localization)

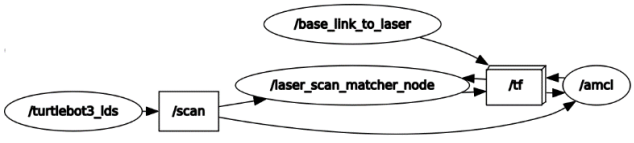


Fig. 6 Estructura de nodos en ejecución para AMCL.

AMCL es una variación del algoritmo de Localización de Monte Carlo que adapta el número de muestras durante la ejecución, es parte de los paquetes de localización proporcionado por ROS para trabajar en mapas 2D.
Para que el nodo *amcl* funcione se debe suscribir a los *topics /scan* (lecturas del sensor), */tf* (transformación de marcos de coordenadas), /*map* (lectura de la información del mapa) e /*initialpose* (posición inicial del robot en el mapa).
El resultado del procesamiento del algoritmo se publica en el *topic /amcl_pose* que presenta la posición y orientación estimada del robot relativa al mapa a ciertos intervalos de tiempo [15].
La estructura de nodos y *topics* para la localización está en la Fig. 6.

D. Navegación

La navegación es la convergencia de todo el sistema que conforma el robot, en esta parte interfiere el controlador, la información del sensor LIDAR, el mapa, y la localización para que el robot sea capaz de desplazarse de forma autónoma. Para la navegación se probaron dos métodos, *Navigation Stack* de ROS y un seguidor de trayectoria.

1) Navigation Stack

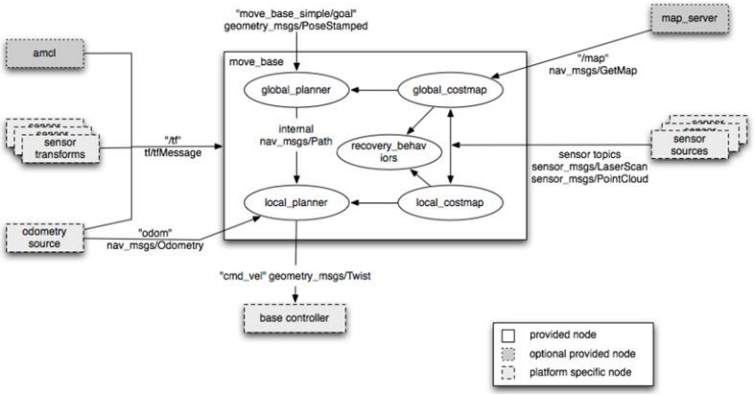


Fig. 7 Navigation Stack.

El diagrama de la Fig. 7 presenta los paquetes que conforman e interfieren con *Navigation Stack* [4]. El paquete *move_base* provee un nodo encargado de procesar la información sensorial del robot, luego planificar una trayectoria dada una meta o *goal* y por último enviar los mensajes de velocidad adecuados para el desplazamiento, funciona junto con la herramienta gráfica Rviz para tener una interfaz más sencilla al momento de manejar todos los nodos involucrados al mismo tiempo que se visualiza la información.
A continuación, se describen las partes más importantes [14], [16]:

- Goal: La posición final a la que debe llegar el robot. Es un mensaje con la información de las coordenadas de un punto en el mapa y su orientación.
- Global Planner: El proceso de planificación de trayectoria desde la posición del robot hasta el punto de meta (*goal*). Existen tres opciones: *carrot_panner, navfn*

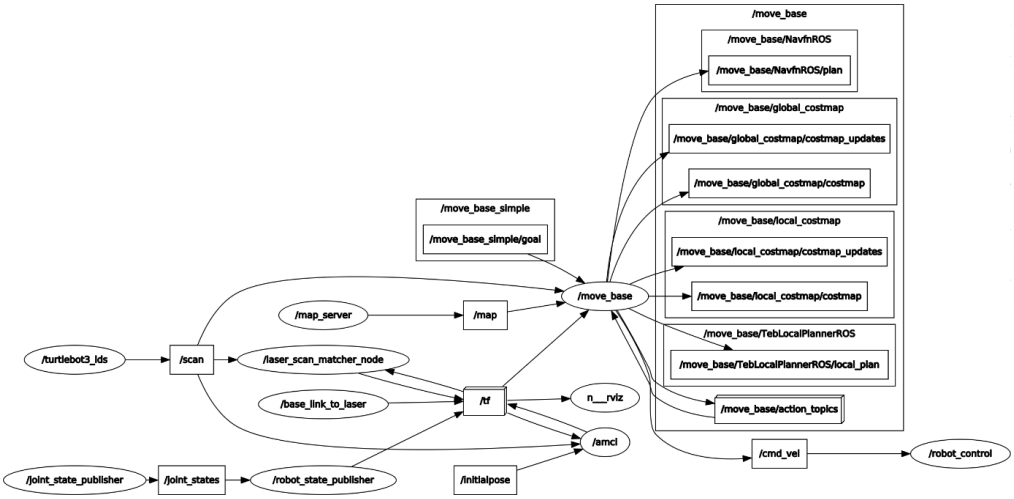


Fig. 8 Estructura de nodos en ejecución para la navegación con Navigation Stack.

y global_planner, estos dos últimos implementan el algoritmo Dijktra y A*.

