Proyecto de reto integrador con Manchester Robotics, navegación y recolección autónoma de objetos con Puzzlebot Equipo 2

Fernando. Cuéllar Martínez[1], Jorge Antonio. Villegas Hernández[2], María Fernanda. Hernández Montes[3], Melissa. Montemayor Riojas[4] and Melissa. Vélez Martínez[5]

[1] *Tecnológico de Monterrey, Escuela de Ingeniería y Ciencias, Nuevo León, México*
[6] *phD. Luis Alberto Muñoz Ubando*
[7] *phD. Rafael Enrique Mendoza Crespo*
[8] *phD. Luis Ricardo Salgado Garza*
[9] *phD. Arturo Eduardo Cerón López*
[10] *phD. Alfonso Ávila Ortega*

Fecha: 06/06/2023

Abstract— The main objective of this challenge is to generate value by increasing and ensuring the quality in production and service chains. The Puzzlebot developed by Manchester Robotics is used to demonstrate a classic loading and unloading algorithm, integrating hardware and software such as cameras and LiDAR sensors. The results aid in the redefinition of operational spaces of the complex scenario that occurs in supply chain warehouses and contributes to improving the quality of manual work by eliminating repetitive and hazardous tasks.

Keywords— Automation, Logistics Industry, Warehouses, Simulation, ROS (Robot Operating System), LiDAR, Puzzlebot

I. Introducción

El reto consiste en explorar un ambiente desconocido con el Puzzlebot implementado en Gazebo y posteriormente en físico. Se desarrolla un algoritmo de localización utilizando el filtro de Kalman y la odometría del robot. Se hace lectura de ArUcos para la medición a través de las transformadas según la imagen recibida por la cámara. Se genera la elipse de incertidumbre según la visión de ArUcos y su posición respecto a obstáculos.

Además, se utilizan los algoritmos Bug para evitar obstáculos y dirigirse hacia cada destino definido. El robot es capaz de atravesar trayectorias desconocidas a través de las mediciones y referencias visuales a partir de ArUcos y los obstáculos.

II. Navegación y antecedentes de mini retos

En esta sección mencionamos la creación de algoritmos de navegación con el robot diferencial con base en algoritmos reactivos como Bug 0 y Bug 2 en la compañía de la localización y reconocimiento de obstáculos mediante el LiDAR en el simulador de Gazebo. Los algoritmos utilizan estrategias precisas para navegar a través de obstáculos y llegar a la posición deseada.

a. Descripción de algoritmos utilizados:

Los algoritmos Bug son un conjunto de pasos para la planeación de movimiento de robots. Son usualmente utilizados en escenarios donde un robot necesita llegar a una localización precisa o hacia algún objeto, estos algoritmos se implementan en ROS para la simulación y desarrollo de los mismos. Son algoritmos de planificación de rutas utilizados en robótica móvil se usan a menudo en robots diferenciales para evitar obstáculos y llegar al destino deseado. [1]

1. El algoritmo Bug 0:

Es el algoritmo más simple, es un orden de pasos que dirige al robot en dirección de los obstáculos, comienza con una línea recta hacia el objetivo hasta encontrarse con algún obstáculo y al tener un objeto frente a él se genera un contorno alrededor, el cual comienza a seguir el robot hasta retomar la línea recta que originalmente seguía. Este algoritmo no considera la figura del obstáculo y podría quedarse bloqueado en caso de no encontrar cómo redirigirse hacia el objetivo.

Este algoritmo no considera la distancia hacia el objeto, e igualmente no considera la figura del obstáculo que está rodeando. [2]

El robot podría no completar el algoritmo si el obstáculo bloquea completamente la trayectoria entre el robot y el objeto. Se considera como un código de baja complejidad y un algoritmo simple, aunque dependiendo de la situación, el obstáculo podría hacer que el robot tome rutas ineficientes y extender el tiempo en resolver una tarea simple. [3]

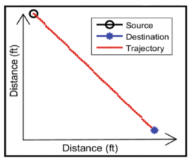

Fig. 1: Se observa el rendimiento del Bug 0 sin obstáculo.

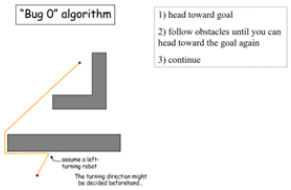

Fig. 2: Se observa la línea recta que sigue el robot, rodea el obstáculo y sigue en la trayectoria indicada.

2. El algoritmo Bug 1:

Este algoritmo mejora la comparación del anterior al utilizar un mecanismo para identificar si el robot se ha ciclado al estar intentando evitar los obstáculos. Registra la distancia más cercana al objeto y si reconoce que ya recorrió una trayectoria previa, regresa al punto más cercano e intenta encontrar otra solución partiendo desde ese punto. [4]

El Bug 1 podría no encontrar una solución si se cicla y no encuentra nuevos caminos que lo dirijan hacia el objeto. Previene que el robot se cicle siguiendo la misma ruta para evitar el obstáculo, pero aun así podría tomar caminos largos dependiendo de la complejidad de la ruta alrededor del obstáculo.

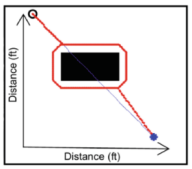

Fig. 3: Se observa el rendimiento del Bug 1 con obstáculo.

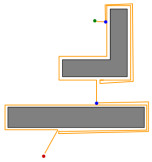

Fig. 4: Se observa la trayectoria del robot donde completa la vuelta y regresa a la posición más cercana al objeto para retomar su trayectoria.

3. El algoritmo Bug 2:

Este algoritmo es una mejora al Bug anterior, en el cual se utiliza la información del entorno para encontrar el camino más eficiente según la figura del obstáculo presente. En este caso el robot se mueve en línea recta hasta encontrar un obstáculo y comienza a seguir el contorno del mismo. Aun así, si el robot identifica que regresó a su punto de inicio, entonces busca un camino alternativo que lo dirija nuevamente al objetivo. El algoritmo considera la figura del obstáculo y toma decisiones a partir de la información obtenida. [5]

El algoritmo es capaz de encontrar una solución en la mayoría de los casos debido a que garantiza encontrar un camino al objeto siempre y cuando tenga conocimiento de su entorno. Este código puede encontrarse con comportamientos ineficientes debido a su entorno. Igualmente, utiliza el backtracking, causando el recalculo de los caminos previos, y utiliza una combinación de trayectorias directas y evasión de obstáculos. [6]

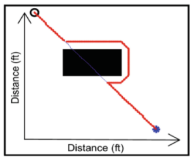

Fig. 5: Se observa el rendimiento del Bug 2 con obstáculo.

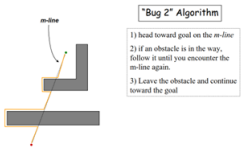

Fig. 6: Se observa la trayectoria del robot considerando la figura del obstáculo y el aprovechamiento de información para retomar la línea recta original.

b. Proceso para la creación del algoritmo Bug 0 y Bug 2

Para la implementación de los algoritmos en un robot diferencial modelado con ROS como el Puzzlebot se siguen los siguientes pasos.

El primer paso para implementar el algoritmo Bug 0 es definir el espacio de búsqueda o mundo en este caso. Para ello, se deben considerar las limitaciones de geometría, tamaño y movimiento del robot. Luego, se debe determinar el objetivo de navegación deseado y se deben identificar los posibles obstáculos en el camino. Una vez que se define el modo de búsqueda, puede comenzar la planificación de la ruta. El algoritmo Bug 0 consta de dos fases: una fase de seguimiento de paredes y una fase de navegación directa. En la fase posterior al muro, el robot se desplaza a lo largo del perímetro de los obstáculos para encontrar un hueco que le permita alcanzar su objetivo. En la fase de navegación directa, el robot se desplaza directamente al destino deseado.

Para implementar el paso de seguimiento de paredes, se debe definir un algoritmo de detección de obstáculos que permita al robot detectar cuando está frente a un obstáculo. Entonces se debe definir una estrategia para seguir el contorno del obstáculo hasta encontrar la abertura deseada. Esto se puede lograr con sensores de distancia o planificando una ruta que siga la forma del obstáculo.
Finalmente, para la fase de navegación directa, se debe definir una estrategia de control de velocidad y dirección para permitir que el robot se mueva en línea recta hacia el objetivo deseado. Esto se puede hacer utilizando un controlador PID (proporcional en nuestro caso) o planificando una ruta que apunte directamente al objetivo. En resumen, la implementación del algoritmo Bug 0 en un robot diferencial en ROS incluye la definición del espacio de búsqueda, la detección de obstáculos, la planificación de la trayectoria, la velocidad y dirección del robot. La implementación exitosa del algoritmo Bug 0 permite que el robot navegue de forma independiente alrededor de los obstáculos y alcance la meta deseada.