- Global costmap: Contiene todos los parámetros referentes al comportamiento del Global_Planner y el costo o dificultad del espacio libre u obstáculos que se encuentran en el mapa.
- Local Planner: Es el encargado de mover al robot planeando una trayectoria pequeña que sigue la ruta creada por el *Global Planner* y procesa la información del sensor LIDAR para evitar obstáculos. Si el robot se estanca o encuentra un obstáculo dinámico, el *Local Planner* se encarga de reconfigurar su trayectoria. Proporciona también los comandos de velocidad lineal y angular al *topic /cmd_vel* que se comunica al controlador del robot por medio de un mensaje Twist. Existen tres opciones principales: *base_local_planner, dwa_local_planner (Dynamic Window Approach) y teb_local_planner (Timed Elastic Band)*, todos emplean algoritmos de muestreo y evaluacion.
- Local costmap: Contiene todos los parámetros referentes al comportamiento del Local_Planner y se encarga principalmente del procesamiento de información al detectar obstáculos para evitar que el robot choque con estos, para ello se crea una capa de inflación alrededor de los obstáculos con el fin de mantener al robot aún más alejado de estos.

Cabe mencionar que cada tipo de algoritmo a utilizarse necesita de la configuración de los parámetros adecuados de acuerdo al tipo, dimensiones y comandos para el correcto funcionamiento del robot. La estructura de nodos y *topics* activos se presenta en la Fig. 8.

2) Método de seguimiento de trayectoria

El proceso de seguimiento de trayectoria o también llamado *Path Tracking* se ocupa de determinar, en cada instante de tiempo, la velocidad y ángulo adecuado para que el robot pueda seguir cierta trayectoria preestablecida [17].

El código empleado en este trabajo está basado en el algoritmo geométrico *Follow The Carrot* y en la implementación propuesta por Subodh Malgonde [18] para la navegación de un robot Ackerman.

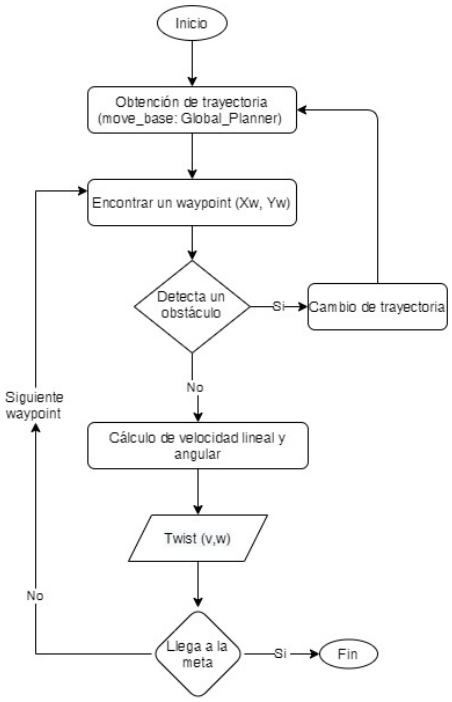


Fig. 9 Diagrama de bloques para el algoritmo de seguimiento de trayectoria.

El código se explica en cinco pasos siguiendo la secuencia presentada en el diagrama de flujo de la Fig. 9:

- La primera parte utiliza el *Global Planner* de *Navigation Stack* para planificar la trayectoria global que puede considerarse como una serie de puntos que representan las coordenadas de la ruta a seguir obtenida a partir del *topic /move_base2/NavfnROS/plan*, esta trayectoria puede variar en el tiempo según el desplazamiento del robot y la interacción con nuevos obstáculos.

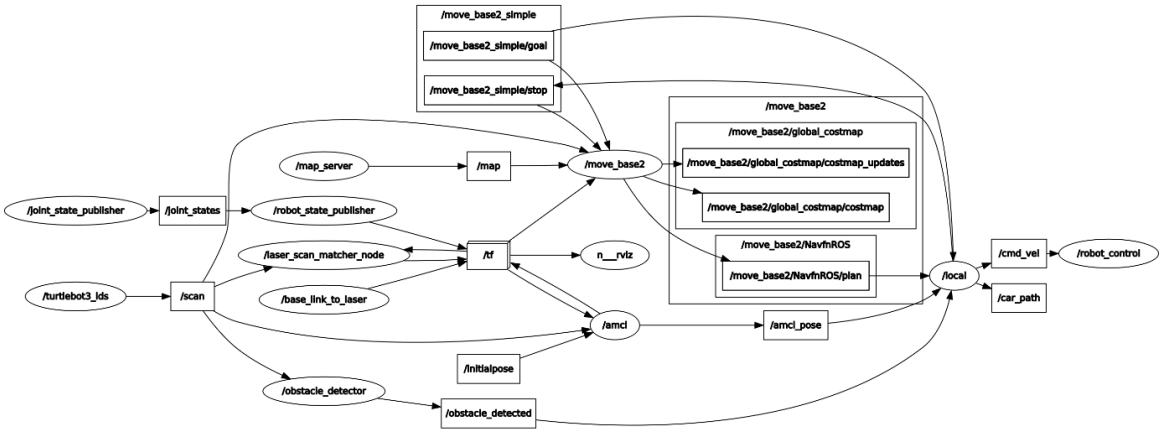


Fig. 10 Estructura de nodos en ejecución para el seguidor de trayectoria.

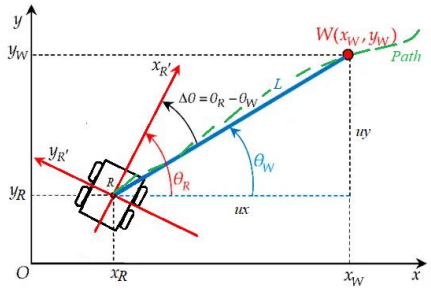


Fig. 11 Representación geométrica de seguimiento de trayectoria.

- El valor de la variable *look_ahead_distance* está preestablecido y representa una distancia L desde la posición actual del robot a un punto en la trayectoria, representando en la Fig. 11, este punto tomará el nombre de *waypoint* y lleva la información de sus coordenadas *Xw* y *Yw*. Luego se calcula las varibles ux y uy que son la diferencia de la distancia desde el robot al *waypoint* en cada eje de coordenadas. La posición del robot está determinada por el *topic* /*amcl_pose*.

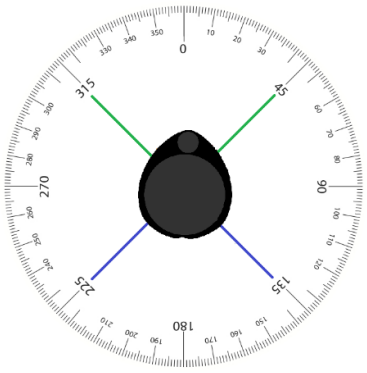


Fig. 12 Lectura del sensor LIDAR para la detección de obstáculos.

- Para la detección y evasión de obstáculos presentes en el entorno y que no son considerados por el *Global_Planner* con anterioridad, se pasan las lecturas del sensor LIDAR por un filtro que calcula los pulsos de la parte frontal del sensor en un rango entre 0° a 45° y 315° a 360°, y de 135° a 225° para la parte posterior, en la Fig. 12 se ilustra como se toman los rangos de detección.
  Con estas consideraciones, el algoritmo detecta si el robot está a una distancia considerablemente corta a un objeto o si se encuentra estancado para realizar maniobras de recuperación y se comunica con el *Global_Planner* para cambiar la trayectoria.

- El siguiente paso es calcular el ángulo θw tomado del centro del robot al *waypoint* y utilizando la ecuación (5) de la tangente inversa, luego se calcula la diferencia de orientación que el robot necesita corregir para alinearse con la trayectoria en (6). La velocidad lineal v varía según la distancia d que cambia conforme el robot se acerca a cada *waypoint* y es calculado con la ecuación (7). Cada valor pasa por un control PD simple para luego enviar los datos por medio de un mensaje tipo Twist al controlador del robot, este por su parte se encarga de tomar estos valores y convertirlos a una señal PWM para cada rueda.

$$\theta w = \tan^{-1}\frac{uy}{ux} \tag{5}$$

$$\Delta\theta = \theta w - \theta R \tag{6}$$

$$v = d = \sqrt{uy^2 + ux^2} \tag{7}$$

- Una vez que el robot llega al *waypoint*, comparando su posición a cada instante de tiempo de ejecución del algoritmo, se procede a encontrar el siguiente *waypoint* y así sucesivamente hasta llegar a la meta.

La estructura de todos los nodos y *topics* activos se presenta en la Fig. 10.

E. Navegación autónoma basada en Reinforcement Learning

Reinforcement Learning (RL) o llamado también Aprendizaje Reforzado es una técnica de Inteligencia Artificial, parte del Machine Learning, en la que el robot aprende de su propio comportamiento mediante acciones en base a un sistema de recompensas positivas y negativas. Los componentes principales que interfieren en el Aprendizaje Reforzado son los siguientes [19]:

- Agente: Es el modelo a entrenar, en este caso corresponde al robot móvil.
- Ambiente: Espacio de interacción del agente.
- Estado: Condición o posición del agente resultado de una acción tomada.
- Acción: Operación que puede hacer el agente al interactuar con el ambiente.

- Recompensa: Resultado positivo o negativo que obtiene el agente al llevar a cabo una acción hacia un estado.

Al principio el agente se encuentra en un ambiente desconocido, pero a partir de tomar la primera acción e intervenir dentro del ambiente, este será recompensado, si la recompensa es positiva se reforzará el comportamiento para el futuro y si es negativa recibirá una penalización.

1) Deep Q-Learning

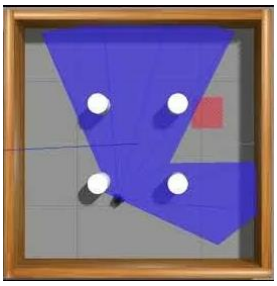


Fig. 13 Entorno de simulación.

Q-Learning es un algoritmo que resuelve el problema de escoger acciones al azar y se enfoca en aprender las recompensas que se obtendrán a largo plazo para cada pareja de estados y acciones. Se representa con la función *Q(s,a)* que devuelve valores Q dependientes de la ejecución de la acción *a* desde el estado *s* y utiliza la ecuación de Bellman para su resolución [20].
El modelo Deep Q-Learning (DQN) combina Q-Learning con Redes Neuronales Profundas para el proceso de aprendizaje del agente, funciona con dos redes neuronales, una principal y una objetivo [21] que aproximan los valores de un estado actual al siguiente, para luego de ejecutarse varios episodios de aprendizaje, poder garantizar recompensas positivas.
Para probar este algoritmo, se utilizó el paquete de Machine Learning proporcionado por Robotis [8] junto con Tensorflow y Keras. El paquete incluye cuatro ambientes de simulación en Gazebo y algoritmos de RL basados en DQN para el robot Turtlebot3 de los que se modificó ciertos parámetros y valores para que funciones con el robot Adeept utilizado en este trabajo. El algoritmo toma las muestras del sensor LDS-01 como los estados del robot en el ambiente de simulación y representan su estado midiendo las distancias al objetivo o a los obstáculos.
Las acciones determinan la velocidad angular para el robot con cinco posibilidades dependiendo del estado en el que se encuentre, representados en la Tabla 1. La velocidad lineal es un valor constante, las recompensas se pueden configurar para cada acción y llegan a un valor de 200 cuando se alcanza el objetivo o -200 cuando choca con un obstáculo. Los demás parámetros configurables son para la función Q de valores.