Similar al código anterior, el algoritmo Bug 2 consta de tres fases: una fase de seguimiento de muros, una fase de evasión de obstáculos y una fase de navegación directa. En la fase de seguimiento de paredes, el robot se desplaza por el perímetro de los obstáculos para encontrar un hueco que le permita alcanzar su objetivo. Cuando el robot encuentra un obstáculo que no puede eludir, entra en la fase de evasión de obstáculos. En ese punto, el robot se mueve alrededor del obstáculo hasta encontrar un hueco que le permita llegar a su destino. En la fase de navegación directa, el robot se desplaza directamente al destino deseado. Para implementar el paso de seguimiento de la pared, se debe definir el algoritmo de detección de obstáculos utilizado en el algoritmo Bug 0, después se debe definir una estrategia que siga el contorno del obstáculo hasta encontrar el espacio deseado. Si el robot no encuentra un espacio después de pasar por alto el obstáculo completo, se mueve hacia el paso por alto del obstáculo.
En la fase de evasión de obstáculos, se debe definir una estrategia de planificación de trayectoria para permitir que el robot se desplace alrededor del obstáculo para encontrar un espacio que le permita llegar a su destino. Esto se puede hacer usando sensores de distancia o planificando una ruta alrededor de un obstáculo. Finalmente, para la fase de navegación directa, se debe definir una estrategia de velocidad y dirección que permita al robot moverse en línea recta hacia el destino deseado. Esto se puede hacer utilizando un controlador PID (proporcional en nuestro caso) o planificando una ruta que apunte directamente al objetivo.

c. Dificultades generales en los algoritmos

1. Problemas de detección de obstáculos: Uno de los principales problemas al implementar el algoritmo Bug 0 o Bug 2 en un robot diferencial en ROS es la detección de obstáculos en el entorno. La precisión de los sensores del robot y la capacidad para detectar obstáculos pueden verse limitadas, lo que puede conducir a una planificación incorrecta de la ruta que puede resultar en una colisión con los obstáculos. Además, la detección de obstáculos puede ser aún más difícil en entornos con obstáculos ocultos (como paredes altas).

2. Calibración de sensores: Otra dificultad que puede surgir al implementar el algoritmo Bug 0 o Bug 2 en un robot de diferenciación ROS es la calibración de los sensores del robot. La precisión y confiabilidad de los sensores son fundamentales para el correcto funcionamiento del algoritmo, y los problemas potenciales con la calibración del sensor pueden afectar su capacidad para detectar y evitar obstáculos con precisión.

3. Limitaciones de hardware: el hardware utilizado para implementar el algoritmo puede ser no óptimo, lo que puede afectar la capacidad del robot para implementar y ejecutar el algoritmo de manera efectiva. Los requisitos de hardware para el procesamiento de imágenes y la planificación de rutas pueden ser muy altos, y si el hardware del robot no cumple con estos requisitos, es posible que el robot no pueda implementar el algoritmo de manera efectiva.

4. Dificultad en la planificación de rutas: Una de las principales tareas del algoritmo Bug 0 o Bug 2 es planificar una ruta para evitar obstáculos y llegar al destino deseado. Sin embargo, la planificación de rutas puede ser una tarea desafiante en entornos complejos con muchos obstáculos y geometría irregular. El algoritmo debe ser capaz de planificar trayectorias efectivas y eficientes en tiempo real para evitar colisiones y llegar a tiempo al destino deseado.

d. Forma en que resolvieron los problemas que se presentaron

Para resolver los problemas presentados, se realizó una amplia investigación respecto a los temas en los que se tenía duda. Se consultaron soluciones ya realizadas e implementadas con anterioridad, para usar las fuentes de referencia generar un código que tuviera robustez. Además, se abordaron diferentes acercamientos para la programación de los códigos, buscando innovar o agregar contenido a las soluciones consultadas previamente. Por otro lado, el proyecto se está desarrollando a la par que otros equipos, por lo que fue posible acudir a ellos y comparar los resultados que estaba teniendo nuestra solución, contra sus implementaciones.

e. Condiciones o circunstancias en las que sería apropiado combinar estos algoritmos

Los algoritmos Bug tienen la finalidad de brindar un conjunto de pasos a seguir para procesos de búsqueda en robótica, siendo implementados en escenarios en los que es necesario llegar a una meta establecida, rodeando una cantidad desconocida de obstáculos. En este contexto, es necesario que el robot rodee el obstáculo de manera cercana al borde hasta que encuentre una apertura para continuar con su camino.

Realizando una comparativa entre estos algoritmos, existen algunos puntos que es necesario considerar de acuerdo con la implementación de cada uno de ellos.

Bug 0. Su principal ventaja es que no requiere de información previa sobre su entorno para funcionar, lo que permite que se le utilice en sistemas robóticos con recursos limitados; aunque, por otra parte, genera complicaciones en entornos en los que el objetivo no se encuentra en línea recta con el punto de partida, además de que no encuentra soluciones óptimas en entornos complejos.

Bug 1. Implementa información heurística para el diseño de una trayectoria hacia el objetivo, lo que le permite sortear objetos pequeños en su camino. No obstante, es posible que este algoritmo se encuentre atrapado en bucles en algunas ocasiones, además de que no es capaz de sortear obstáculos grandes, por lo que no puede encontrar una solución en entornos muy complejos.

Bug 2. Es un algoritmo mucho más eficiente que los dos anteriores, pues es capaz de sortear obstáculos grandes, empleando información heurística para optimizar el camino. Sin embargo, debido a que es el algoritmo más complejo de los tres Bug, usualmente requiere de más recursos computacionales, además de que aún está limitado en entornos extremadamente complejos, tales como aquellos en los que el objetivo se encuentra en una región cercada por obstáculos.

La combinación de los algoritmos Bug 0, y Bug 2 puede ser una estrategia muy útil para resolver el problema de la navegación autónoma de los robots móviles en entornos complejos. Cada uno de estos algoritmos tiene sus propias ventajas y limitaciones, y una combinación de ellos puede superar estas limitaciones.

El algoritmo Bug 0 es útil para evitar obstáculos, pero puede resultar en una trayectoria no óptima hacia el objetivo. El algoritmo Bug 2 puede evitar obstáculos y encontrar el camino más corto, pero puede ser menos eficiente que otros algoritmos en entornos complejos.

Una combinación de estos algoritmos puede aprovecharlos todos y minimizar sus limitaciones. Por ejemplo, el algoritmo Bug 0 o Bug 2 se puede usar para sortear obstáculos para acercarse lo más posible a la meta, y luego con apoyo del LiDAR y las cámaras se puede usar para encontrar la ruta más corta desde la ubicación actual hasta la meta. Alternativamente, el algoritmo 0 se puede usar para encontrar una ruta inicial aproximada, y luego el algoritmo Bug 2 se puede usar para refinar la ruta y evitar obstáculos en el camino.

En general, una combinación de algoritmos de navegación puede ser una estrategia muy útil para resolver problemas de navegación autónoma en entornos complejos. La elección de la estrategia adecuada depende de las necesidades y limitaciones específicas del proyecto.

f. Video de demostración

Dentro de la siguiente carpeta se encuentran 2 simulaciones sobre los códigos presentados, Bug 0 y Bug 1, ambos están presentados en simulación en el mundo de Gazebo, donde se puede ver presentado el reconocimiento de obstáculos, seguimiento de paredes y llegada hacia objetivos con los diferentes funcionamientos.

Link: Videos de demostración.

III. Reto con Manchester Robotics

a. Reconocimiento de ArUcos

Durante el mapeo y la navegación que realiza el Puzzlebot, la identificación de ArUcos utilizando visión funciona como puntos guía a lo largo de la pista para ubicar la posición actual en la que se encuentra el robot. Al tener una mayor cantidad y localización estratégica de ArUcos en el mapa, sirven como referencia para tener una mejor precisión en la ubicación origen y la ubicación destino. Cada uno de los ArUcos posicionados en el mundo cuentan con una imagen distinta que le permite al Puzzlebot llevar un mejor registro de la información, además si el ArUco se encuentra visible en la totalidad de las caras del objeto, en este caso un cubo, reduce la probabilidad de errores de identificación.