Tabla 1 Acciones RL

Acción	Velocidad Angular (rad/s)
0	-1.5
1	-0.75
2	0
3	0.75
4	1.5

La Fig. 13 representa uno de los ambientes de simulación en Gazebo con el robot Turtlebot3 Burger y uno de los objetivos.

III. PRUEBAS Y RESULTADOS

A. Resultados del sistema de mapeo

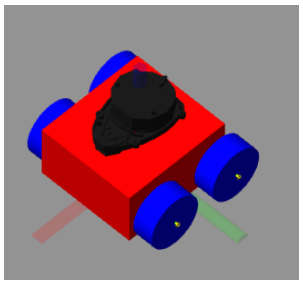


Fig. 14 Modelo URDF del robot.

Antes de realizar el mapeo y localización fue necesario diseñar un modelo URDF del robot para las respectivas simulaciones en Gazebo y para utilizarlo con el gestor de visualización Rviz. El resultado del modelo está en la Fig.14 y se configuró con el plugin Steer Drive de Gazebo que simplifica en gran medida la configuración del controlador del robot, sin embargo, hubo inconvenientes con los parámetros inerciales por el peso y dimensiones reducidas del robot, este problema se solucionó estableciendo valores normalizados de ixx, iyy e izz.

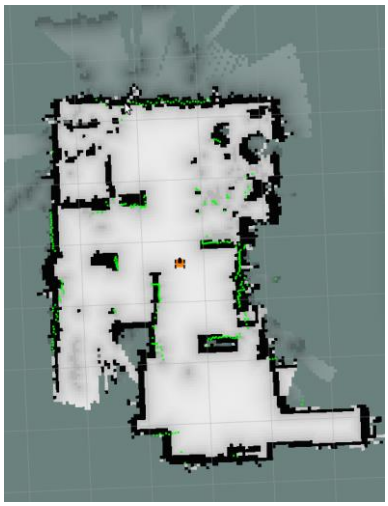


Fig. 15 Mapa obtenido con SLAM.

El siguiente paso fue obtener un mapa en el que el robot pueda navegar. Se empleó la técnica de *SLAM* con el paquete de ROS *gmapping*, ya que fue el que dio mejores resultados al utilizarlo junto con la odometria mediante el sensor LIDAR. En la Fig. 15 está el resultado final del mapa obtenido con esta técnica.

B. Resultados del sistema de navegación con Navigation Stack

La localización del robot se realiza mediante AMCL, un algoritmo probabilístico que emplea el método de Monte Carlo para la localización en un plano de dos dimensiones.

Para la navegación se utilizó el paquete *Navigation Stack* de ROS que presenta una configuración de *Costmaps 2D* y *Layauts* para crear un *global planner* y un *local planner*.
El global planner utilizado es el llamado *Navfn* que emplea el algoritmo Djikstra. También se probó la configuración con A*, pero este genera una trayectoria sin suavizado que hará que el robot no se pueda desplazar con normalidad ya que se trata de un sistema no holónomico.
Para el *local planner* se probaron tres algoritmos: *DWA Local Planner*, *Base Local Planner y Teb Local Planner*, siendo este último el que dio mejores resultados para la navegación.
Es necesario la configuración de varios parámetros según el tipo de robot y las características del controlodar. Esta configuración es a base de prueba y error, los parámetros resultantes de *Teb Local Planner* se presenta en la Tabla 2.

Tabla 2 Parámetros Teb Local Planner

TebLocalPlannerROS			
odom_topic:		/odom	
map_frame:		/map	
Trajectory		Optimization	
teb_autosize:	True	no_inner_iterations:	5
dt_ref:	0.3	no_outer_iterations:	4
dt_hysteresis:	0.1	optimization_activate:	True
global_plan_overwrite_orientation:	True	optimization_verbose:	False
max_global_plan_lookahead_dist:	2	penalty_epsilon:	0.02
feasibility_check_no_poses:	3	weight_max_vel_x:	2
Robot		weight_max_vel_theta:	1
max_vel_x:	0.15	weight_acc_lim_x:	1
max_vel_x_backwards:	0.15	weight_acc_lim_theta:	1
max_vel_theta:	4	weight_kinematics_nh:	1000
acc_lim_x:	2.5	weight_kinematics_forward_drive:	50
acc_lim_theta:	3.5	weight_kinematics_turning_radius:	1
min_turning_radius:	1	weight_optimaltime:	1
footprint_model:	type: point	weight_obstacle:	50
Goal Tolerance		Homotopy Class Planner	
xy_goal_tolerance:	0.4	enable_homotopy_class_planning:	True
yaw_goal_tolerance:	0.6	enable_multithreading:	True
free_goal_vel:	False	simple_exploration:	False
Obstacles		max_number_classes:	4
min_obstacle_dist:	0.2	roadmap_graph_no_samples:	15
include_costmap_obstacles:	True	roadmap_graph_area_width:	5
costmap_obstacles_behind_robot_dist:	1	h_signature_prescaler:	0.5
inflation_dist:	0.7	h_signature_threshold:	0.1
obstacle_poses_affected:	30	obstacle_keypoint_offset:	0.1
costmap_converter_plugin:		obstacle_heading_threshold:	0.45
costmap_converter_spin_thread:	True	visualize_hc_graph:	False
costmap_converter_rate:	5		

C. Resultados del sistema de navegación por seguimiento de trayectoria

Fig. 16 Seguidor de trayectoria.