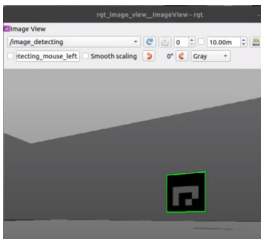

Fig. 7: Se implementó la detección de ArUcos para funcionar como guías del Puzzlebot a través de la pista.

1. Reconocimiento de distancia hacia los ArUcos:

El proceso de reconocimiento de ArUcos comienza con la verificación de la detección de un ArUco, verificando los extremos de la imagen y detectando el punto central.

El método *publish_sensor_data* es el encargado de calcular la distancia en *delta_x* y *delta_y* hacia el ArUco detectado en relación con la posición actual del robot. Aquí se realiza el cálculo de las diferencias en las coordenadas x e y entre la posición actual del robot y las coordenadas del ArUco detectado. Estas diferencias se almacenan en las variables *delta_x* y *delta_y*. Luego, se utiliza la función *euclidean_distance* para calcular la distancia euclidiana entre el robot y el ArUco utilizando las diferencias en las coordenadas x e y.

Además de calcular la distancia, el método también calcula el ángulo alpha entre la posición actual del robot y el ArUco. El ángulo $alpha$ se calcula utilizando la función $np.arctan2(delta\_y, delta\_x) - self.current\_angle$. Aquí, $np.arctan2$ se utiliza para calcular el ángulo en radianes, y se resta el ángulo actual del robot ($self.current\_angle$) para obtener el ángulo relativo entre el robot y el ArUco.

El método también calcula una matriz de relación entre el sensor (cámara) y el estado (posición y orientación del robot). Esta matriz se almacena en la variable *relation_matrix_between_sensor_and_state* y se publica utilizando el objeto *self.relation_matrix_between_sensor_and_state_pub*.

Además de calcular la distancia y el ángulo, el método también obtiene el ángulo real alpha utilizando la función *get_alpha*. Esto se utiliza para obtener una lectura "real" del sensor utilizando el escaneo láser. Se utiliza el método *get_laser_index_from_angle* para obtener el índice correspondiente en el escaneo láser según el ángulo real alpha. Luego, se obtiene la distancia real p desde el escaneo láser utilizando el índice obtenido.

Finalmente, se publican los resultados de las mediciones estimadas y reales utilizando los objetos *self.estimated_sensor_reading_pub* y *self.real_sensor_reading_pub*.

En resumen, el método *publish_sensor_data* calcula la distancia y el ángulo entre el robot y el ArUco detectado, y publica los resultados en los tópicos de ROS correspondientes. Estos datos pueden ser utilizados para controlar el movimiento del robot en relación con el ArUco.

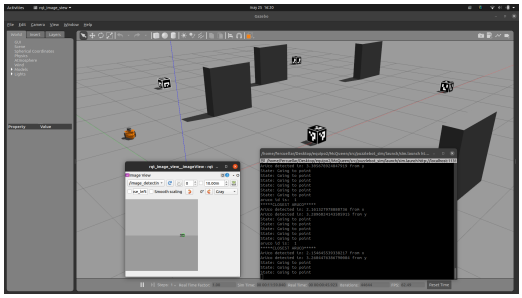

Fig. 8: Se implementó la lectura de distancia hacia los ArUcos.

b. Tópicos de Odometría con y sin filtro de Kalman:

La odometría es un método utilizado para estimar la posición y orientación de un robot móvil utilizando información de las ruedas. En el caso de un robot diferencial, como el que se representa en este código, se utilizan las velocidades de las ruedas para calcular el desplazamiento del robot.

El nodo de odometría recibe las velocidades de las ruedas derecha (wr) e izquierda (wl) a través de los suscriptores '/wr' y '/wl' respectivamente. Estas velocidades se utilizan para calcular la velocidad lineal (v) y la velocidad angular (w) del robot.

El nodo utiliza la fórmula cinemática de un robot diferencial para calcular el desplazamiento del robot en el tiempo delta t. El desplazamiento se calcula en función de las velocidades de las ruedas y los parámetros geométricos del robot, como el radio de las ruedas (*puzzlebot_wheel_rad*) y la distancia entre ruedas (*puzzlebot_wheel_to_wheel_dist*).
El nodo también realiza la estimación de la pose del robot. Utiliza la posición inicial y la orientación (*initial_x*, *initial_y*, *initial_theta*) y las actualiza en función del desplazamiento calculado. Además, se utiliza el filtro de Kalman para estimar y actualizar la covarianza de la pose del robot. El filtro de Kalman utiliza un modelo jacobiano (*puzzlebot_model_jacobian*), la velocidad lineal (v) y la velocidad angular (w) para estimar la covarianza.

La covarianza se actualiza en cada iteración utilizando la fórmula del filtro de Kalman. La covarianza resultante se utiliza para calcular la matriz de covarianza del mensaje de odometría. La matriz de covarianza se almacena en la variable *covariance_matrix* y se asigna al campo *pose.covariance* del mensaje odometría. El nodo también publica la estimación de la pose del robot en el tópico $'/kalman\_prediction\_odom'$ utilizando el publicador *puzzlebot_odom_pub*.

Además de la estimación de la pose, el nodo también tiene un suscriptor $'/kalman\_corrected\_odom'$ para recibir una corrección de la odometría. Cuando se recibe un mensaje de corrección, se actualiza la estimación de la pose del robot y se actualiza la covarianza utilizando los datos del mensaje.
En resumen, este nodo implementa la odometría para un robot diferencial utilizando las velocidades de las ruedas. Utiliza el filtro de Kalman para estimar y actualizar la pose del robot y la covarianza. También permite la corrección de la odometría mediante mensajes de corrección recibidos en el tópico $'/kalman\_corrected\_odom'$.

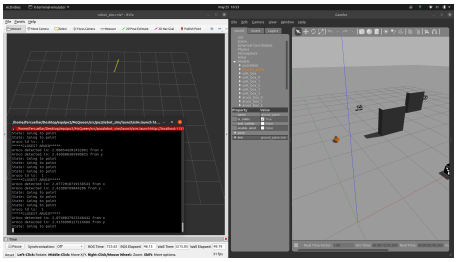

Fig. 9: Matriz de covarianza de tamaño pequeño debido a localización de ArUco y posición.

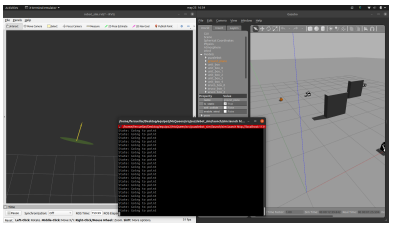

Fig. 10: Aumento de tamaño de matriz debido a uso de odometría tradicional ya qué no se localiza un punto de referencia (ArUco).

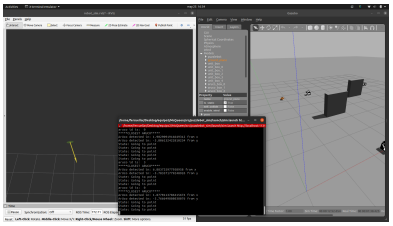

Fig. 11: Odometría corregida debido a identificación de ArUco en giro a siguiente punto.

En el transcurso de la figura 3 a 5 se ve como se tiene un incremento y decremento del área de incertidumbre al momento de perder de vista uno de los puntos de referencia, en este caso el ArUco. Al momento de perder esta ubicación la incertidumbre crece continuamente, cuando se localiza un ArUco esta vuelve a un punto mínimo ya que se reubica en el espacio

c. Bug 0

1. Detección de obstáculos por el algoritmo Bug 0:

El Bug 0 basa su funcionamiento en un proceso simple: traza una línea recta hacia su objetivo, al encontrarse con un obstáculo genera un contorno alrededor de él, el cual comienza a seguir hasta retomar el camino recto que originalmente seguía. Al encontrarse próximo a un obstáculo, se establece una distancia máxima de acercamiento para evadirlo y rodearlo. Sin embargo, existen algunos puntos que no considera, como lo son la figura del obstáculo y la distancia hacia el objeto, lo cual podría ocasionar complicaciones.

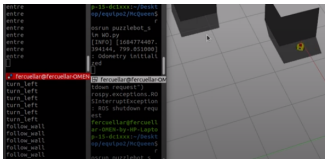

Fig. 12: Se implementó la toma de decisiones a partir de la cercanía a paredes.

Link: Detección de obstáculos con 3 paredes.

2. Funcionamiento dentro de algoritmo

Dentro de nuestro algoritmo existen diversos métodos los cuales funcionan bajo la clase de Bug 0 para el control y funcionamiento mismo, una descripción detallada de cada método se presenta a continuación.

En la sección de importaciones, se incluyen los módulos necesarios para el funcionamiento del código, como math, rospy, nav functions, numpy, y varios mensajes de ROS. La clase Bug 0 se inicializa con los parámetros *targetx*, *targety* y *wall_distance*, que representan la posición objetivo en coordenadas x e y, y la distancia que se debe mantener respecto a los obstáculos mientras se sigue una pared.
En el método *init*, se realiza la inicialización de ROS y se suscribe a los temas relevantes, como el odómetro (*/kalman_corrected_odom*) y el escaneo láser (*/scan*). También se publica en el tema */cmd_vel* para enviar comandos de velocidad al robot. El método scan callback se utiliza como una función de devolución de llamada para el tema del escaneo láser. Actualiza el atributo scan con los datos del escaneo láser recibidos. El método odom callback se utiliza como una función de devolución de llamada para el tema del odómetro. Actualiza los atributos *current_position_xy_2d* y *current_angle* con la posición y orientación actuales del robot.

El método *turn_left* se utiliza para girar el robot a la izquierda por un ángulo especificado. Toma como entrada el ángulo objetivo *p2p_target_angle* y el ángulo para girar a la izquierda *angle_to_rotate*. Ajusta la velocidad angular del robot para realizar el giro y actualiza el estado según corresponda.
El método *go_to_point_controller* controla el movimiento del robot hacia la posición objetivo. Recibe como entrada el error angular *angle_error* y el error de distancia *distance_error*. Calcula la velocidad lineal y angular del robot en función de los errores y establece los valores de velocidad en el mensaje *vel_msg*. Los métodos *get_laser_index_from_angle* y *get_mean_laser_value_at_fov* se utilizan para obtener índices y valores promedio del escaneo láser en un campo de visión (FOV) especificado.

El método *get_values_at_target* se utiliza para obtener los valores del escaneo láser en un FOV centrado en el punto objetivo.
El método *target_path_is_clear* comprueba si el camino hacia el punto objetivo está despejado. Utiliza una regla basada en la distancia a las paredes y la orientación del robot. El método *obstacle_in_front* comprueba si hay un obstáculo frente al robot. Utiliza la información del escaneo láser para determinar la distancia a los obstáculos.
El método *right_hand_rule_controller* controla el comportamiento del robot utilizando la regla de la mano derecha. Dependiendo de las condiciones, el robot puede cambiar al estado "go to point", "turn left" o "follow wall". Calcula los valores de velocidad lineal y angular en función de la posición del robot y las lecturas del escaneo láser.

d. Bases del Filtro de Kalman:

La estimación y el filtrado de datos es un proceso fundamental para la obtención de información precisa y confiable. Es así que el Filtro de Kalman permite la reducción de las perturbaciones externas, mejorando la precisión al calcular la verosimilitud de un modelo dinámico lineal, obteniendo así parámetros del modelo y finalmente sus predicciones.

Éste consta de dos etapas principales: Predicción y Corrección. En la Predicción se utiliza un modelo dinámico para predecir el estado futuro; mientras que en la Corrección se incorporan las mediciones reales para el ajuste de la estimación preliminar. Este proceso permite obtener una estimación confiable del estado del sistema.

El siguiente diagrama muestra el algoritmo correspondiente a este filtro.

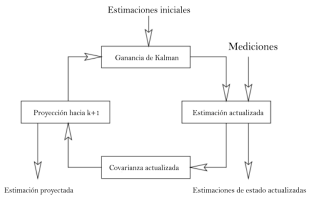

Fig. 13: Algoritmo para el filtro de Kalman.

Fig. 14: Objeto rojo identificado.

Link: Filtro de Kalman.

En el video se puede observar que se identifica un objeto color rojo, cuyas coordenadas son publicadas y actualizadas a tiempo real.

e. Integración de Modelo de Gripper a Gazebo

Con la finalidad de diseñar un sistema autónomo capaz de desplazarse sobre un entorno físico e interactuar con éste por medio de un efector final se ha implementado un Gripper que mantiene conexión e intercambio de datos a través de un microcontrolador de la familia STM32. El mismo se ha representado por medio de un modelo sólido en .STL que fue integrado exitosamente en Gazebo. Para lograrlo, se modificaron los documentos .xacro y .gazebo que generan el modelo en el simulador. A continuación explicamos brevemente las modificaciones que se realizaron tanto en los archivos URDF como dentro de los meshes.

1. Meshes:

Dentro de las carpetas Puzzlebot gazebo se encuentran los modelos en formato .STL, por lo que se integró el modelo del Gripper que utilizamos.

Link: Video de demostración.

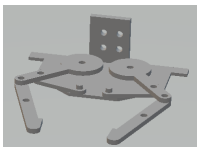

Fig. 15: Gripper integrado en una sola pieza .STL .

2. URDF:

Los documentos modificados son los archivos parameters.xacro, puzzlebot.xacro y puzzlebot.gazebo.
El archivo parameters.xacro contiene las propiedades de cada elemento, y obtiene las piezas a través del .STL de las piezas.
El archivo puzzlebot.xacro contiene los links y modifican la posición en la que se encuentran, modificando su "Roll, Pitch y Yaw" además de su localización en X Y Z. Igualmente en este documento se generan las colisiones, es decir el cómo están unidas las piezas, en este caso la unión del Gripper y el Puzzlebot.
El archivo puzzlebot.gazebo controla los colores de las piezas, la inercia, la fricción, y propiedades de movimiento de cada una.

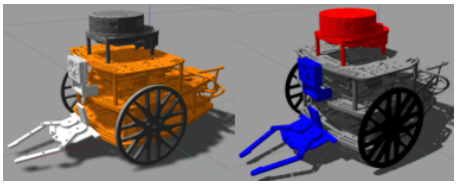

Fig. 16: Integración del Gripper al modelo y manipulación de colores.

3. Elipsoide de incertidumbre con Gripper incluido:

Si se agrega un módulo de tipo Gripper al frente del robot diferencial, esto afectaría la incertidumbre y, por lo tanto, el elipsoide de incertidumbre de la estimación de la pose del robot. La adición del Gripper introduce una fuente adicional de incertidumbre debido a la presencia de un nuevo elemento en el robot.

La incertidumbre se refiere a la falta de precisión o exactitud en la estimación de la pose del robot. En el caso de la odometría, se estima la pose del robot basándose en las mediciones de las ruedas y otros parámetros. Esta estimación tiene cierta incertidumbre debido a las limitaciones de las mediciones y los modelos utilizados.

Cuando se agrega un módulo de tipo Gripper al frente del robot, se introducen nuevos factores que pueden afectar la estimación de la pose. Por ejemplo, el Gripper puede tener un peso adicional que afecta el equilibrio del robot, o puede introducir fricción o deslizamiento en las ruedas al interactuar con objetos. Estos factores pueden alterar las mediciones de las velocidades de las ruedas y, por lo tanto, afectar la estimación de la pose.

La adición del Gripper también puede tener un impacto en los parámetros del modelo utilizado para la odometría. Por ejemplo, el radio efectivo de las ruedas puede cambiar debido a la presencia del Gripper, lo que afecta la cinemática del robot. Además, el comportamiento dinámico del robot puede verse alterado debido a la interacción con objetos a través del Gripper.

En términos del elipsoide de incertidumbre, este representa una región en el espacio que describe la incertidumbre en la pose estimada del robot. Si se agrega un Gripper, es probable que la forma y el tamaño del elipsoide de incertidumbre cambien. El nuevo elipsoide puede ser más grande o tener una forma diferente debido a la mayor incertidumbre generada por el Gripper.

El cambio en el elipsoide de incertidumbre dependerá de la magnitud de la incertidumbre generada por el Gripper y cómo esta incertidumbre se propaga a través del modelo de odometría y el filtro de Kalman utilizado para estimar la pose del robot. Si el Gripper tiene un impacto significativo en las mediciones y en el comportamiento del robot, es probable que el elipsoide de incertidumbre se expanda y se distorsione en la dirección correspondiente al Gripper.