La siguiente propuesta fue utilizar el *global planner* del nodo *move_base* junto a una *Path Tracking* configurado de modo que pueda seguir las coordenadas de la trayectoria a base de *waypoints* con un valor constante de *look ahead distances* de 70 cm. La Fig. 16 representa como se van obteniendo los llamados *waypoints* siguiendo la trayectoria global. Este código funcionó de manera óptima para la navegación autónoma del robot y es usado para todas las pruebas presentadas a continuación.

1) Error en traslación

Se realizaron pruebas con respecto al desplazamiento del robot utilizando el algoritmo de navegación propuesto para obtener un valor de error absoluto y porcentual.
Los valores de traslación real que se presentan en la Tabla 3 están calculados con la fórmula de distancia entre dos puntos (8), el punto inicial esta tomado del *topic Initialpose* y el final del *topic move_base2_simple/goa*l.

$$Tras.Real = \sqrt{(x_{final} - x_{inicial})^2 + (y_{final} - y_{inicial})^2} \qquad (8)$$

```
pose:
  pose:
    position:
      x: 2.84814842007
      y: -2.97222206088
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: -0.570721766577
      w: 0.821143510694
```

Fig. 17 Ejemplo de mensaje amcl_pose.

Mientras la traslación medida pertenece al desplazamiento del robot obtenida del *topic /amcl_pose* que indica las coordenadas x, y, como también la rotación del robot. Un ejemplo de las coordenadas obtenidas está en la Fig. 17.

En total se tomaron cuatro distancias para el cálculo con un resultado de error promedio igual al 13.253%.

Tabla 3 Error en traslación

Traslación Medida (m)	Traslación Real (m)	Error Absoluto	Error Relativo
0.465	0.500	0.035	0.069
0.755	1.000	0.245	0.245
1.341	1.500	0.159	0.106
1.779	2.000	0.221	0.110

Error promedio %	13.253

2) Error en rotación

Al igual que para los cálculos de traslación se utilizaron los *topics* respectivos para realizar los cálculos de rotación del robot a distintos ángulos. Las orientaciones obtenidas están en notación de cuaterniones[2] y se convirtieron en ángulos de Euler

[2] Números hipercomplejos de cuatro componentes, una real y tres imaginarias.

para facilitar los cálculos. El resultado se muestra en la Tabla 4 con un error promedio de 22.758%.

Tabla 4 Error en rotación

Rotación Medida (Grados)	Rotación Real (Grados)	Error Absoluto	Error Relativo
48.107	52.634	4.527	0.086
68.148	90.884	22.736	0.250
266.767	207.223	59.544	0.287
230.369	179.024	51.345	0.287
		Error promedio %	22.758

El error de rotación es más significativo lo que conlleva a una acumulación en la desorientación del robot con respecto a la posición final asignada. Se debe tomar en cuenta también que el algoritmo está configurado con un valor umbral en la rotación de 25 grados, esto significa que si la diferencia entre el ángulo del *waypoint* y el ángulo de la posición del robot, obtenidos en un instante de tiempo, es menor o igual a 25 grados, el robot no rotará para evitar que entre en un bucle de oscilación girando entre la derecha y la izquierda de su posición.

3) Escenarios de prueba

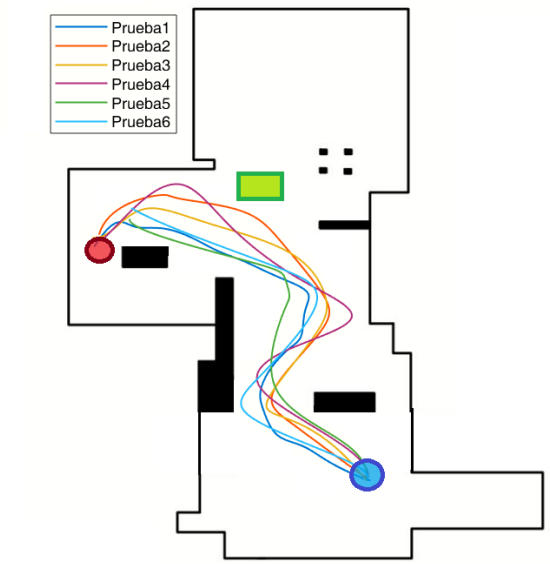


Fig. 18 Escenario 1.

Se dispuso de dos escenarios para probar el algoritmo de navegación, ambos con la misma posición inicial y un obstáculo que no pertenece al mapa estático obtenido con SLAM. El objetivo de estas pruebas es que el robot se desplace por el mapa siguiendo la trayectoria obtenida con el *global planner* de *Navigation Stack* de ROS y que evada cualquier obstáculo. De cada escenario se realizaron seis pruebas.

La Fig. 18 muestra el primer escenario, el punto azul indica la posición inicial, el punto rojo la posición de meta, el recuadro verde indica el obstáculo no estático y las curvas de color representan las seis trayectorias del robot. El resultado obtenido cumple el objetivo planteado, que es llegar a la meta de forma autónoma y sin colisiones, pero como se indicó anteriormente existirá un porcentaje de error con respecto a su posición final, el análisis de este error está en la Tabla 5 que compara las coordenadas de cada posición final del robot con respecto a las coordenadas del punto de meta común asignado.

El valor más significativo se muestra en la rotación, que acumula un error mientras sigue la trayectoria, con un valor del 35.62%. También se obtuvo la distancia promedio entre la posición final del robot y la meta con un valor de 0.281 m.