En resumen, la adición de un módulo de tipo Gripper al frente del robot diferencial introduce incertidumbre adicional en la estimación de la pose. Esto afectará el elipsoide de incertidumbre, que representa la región en el espacio que describe la incertidumbre en la pose estimada del robot. El tamaño y la forma del elipsoide de incertidumbre cambiarán para reflejar la mayor incertidumbre introducida por el Gripper y su impacto en las mediciones y el modelo de odometría.

f. Resultados

Videos de funcionamientos y pruebas exitosas con diferentes puntos de localización, mundos y obstáculos para la solución de reto final de Manchester Robotics
Link: Bug 0 + ArUcos(Identificación) - 1 Punto.
Link: Bug 0 + ArUcos(Identificación) - 2 Punto.
Link: Bug 0 + ArUcos(Identificación) - 3 Punto.
Link: Bug 0 + ArUcos(Identificación) - 4 Punto.

Resultado final de reto por Manchester Robotics

Código integrado en simulación con filtro de Kalman, odometría, identificación de ArUcos y posición. Link: Bug 0 + ArUcos(Odometría, identificación de distancia) - 4 Punto.

IV. Reto de bloque, implementación física

a. Diagramas de implementación

1. Arquitectura de Hardware

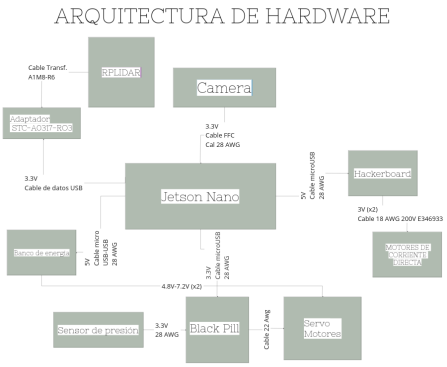

Fig. 17: Diagrama de Arquitectura de Hardware

Un diagrama de arquitectura de hardware nos permite visualizar y analizar la estructura física del sistema, mostrando las interconexiones y componentes clave. En este caso, se representarían los componentes de hardware relevantes, como la Jetson Nano, la STM32 Blackpill, las cámaras y el LiDAR. Esto nos ayuda a comprender la forma en que estos dispositivos están conectados y cómo interactúan entre sí, proporcionando una visión general de la configuración física del sistema.

2. Arquitectura de Datos

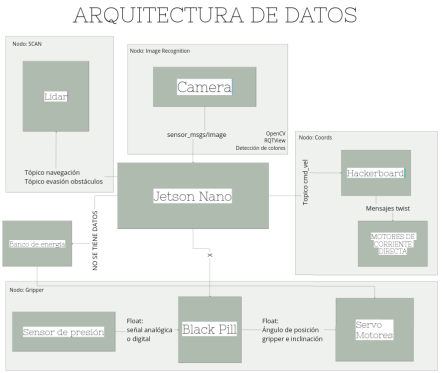

Fig. 18: Diagrama de Arquitectura de Datos

Por otro lado, un diagrama de arquitectura de datos nos permite comprender la forma en que los datos se transmiten y procesan dentro del sistema. En este contexto, el diagrama mostraría cómo los datos se capturan y se envían desde los módulos de cámara y LiDAR hacia la Jetson Nano y la STM32 Blackpill. También se representaría cómo se procesan y se utilizan esos datos para controlar el robot diferencial. Además, el diagrama podría indicar cómo se transmiten los comandos de control desde la Jetson Nano hacia la STM32 Blackpill.
En conjunto, estos diagramas nos brindan una visión integral del sistema, permitiéndonos comprender la estructura física y el flujo de datos en detalle. Esto facilita la identificación de posibles cuellos de botella, puntos de fallo o áreas de mejora en el sistema, y también nos ayuda a tener una visión clara de la interacción entre los diferentes componentes. En última instancia, el uso de estos diagramas contribuye a una comprensión más completa y precisa del sistema en su conjunto.

b. Navegación por Bug 2

El algoritmo Bug 2 es un enfoque para la navegación en entornos desconocidos o parcialmente conocidos que implica seguir una línea recta hacia una meta y sortear obstáculos cuando se encuentran en el camino. A continuación las diferentes funcionalidades de nuestro código, proceso para que este funcione y métodos utilizados:

1. En el constructor de la clase Bug 2, se inicializan varias variables y se definen los parámetros de control y meta. Se suscriben a los tópicos /scan y /odom para obtener datos del escáner láser y la odometría del robot.

2. En la función scanCallback, se obtienen los valores de distancia medidos por el escáner láser en las direcciones frontal y derecha del robot. Estos valores se utilizan más adelante para detectar obstáculos.

3. En la función getPuzzlePos, se obtiene la posición actual del robot a partir de los datos de odometría. El ángulo de orientación del robot también se calcula utilizando la función getRotation.

4. La función error calcula el error de posición y distancia a la meta utilizando la posición actual del robot y la posición de la meta.

5. La función index se utiliza para convertir un ángulo en un índice correspondiente en el arreglo de valores de rango del escáner láser. Esto se utiliza posteriormente para obtener las distancias medidas en ángulos específicos.

6. La función wallDirection calcula el ángulo de error entre el robot y una pared cercana utilizando los valores de distancia medidos por el escáner láser. Este ángulo se utiliza para seguir el contorno de un obstáculo.

7. En la función followObs, se implementa el seguimiento del contorno del obstáculo. Se utilizan constantes de control para ajustar el ángulo del robot en relación con la pared y mantener una distancia constante del obstáculo.

8. La función lineM calcula la distancia entre la posición actual del robot y una línea recta definida por la posición de la meta y una pendiente (m) y un punto (b) predeterminados. Esta distancia se utiliza para detectar si el robot ha encontrado la línea recta hacia la meta.

9. En la función main, se realiza el bucle principal de control del robot. Se actualizan la posición y el ángulo del robot, se calcula el error de posición y distancia, y se realiza la navegación hacia la meta o el seguimiento del contorno del obstáculo según el estado actual del robot.

Demostración en video de Bug 2: Evasión de 2 obstáculos mediante Bug 2.

c. Localización de ArUco

Para la localización de los ArUcos se basa en la detección mediante la librería de cv2 en el método de ArUco cv.aruco y mediante la función de obtener esquinas y IDs para el listado de marcadores

```python
# Get the image size
image_height, image_width, _ = cv_image.shape

# Convert the image to grayscale
gray = cv2.cvtColor(cv_image, cv2.COLOR_BGR2GRAY)

# Define the dictionary of ArUco markers and parameters
aruco_dict = cv2.aruco.Dictionary_get(cv2.aruco.DICT_4X4_250)
parameters = cv2.aruco.DetectorParameters_create()

# Detect ArUco markers
corners, ids, _ = cv2.aruco.detectMarkers(gray, aruco_dict, parameters=parameters)
```

Fig. 19: Identificación de ArUco 4x4.

1. Control para seguimiento de ArUco

Dentro de esta abstracción de código el control se basa en una medida proporcional con el error del ángulo y un control PID para así tener una velocidad angular variable en base al error. Se tiene limitada la velocidad angular para tener movimientos controlados y no agresivos los cuales puedan repercutir en el voltaje.

```python
        # PID control
        # Calculate the proportional term
        proportional = Kp * error_angular
        # Calculate the integral term
        integral += Ki * error_angular
        # Calculate the derivative term
        derivative = Kd * (error_angular - prev_error)
        # Calculate the angular velocity using PID control
        angular_velocity = proportional + integral + derivative
        # Limit angular velocity
        angular_velocity = min(MAX_ANGULAR_VELOCITY, max(-MAX_ANGULAR_VELOCITY, angular_velocity))
        # Store the current error for the next iteration
        prev_error = error_angular
```

Fig. 20: Ajuste de velocidad angular mediante control PID.

La corrección angular con 40 píxeles de ajuste se realiza una vez qué se localiza el ArUco en visión, ya si se incrementa este error una vez qué empiece la velocidad lineal este cambia de valor dentro del camino

```python
else:
    # If the marker is not centered, rotate with a proportional angular velocity
    if abs(error_angular) > 40:
        #twist_cmd.linear.x = 0.0
        twist_cmd.angular.z = -angular_velocity
    else:
        # If the marker is centered, move forward with a constant linear velocity
        twist_cmd.linear.x = LINEAR_VELOCITY
        twist_cmd.angular.z = -angular_velocity

# Publish the velocity command
cmd_vel_pub.publish(twist_cmd)

# Print the ID of the current marker
print("Current marker ID:", current_marker.id)
```

Fig. 21: Corrección angular con 40 píxeles de ajuste.

En el dado caso de que no se detecte el ArUco objetivo este empezará a girar en contra del reloj para la identificación de el ArUco objetivo con una velocidad de 0.05 radianes por segundo

2. Detección de cercanía hacia ArUco

Para la detección de cercanía se utiliza las proporcionalidad en porcentaje que toma el ArUco con respecto a la imagen

```python
else:
    # If no marker is detected or the detected marker has a different ID, stop the robot
    twist_cmd.linear.x = 0.0
    twist_cmd.angular.z = 0.05
    cmd_vel_pub.publish(twist_cmd)

rate.sleep()
```

Fig. 22: Función de girar para identificar si es que no encuentra un ArUco en vista.

total, esta está calibrada para tener la distancia necesaria para tomar el objeto.

```python
# If the marker area covers more than 10% of the image, stop the robot
if marker_area > 0.08 * image_width * image_height and (current_marker.id == 0 or current_marker.id == 2 or current_marker.id == 1):
    # Publish "agarrar" message
    gripper_jetson_pub.publish("agarrar")

    # Wait until "cerrado" message is received
    while not received_cerrado_message:
        rate.sleep()

    # Reset received_cerrado_message flag
    received_cerrado_message = False

    twist_cmd.linear.x = 0.0
    twist_cmd.angular.z = 0.0
    cmd_vel_pub.publish(twist_cmd)
    return  # Exit the function to search for another ArUco
```

Fig. 23: Condicional para determinar el área del ArUco dentro de la imagen.

3. Filtro de Kalman y uso en seguimiento de ArUcos

El filtro de Kalman es un algoritmo utilizado para estimar el estado de un sistema basado en mediciones imperfectas y ruidosas. Combina información de las mediciones actuales con estimaciones anteriores del estado del sistema para obtener una estimación más precisa del estado actual.

4. Pasos para la implementación

1. Se crea una instancia del filtro de Kalman utilizando la clase KalmanFilter de la biblioteca filterpy.kalman.
2. Se define la matriz de transición de estado F que describe cómo evoluciona el estado del sistema con el tiempo. En este caso, se utiliza una matriz de dimensión 4x4 para representar las coordenadas x e y del marcador y sus velocidades correspondientes.
3. Se define la matriz de observación H que mapea el estado del sistema al espacio de observación. En este caso, se utiliza una matriz de dimensión 2x4 para relacionar las coordenadas x e y del marcador con las coordenadas observadas en la imagen.
4. Se ajusta la covarianza de la incertidumbre inicial del estado del sistema mediante la multiplicación de la matriz de covarianza P por un factor de 10. Esto se realiza en la línea kf.P *= 10.
5. Se define la matriz de covarianza de ruido de medición R que representa el ruido en las mediciones. En este caso, se utiliza una matriz diagonal de dimensión 2x2 con un valor de 0.5 en la diagonal.
6. En la función image-callback, se aplica el filtro de Kalman para suavizar la posición del marcador. La posición del marcador se mide como el centro del marcador detectado en la imagen y se utiliza como entrada para el filtro de Kalman. Después de la predicción y actualización del filtro, se obtiene una estimación suavizada de la posición del marcador en las variables current-marker.x y current-marker.y.

```
# Create a Kalman filter for tracking the marker position
kf = KalmanFilter(dim_x=4, dim_z=2)
kf.F = np.array([[1, 0, 1, 0],
                 [0, 1, 0, 1],
                 [0, 0, 1, 0],
                 [0, 0, 0, 1]])
kf.H = np.array([[1, 0, 0, 0],
                 [0, 1, 0, 0]])
kf.P *= 10  # Initial uncertainty covariance
kf.R = np.diag([0.5, 0.5])  # Measurement noise covariance
```

Fig. 24: Creación de matrices y valores iniciales para las bases del filtro de Kalman.

```
# Apply Kalman filter to smooth the marker position
measurement = np.array([[marker_center_x], [marker_center_y]])
kf.predict()
kf.update(measurement)
current_marker.x = kf.x[0, 0]
current_marker.y = kf.x[1, 0]
```

Fig. 25: Uso de valores predichos y actualización de valores sobre el marcador del ArUco.

El filtro de Kalman ayuda en el seguimiento de ArUco al proporcionar una estimación suavizada y más precisa de la posición del marcador. Esto es especialmente útil cuando las mediciones son ruidosas o cuando se producen pérdidas ocasionales del marcador. Al reducir el ruido y suavizar las estimaciones, el filtro de Kalman permite que el control PID tome decisiones más estables y precisas para el seguimiento del marcador.

En conjunto con el control PID implementado en la función control-loop, el filtro de Kalman ayuda a controlar la velocidad angular del robot para mantener el marcador centrado en la imagen. El error angular entre la posición del marcador y el centro de la imagen se calcula y se utiliza como entrada para el control PID. El control PID calcula la velocidad angular deseada en función del error angular y los coeficientes de ganancia proporcionados (Kp, Ki, Kd). Esta velocidad angular se limita dentro de ciertos límites máximos y se utiliza junto con una velocidad lineal constante para generar los comandos de velocidad enviados al robot.

En resumen, el filtro de Kalman se utiliza para estimar y suavizar la posición del marcador, mientras que el control PID se encarga de generar los comandos de velocidad adecuados para mantener el marcador centrado en la imagen. Esto permite un seguimiento más preciso y estable del marcador ArUco.

Demostración en video de filtro de Kalman para seguimiento de ArUco: Seguimiento a ArUco con ID:0 con PID y filtro de Kalman.

d. Pruebas unitarias para la comunicación y espera de respuesta en tópicos

Se han llevado a cabo experimentos con el propósito de establecer un marco de comunicación preestablecido a través de un sistema de estados. Este sistema se ha diseñado con zonas de recolección, donde se emplea el reconocimiento de ArUcos, y zonas de entrega que siguen un enfoque similar. Con el fin de lograr este objetivo, se han desarrollado tópicos específicos, así como un mecanismo de publicación y suscripción. Estos elementos permiten el intercambio de mensajes y la retroalimentación a través de los tópicos, garantizando así la continuidad del programa y su estabilidad.

```
    rospy.Subscriber("STMToJetson", String, self.callback_STM)

    #STM - JETSON
    self.message_STM_Jetson = ''

def callback_STM(self, data):
    rospy.loginfo("Mensaje recibido de STM: %s", data.data)
    self.message_STM_Jetson = data.data #obtener mensaje de la stm
```

Fig. 26: Creación suscriptores y publicadores, STMToJetson y JetsonToSTM.

Una vez obtenido este enfoque, es posible establecer estados de espera para continuar con el algoritmo una vez que se haya cumplido la condición requerida. Dado que se ha determinado que siempre se realizará una recolección inicial de una caja, se implementa un intercalador de acciones que determina si se debe publicar un mensaje para recolectar o soltar dicha caja. Al mismo tiempo, se actualizan los valores de las variables temporales necesarias para ejecutar las acciones, asegurando así que cada una de ellas se realice de manera precisa y sin duplicaciones.

```
for id in self.target_aruco_ids:
    print("Following ArUco with ID:", id)
    self.control_loop(id)
    if close_gripper:
        # Enviar mensaje "abre gripper" en los ArUcos pares
        self.JetsonToSTM_pub.publish("abre gripper")
        self.gripper_status_received = False  # Reiniciar la variable para esperar la respuesta del gripper
        while self.message_STM_Jetson != "abierto":
            print("esperando")
            rate.sleep()
        #hacer el robot para atras1 segundo
        start_time = time.time()
        while time.time() - start_time < 0.2:
            print(time.time() - start_time)
            self.vw.linear.x=-0.05
            self.vw.angular.z=0
            self.cmd_vel_pub.publish(self.vw)

    else:
        # Enviar mensaje "cierra gripper" solo en los ArUcos impares
        self.JetsonToSTM_pub.publish("cierra gripper")
        self.gripper_status_received = False  # Reiniciar la variable para esperar la respuesta del gripper
        while self.message_STM_Jetson != "cerrado":
            print("esperando")
            rate.sleep()

    self.message_STM_Jetson = "disponible"

    close_gripper = not close_gripper  # Alternar el estado del gripper para la siguiente iteracion
```

Fig. 27: Creación de estados de espera/abierto/cerrado para la continuidad de código.