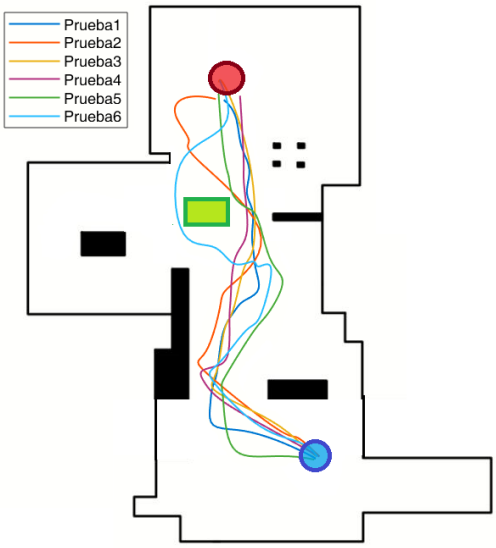


Fig. 19 Escenario 2.

El siguiente escenario presenta una trayectoria más lineal como se muestra en la Fig. 19 donde es más evidente el cambio en la ruta una vez que se detectó el obstáculo en verde, así mismo, se realizaron seis pruebas, los resultados se presentan en la Tabla 6.

Por las características de la trayectoria del robot, el error de rotación disminuye y el error de traslación aumenta con un valor del 22.422% en el eje X y de 31.501% en el eje Y. La distancia promedio en este escenario es de 0.187 m.

Para poder comparar los resultados, se realizaron las mismas seis pruebas en ambos escenarios con el robot Turtlebot 3 Burger ocupando el *Navigation_Stack* de ROS y aplicando el algoritmo de navegación *DWA*. Los resultados en el primer escenario en la Tabla 7 muestran un porcentaje de error menor que con el robot Adeept y la distancia promedio es 8.1 cm menor.

En el segundo escenario con los resultados de la Tabla 8 se obtiene un error considerable en el eje *Y* de 49.892% y la distancia promedio también sube 4.8 cm con respecto a los resultados del robot Adeept.

Para finalizar, se calculo el intervalo de confianza para las tres coordenadas en cada escenario propuesto, tanto para el robot Adeept como para el robot Turtlebot 3. El resultado se presenta en cada una de las tablas.

Tabla 5 Resultados escenario 1

Escenario 1		Pos. Final Robot	Goal	Error Absoluto	Error Relativo	Distancia (m)
1	x	-0.982	-1.040	0.058	0.056	0.209
	y	-2.048	-2.249	0.201	0.089	
	θ grados	-139.171	-91.673	47.498	0.518	
2	x	-0.947	-1.040	0.093	0.098	0.244
	y	-2.023	-2.249	0.226	0.112	
	θ grados	-104.393	-91.673	12.720	0.122	
3	x	-0.997	-1.040	0.043	0.043	0.199
	y	-2.055	-2.249	0.194	0.094	
	θ grados	-143.239	-91.673	51.566	0.360	
4	x	-0.938	-1.040	0.102	0.109	0.184
	y	-2.096	-2.249	0.153	0.073	
	θ grados	-141.177	-91.673	49.504	0.351	
5	x	-0.709	-1.040	0.331	0.467	0.508
	y	-1.863	-2.249	0.386	0.207	
	θ grados	-168.278	-91.673	76.604	0.455	
6	x	-0.902	-1.040	0.138	0.153	0.344
	y	-1.934	-2.249	0.315	0.163	
	θ grados	-137.109	-91.673	45.436	0.331	

Error Promedio %	
x	15.428
y	12.309
θ	35.620

Distancia Promedio(m)	0.281

Intervalo de Confianza	
x	-0,91 ± 0.11
y	-2,00 ± 0.09
θ	-138,90 ± 21,41

Tabla 6 Resultados escenario 2

Escenario 2		Pos Final Robot	Goal	Error Absoluto	Error Relativo	Distancia (m)
1	x	0.522	0.472	0.050	0.106	0.245
	y	-0.704	-0.464	0.240	0.517	
	θ grados	101.012	89.210	11.803	0.132	
2	x	0.258	0.472	0.214	0.453	0.312
	y	-0.691	-0.464	0.227	0.489	
	θ grados	65.489	89.210	23.720	0.266	
3	x	0.361	0.472	0.111	0.235	0.123
	y	-0.516	-0.464	0.052	0.112	
	θ grados	102.445	89.210	13.235	0.148	
4	x	0.596	0.472	0.124	0.263	0.201
	y	-0.622	-0.464	0.158	0.341	
	θ grados	92.074	89.210	2.865	0.032	
5	x	0.359	0.472	0.113	0.239	0.207
	y	-0.638	-0.464	0.174	0.375	
	θ grados	72.995	89.210	16.215	0.182	
6	x	0.495	0.472	0.023	0.049	0.035
	y	-0.490	-0.464	0.026	0.056	
	θ grados	37.471	89.210	51.738	0.580	

Error Promedio %	
x	22.422
y	31.501
θ	22.340

Distancia Promedio (m)	0.187

Intervalo de Confianza	
x	0,43 ± 0.13
y	-0,61 ± 0.09
θ	78,58 ± 26,33

Tabla 7 Resultados escenario 1 con Turtlebot 3

Escenario 1 Turtlebot 3		Pos. Final Robot	Goal	Error Absoluto	Error Relativo	Distancia (m)
1	x	-1.009	-1.040	0.031	0.030	0.162
	y	-1.889	-2.048	0.159	0.078	
	θ grados	-97.575	-91.685	5.889	0.064	
2	x	-0.922	-1.040	0.118	0.128	0.300
	y	-1.772	-2.048	0.276	0.156	
	θ grados	-134.301	-91.685	42.616	0.317	
3	x	-0.960	-1.040	0.080	0.083	0.146
	y	-1.926	-2.048	0.122	0.063	
	θ grados	-85.256	-91.685	6.429	0.075	
4	x	-1.023	-1.040	0.017	0.017	0.161
	y	-1.888	-2.048	0.160	0.085	
	θ grados	-92.475	-91.685	0.790	0.009	
5	x	-1.037	-1.040	0.003	0.003	0.179
	y	-1.869	-2.048	0.179	0.096	
	θ grados	-94.996	-91.685	3.311	0.035	
6	x	-0.980	-1.040	0.060	0.061	0.250
	y	-1.805	-2.048	0.243	0.135	
	θ grados	-114.821	-91.685	23.135	0.201	