Demostración en video de espera de estados: Simulación de respuesta a estados para la continuidad de programa.

e. Comunicación con Arduino para interacción con Gripper

La comunicación entre dispositivos electrónicos es fundamental en muchos proyectos y aplicaciones. Una forma común de lograr esto es utilizando la biblioteca ROSserial junto con plataformas como Arduino y Jetson Nano. En este ensayo, exploraremos qué es ROSserial, cómo se puede utilizar para establecer la comunicación entre Arduino y Jetson Nano, y proporcionaré un breve resumen de un código de ejemplo que utiliza ROSserial para controlar servomotores.

ROSSerial es una biblioteca de ROS (Robot Operating System) que permite la comunicación serie entre un microcontrolador, como Arduino, y una computadora, como Jetson Nano, utilizando el protocolo de mensajes de ROS. Esta biblioteca permite enviar y recibir mensajes ROS entre los dos dispositivos, lo que facilita la integración de hardware y software en aplicaciones robóticas y de IoT.

Para utilizar ROSserial en una Jetson Nano y un Arduino,

es necesario tener instalado el entorno de desarrollo de ROS en la Jetson Nano y cargar el firmware de ROSserial en el Arduino. Una vez configurados, el Arduino se conecta físicamente a la Jetson Nano a través de un puerto serie, como USB. Luego, se pueden crear nodos ROS en la Jetson Nano y suscribirse o publicar mensajes a través del Arduino utilizando ROSserial.

El codigo de implementación y funcionalidad esta de igual manera en el github, en la parte de resultados

f. Implementación de STM32 en Sistemas de Control de Motores

1. Mapa de impacto

De igual manera se le ha brindado un enfoque de aplicación adicional al desarrollo previamente planteado, donde el sistema Puzzlebot actuará como sorteador de frutas configurado para identificar y desechar frutas en mal estado dado un lote de cosecha.

Para el análisis de sus implicaciones se ha creado un mapa de impacto y contexto de uso, considerando al usuario, entorno local, servicio de suministros y sistema de soporte. Al considerar el entorno local, se deben tener en cuenta diversos factores. En primer lugar, la posición de los objetos que se deben clasificar es fundamental, además de que la iluminación del entorno puede influir en la visión y percepción de los colores, especialmente si la clasificación se basa en esta variable. Por otro lado, es crucial que el robot tenga la capacidad de evitar colisiones con obstáculos presentes en su entorno. En los suministros se considera la materia prima, contenedores para las frutas y fuente de poder para ejecutar el proceso con el robot. Para llevar un monitoreo y sistema de soporte, se considera el estatus actual del sistema, teniendo un sistema de retroalimentación para poder realizar actualizaciones en caso de ser necesario. Finalmente, el usuario que maneja el robot podrá confirmar el proceso de clasificación y selección, proveyéndole un manual de usuario. Este planteamiento puede ser observado en el diagrama 28.

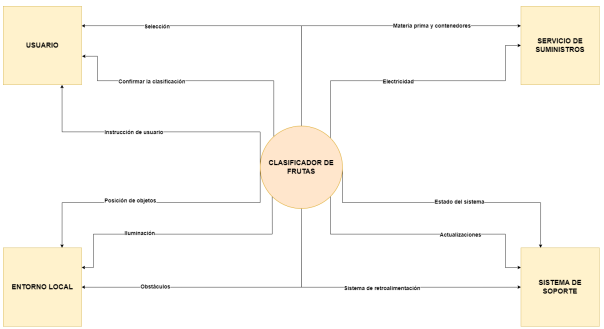

Fig. 28: Mapa de impacto y contexto de uso.

Con el propósito de tener una idea específica del funcionamiento a desarrollar, se creó un mapa con 4 campos que se desglosan se la siguiente manera:

Fig. 29: Consideraciones del sistema.

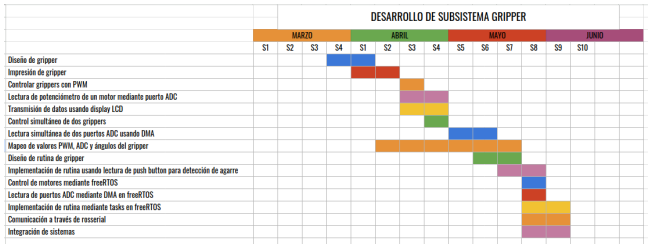

Fig. 30: Distribución de tareas del proyecto.

g. Tareas Subsistema Gripper

1. Requerimientos del proyecto

Las placas de desarrollo STM32 son ampliamente utilizadas en el diseño de sistemas embebidos debido a su versatilidad, potencia y amplia gama de características y periféricos integrados. Entre las variadas utilidades con las que cuenta se puede mencionar la potencia de procesamiento, pues proporciona un alto rendimiento y es capaz de ejecutar tareas complejas y algoritmos en tiempo real. Cuenta con una gran variedad de periféricos integrados, como puertos GPIO, ADC, DAC, generadores de PWM e interfaces de comunicación SPI, I2C y UART, entre otros. Las tarjetas STM32 son compatibles con diferentes interfaces de desarrollo, lo que facilita la reutilización de código y la integración con otros dispositivos y sistemas existentes.

Se ofrece una amplia gama de microcontroladores STM32 con diferentes capacidades de memoria, clock rate, periféricos y opciones de conectividad. Esto permite seleccionar la tarjeta más adecuada para las funciones que se planean ejecutar, ya sean aplicaciones de bajo consumo de energía, de alto rendimiento, etc.

El microcontrolador utilizado en este proyecto es el modelo STM32F411CEU6, con 32 bits basado en ARMCortex-M4 con una velocidad de reloj de hasta 100MHz. Cuenta con las siguientes especificaciones:

Aspecto	Descripción
Memoria flash	512 kB
Memoria RAM	128 kB
GPIO:	82
ADC	3 de 12 bits
Temporizadores	14
Comunicación serial	3 puertos USART, 2 puertos SPI, 3 puertos I2C
Comunicación USB	1 puerto USB 2.0
Periféricos adicionales	DMA, RTC, WDG, CRC1, RNG

Fig. 31: Especificaciones de modelo STM32F411CEU6

Se optó por este modelo de STM principalmente por su potencia de procesamiento, pues permite una amplia capacidad de almacenamiento para el código del programa, también debido a su conectividad y eficiencia energética.

h. Implementación de STM32

1. Declaración de puertos y lectura de Voltaje

Se realizó el pinout y el cableado necesario para establecer la comunicación y controlar la posición angular. Igualmente se añadió un display digital para mostrar el estatus del subsistema del manipulador.

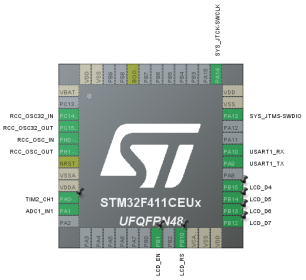

Fig. 32: Programación de puertos en el microcontrolador STM32.

Para lograr la lectura fue necesario programar el microcontrolador de la manera específica en que se muestra el diagrama. Se declararon USART los puertos PA9 y PA10, con la finalidad de permitir la comunicación síncrona con el puerto ADC.

Para los puertos Reset and Clock Control (RCC) fueron definidos PH0, PH1, PC14 y PC15. La Inicialización de éstos permite establecer la frecuencia del sistema y la habilitación o des-habilitación los relojes de los periféricos.

Como puertos SYS fueron designados PA13 y PA14, en este caso ambos fueron utilizados para la interfaz de depuración y programación Serial Wire Debug (SWD).

Por una parte, el PA13 SWDIO funciona como señal bidireccional entre los dispositivos. En este caso nos permite controlar y monitorear el estado del microcontrolador durante la ejecución. Por otra parte, el puerto A14 (SWCLK) se configuró como una señal de reloj que sincroniza la transferencia de datos entre el depurador y el microcontrolador.

Los puertos definidos para el LCD fueron PB1, PB10, PB12, PB13, PB14 y PB15, los cuales permiten una comunicación estable con la pantalla para monitorear el voltaje recibido por el motor.

Finalmente, el puerto TIM2 (PA0) es un periférico de temporización; y ADC1 (PA1) es el convertidor analógico-digital, el cual recibe la señal del motor para transmitir los datos a la pantalla LCD.