Error Promedio %	
x	5.370
y	10.195
θ	11.698

Distancia Promedio(m)	0.200

Intervalo de Confianza	
x	-0,99 ± 0.05
y	-1,86 ± 0.06
θ	-103,24 ± 19

Tabla 8 Resultados escenario 2 con Turtlebot 3

Escenario 2 Turtlebot 3		Pos. Final Robot	Goal	Error Absoluto	Error Relativo	Distancia (m)
1	x	0.461	0.472	0.011	0.023	0.278
	y	-0.742	-0.464	0.278	0.599	
	θ grados	99.637	89.210	10.428	0.117	
2	x	0.487	0.472	0.015	0.032	0.293
	y	-0.757	-0.464	0.293	0.631	
	θ grados	106.799	89.210	17.590	0.197	
3	x	0.507	0.472	0.035	0.074	0.145
	y	-0.605	-0.464	0.141	0.304	
	θ grados	88.694	89.210	0.516	0.006	
4	x	0.491	0.472	0.019	0.040	0.322
	y	-0.785	-0.464	0.321	0.692	
	θ grados	96.887	89.210	7.678	0.086	
5	x	0.391	0.472	0.081	0.172	0.189
	y	-0.635	-0.464	0.171	0.369	
	θ grados	87.147	89.210	2.063	0.023	
6	x	0.471	0.472	0.001	0.002	0.185
	y	-0.649	-0.464	0.185	0.399	
	θ grados	100.210	89.210	11.001	0.123	

Error Promedio %	
x	5.720
y	49.892
θ	9.206

Distancia Promedio (m)	0.235

Intervalo de Confianza	
x	0,47 ± 0.04
y	-0,70 ± 0.08
θ	96,56 ± 7,83

D. Resultados de la navegación aplicando Reinforcement Learning

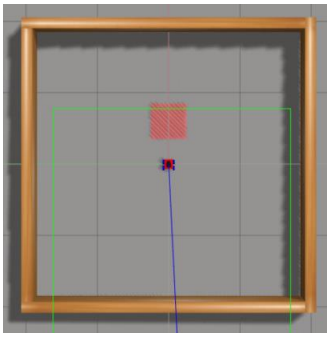


Fig. 20 RL escenario 1.

Para probar el aprendizaje reforzado aplicando *Deep-Q Learning* se utilizaron dos escenarios de simulación. El primer escenario presenta un ambiente limitado y sin obstáculos (Fig. 20), el robot debe completar al menos 190 episodios para aprender las acciones positivas que le lleven al objetivo con mayor probabilidad de éxito. Cada diez episodios los resultados se guardan en un archivo de extensión .h5 a los que el algoritmo puede acceder después.

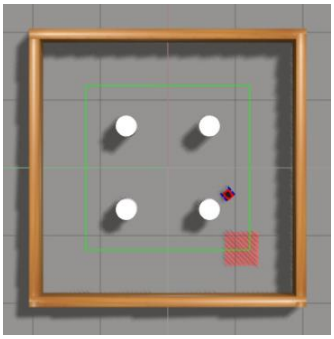


Fig. 21 RL escenario 2.

El segundo escenario (Fig. 21) dispone de cuatro obstáculos estáticos aumentando la dificultad para el aprendizaje siendo necesario un mínimo de 270 para que el robot presente mejores resultados.

Los dos escenarios restantes no llegaron a completarse por la demanda computacional del algoritmo, lo que resultó en una ejecución considerablemente lenta para el aprendizaje del robot.

IV. Conclusiones y Recomendaciones

Este proyecto presenta el desarrollo de un sistema autónomo para un robot móvil utilizando principalmente un sensor LIDAR montado en una Raspberry Pi, junto con el *framework* ROS que incluye todos los paquetes, librerías y herramientas de software libre que fueron necesarios para cumplir con el objetivo del trabajo.

Los algoritmos tanto de SLAM, localización y navegación necesitan datos odométricos para funcionar y la solución a este problema fue emplear la librería *laser_scan_matching* que utiliza solo al sensor LIDAR como fuente de datos para la odometría. Sin embargo, el resultado de los errores que presentan las pruebas se podría disminuir combinando al sensor LIDAR con una cámara RGB-D y obtener una mejor lectura del entorno del robot, así como también obtener mapas 3D. Así mismo, los errores obtenidos se compararon con los resultados de un robot Turtlebot 3 Burger y se recomienda probar los escenarios planteados con otros robots móviles para medir su desempeño y mejorar los resultados.

También se propone aplicar este sistema de autonomía a robots más robustos enfocados a la industria y actividades de trabajo pesado, como una herramienta para el traslado de objetos y materiales, o también como una herramienta para utilizarlo dentro del hogar a la que se le puede dar múltiples tareas e incluso desarrollar una aplicación móvil para manejar todos los comandos de una forma sencilla y eficiente.

Para trabajos futuros, se recomienda migrar el sistema a ROS2, la nueva versión de ROS que trae paquetes actualizados con

mayor compatibilidad con otros sistemas operativos y lenguajes de programación, y también utilizar otro tipo de microcontrolador como, por ejemplo: el de los módulos NVIDIA Jetson para mejorar el procesamiento de algoritmos de IA, procesamiento de imágenes y algoritmos de visión por computadora. En este punto se propone ampliar la experimentación con el algoritmo de Reinforcement Learning y comenzar con las pruebas en un entorno real.

REFERENCIAS

[1] F. Fandiño, «Un Robot Móvil Autónomo y un Vehículo Guiado Tradicional. ¿Cuál es la diferencia para la industria? - TecnoAlimen», *Tecnoalimen*, 2019. https://www.tecnoalimen.com/articulos/20190423/diferencias-entre-robot-movil-autonomo-vehiculo-guiado-tradicional#.YP5OfehKg2x (accedido jul. 26, 2021).

[2] V. Seib, R. Memmesheimer, y D. Paulus, «A ROS-Based System for an Autonomous Service Robot», *Robot Oper. Syst.*, vol. 1, 2016.

[3] J. Iqbal, R. Xu, S. Sun, y C. Li, «Simulation of an Autonomous Mobile Robot for LiDAR-Based In-Field Phenotyping and Navigation», *Robotics*, vol. 9, n.º 2, Art. n.º 2, jun. 2020, doi: 10.3390/robotics9020046.

[4] «Documentation - ROS Wiki». http://wiki.ros.org/ (accedido jul. 02, 2021).

[5] S. LaValle, *Planning Algorithms*, First Edition. Cambridge University Press. Accedido: jul. 02, 2021. [En línea]. Disponible en: http://lavalle.pl/planning/

[6] Magnus Egerstedt, *Control of Mobile Robots- 2.2 Differential Drive Robots*. Accedido: jul. 02, 2021. [En línea Video]. Disponible en: https://www.youtube.com/watch?v=aE7RQNhwnPQ&list=LL&index=12

[7] Hadabot, «Unicycle to Differential Drive: Coursera's Control of Mobile Robots with ROS and ROSbots — Part 2», *Medium*, mar. 19, 2018. https://medium.com/hackernoon/unicycle-to-differential-drive-courseras-control-of-mobile-robots-with-ros-and-rosbots-part-2-6d27d15f2010 (accedido jul. 02, 2021).

[8] ROBOTIS, «ROBOTIS e-Manual», *ROBOTIS e-Manual*. https://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/ (accedido jul. 06, 2021).

[9] H. A. Sidharta, S. Sidharta, y W. P. Sari, «2D Mapping and boundary detection using 2D LIDAR sensor for prototyping Autonomous PETIS (Programable Vehicle with Integrated Sensor)», *Kinet. Game Technol. Inf. Syst. Comput. Netw. Comput. Electron. Control*, pp. 107-114, mar. 2019, doi: 10.22219/kinetik.v4i2.731.

[10] A. Censi, «CSM». https://censi.science/software/csm/ (accedido jul. 06, 2021).

[11] S. Thrun, W. Burgard, y D. Fox, *PROBABILISTIC ROBOTICS*. The MIT Press, 2005.

[12] G. Grisetti, C. Stachniss, y W. Burgard, «OpenSLAM.org». https://openslam-org.github.io/gmapping.html (accedido jul. 09, 2021).

[13] J. M. Santos, D. Portugal, y R. P. Rocha, «An evaluation of 2D SLAM techniques available in Robot Operating System», en *2013 IEEE International Symposium on Safety, Security, and Rescue Robotics (SSRR)*, Linkoping, Sweden, oct. 2013, pp. 1-6. doi: 10.1109/SSRR.2013.6719348.

[14] M. Quigley, B. Gerkey, y W. Smart, *Programming Robots with ROS*. Sebastopol: O'REILLY, 2015.

[15] A. Mahtani, L. Sanchez, E. Fernandez, A. Martinez, y L. Joseph, *ROS Programming: Building Powerful Robots*. Birmingham: Packt Publishing Ltd., 2018.

[16] C. Fairchild y T. Harman, *ROS Robotics By Example*. Birmingham: Packt Publishing Ltd., 2016. [En línea]. Disponible en: www.packtpub.com

[17] M. Lundgren, «Path Tracking for a Miniature Robot», *Umeå University*, p. 9, 2003.

[18] S. Malgonde, «Building an actual self driving car», *Medium*, jul. 29, 2019. https://medium.com/@subodh.malgonde/building-an-actual-self-driving-car-53f67ca41566 (accedido jul. 11, 2021).

[19] R. Gandhinathan y L. Joseph, *ROS Robotics Projects*, 2nd ed. Birmingham: Packt Publishing Ltd., 2019.

[20] M. S. Ausin, «Introducción al aprendizaje por refuerzo. Parte 2: Q-Learning.», *Medium*, nov. 25, 2020. https://markelsanz14.medium.com/introducci%C3%B3n-al-aprendizaje-por-refuerzo-parte-2-q-learning-883cd42fb48e (accedido jul. 24, 2021).

[21] M. S. Ausin, «Introducción al aprendizaje por refuerzo. Parte 3: Q-Learning con redes neuronales, algoritmo DQN.», *Medium*, nov. 25, 2020. https://markelsanz14.medium.com/introducci%C3%B3n-al-aprendizaje-por-refuerzo-parte-3-q-learning-con-redes-neuronales-algoritmo-dqn-bfe02b37017f (accedido jul. 24, 2021).