2. Casos de uso
Estados Gripper y articulación

El funcionamiento general del Gripper se basa en dos grados de libertad. El primero maneja los grados de giro para asegurar la toma del objeto e impedir que caiga al suelo en el momento de la navegación. Éste tiene una señal PWM de 83 y ADC entre 800-900 cuando se encuentra en 0°, al momento de girar 90° la señal PWM sube a 197 y ADC entre 2000-2200. La segunda articulación se encarga de abrir y cerrar las tenazas para sostener el objeto. Los estados iniciales o punto cero es cuando la articulación se encuentra en 0° y las tenazas abiertas.

Se establecieron tres estados del Gripper y articulación dependiendo de su grado de apertura y la señal detectada por el sensor de presión, sustituido por un botón push que indica cuando el Gripper está tomando el objeto y cuando no.

1. **Caso 0. Tenazas cerradas y objeto sostenido:** Una vez que el cubo es identificado por medio del ArUco y encontrarse a una distancia cercana, procede a cerrar las tenazas. Al mismo tiempo, el botón push tiene un estado booleano que asegura cuando el objeto está siendo tomado o no. Se tiene una señal de PWM de 131, ADC entre 1300-1400, y un estado de botón 1. Se procede a rotar el Gripper 90°.

2. **Caso 1. Tenazas cerradas y objeto no sostenido:** Pasa a este estado cuando el robot identifica el objeto y cierra las tenazas, sin embargo, el botón no recibe una señal de estar sosteniéndolo. Se procede a abrir el Gripper y cambiar de estado al caso 2.

3. **Caso 2. Tenazas abiertas:** Se encuentra en este estado cuando el robot está en proceso de búsqueda de los ArUcos o cuando se llega a la ubicación destino de descarga. Se recibe una señal PWM de 60 y ADC entre 500-800. Se procede a cerrar el Gripper, en caso que el objeto sea sostenido, éste se abre, en caso contrario se cierra.

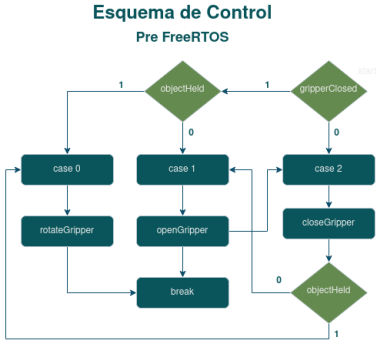

Fig. 33: Esquema de control previo a FreeRTOS.

El diagrama de flujo de la fig. 33 muestra la lógica del comportamiento esperado para la la implementación de

FreeRTOS. El cambio de estados se basa principalmente el valor de variables como objectHeld y gripperClosed.

i. Display en pantalla LCD

Con la finalidad de observar el monitoreo del proceso, el valor de los estados son actualizados a tiempo real en una pantalla LCD. La función printLCD toma los parámetros recibidos del 'Gripper' y 'articulación', se especifican sus tamaños y se recibe la información del estado actual.

```
void printLCD(int gripper, int articulacion)
{
    char bufferG[10];
    char buffer_Gripper[4];
    char bufferA[14];
    char buffer_Art[2];

    //Gripper
    lcd_setCursor(&lcd, 0, 0);
    sprintf(bufferG, "Gripper: ");
    lcd_print(&lcd, bufferG);

    //Estado
    lcd_setCursor(&lcd, 9, 0);
    sprintf(buffer_Gripper, "%u", gripper);
    lcd_print(&lcd, buffer_Gripper);

    sprintf(buffer_Gripper, "   ");
    lcd_print(&lcd, buffer_Gripper);
```

Fig. 34: Envío de datos a LCD.

j. Implementación de FreeRTOS

Como sabemos, la implementación de sistemas embebidos nos permite optimizar la utilización de recursos para el desarrollo de tareas específicas y preestablecidas. No obstante, esto no quiere decir que por sí mismos no requieran igualmente de una administración eficiente de recursos y tareas para garantizar su correcto funcionamiento. En este contexto, FreeRTOS se destaca como un sistema operativo en tiempo real de código abierto, diseñado especialmente para aplicaciones de sistemas embebidos como la que nos encontramos desarrollando con la tarjeta de distribución STM32.

FreeRTOS fue desarrollado por Richard Barry y su objetivo es brindar un a solución ligera y de alto rendimiento para sistemas embebidos con restricciones de recursos. Se encuentra basado en un modelo de programación de tareas y ofrece una amplia gama de funcionalidades, por lo que puede ser interpretado como una librería que provee capacidades multi-tasking para aplicaciones de control de hardware. FreeRTOS nos brinda un conjunto de documentos desarrollados y compilados en lenguaje C, donde algunos de los mismos son específicos para un puerto, mientras que otros son específicos para un solo puerto.

Bajo este contexto, con la finalidad de diseñar una rutina que permitiera la comunicación en tiempo real con la computadora Jetson Nano, se realizó la implementación de FreeRTOS para el diseño y ejecución de tareas, mismas que están estructuradas con el objetivo de brindar un entorno óptimo de ejecución de los comandos enviados por el Puzzlebot cuando los objetos de interés son identificados. Las tareas desarrolladas son las siguientes:

1. **receiveJetson.** Tarea definida con prioridad Realtime con la finalidad de ser la encargada de interpretar los datos de comunicación recibidos por la tarjeta STM32 desde la Jetson Nano. Su proceso de interpretación es basado en el diagrama 1, implementado por medio de un switch case que lee los dos bytes del buffer recibido de manera serial, de acuerdo con los cuales identifica la rutina a ejecutar y activa banderas para permitir que el resto de las tareas las interpreten e identifiquen si deben llevarse a cabo o no. Las banderas usadas son gripperFlag, artFlag y stopFlag. El código implementado se observa en la figura 35.

```
/* USER CODE END Header_StartReceiveJetson */
void StartReceiveJetson(void *argument)
{
  /* USER CODE BEGIN StartReceiveJetson */
  /* Infinite loop */
  for(;;)
  {
    read();
    switch (comm[0]){
        case 1:
            if (comm[1]==1)
              {
                gripperFlag = 1;
              }
            else if (comm[1]==2)
              {
                gripperFlag = 2;
              }
            else
              {
                gripperFlag = 0;
              }
            break;
        case 2:
            if (comm[1]==1)
              {
                artFlag = 1;
              }
            else if (comm[1]==2)
              {
                artFlag = 2;
              }
            else
              {
                artFlag = 0;
              }
            break;
        case 3:
              stopFlag = 1;
              break;
       default:
              gripperFlag = 0;
              artFlag = 0;
              stopFlag = 0;
              break;
      }
      osDelay(20000);
  }
  /* USER CODE END StartReceiveJetson */
}
```

Fig. 35: Tarea receiveJetson.

2. **emStop.** Declarada con prioridad AboveNormal, pues es crucial que sea interpretada tan pronto como termina la ejecución de la tarea receiveJetson. El motivo de esta decisión se basa en el conocimiento de que, en caso de ser necesario que el sistema físico se detenga debido a la presencia de algún error complejo en el sistema principal, es importante que su rutina se ejecute previo a que alguna otra sea implementada. Esta tarea desactivará todas las banderas, de manera que el resto de las tareas identifiquen que sus rutinas no deben ser llevadas a cabo. Una vez que ha sido ejecutada activa la bandera stopOK. El código implementado se observa en la figura 36.

```
/* USER CODE END Header_StartEmStop */
void StartEmStop(void *argument)
{
  /* USER CODE BEGIN StartEmStop */
  /* Infinite loop */
  for(;;)
  {
      if(stopFlag == 1){
          gripperFlag = 0;
          artFlag = 0;
          stopFlag = 0;
      }
    osDelay(1);
  }
  /* USER CODE END StartEmStop */
}
```

Fig. 36: Tarea emStop.

3. **artTask.** Esta tarea se define con una prioridad Normal1, ya que leerá las banderas previamente activadas y se encargará de posicionar adecuadamente la articulación de una manera específica en caso de que la tarea receiveJetson identifique esta orden enviada por la com-

putadora Nano Jetson. La lectura se realiza con la bandera artFlag, donde los valores declarables son 1 para subir y 2 para bajar. Una vez que ha sido ejecutada activa la bandera artOK. El código implementado se observa en la figura 37.

```
/* USER CODE END Header_StartArtTask */
void StartArtTask(void *argument)
{
  /* USER CODE BEGIN StartArtTask */
  /* Infinite loop */
  for(;;)
  {
    if(artFlag == 1)
    {
      TIM2->CCR2 = artUp;
      osDelay(50);
      artFlag = 0;
      artOK = 1;
    }
    else if (artFlag == 2)
    {
      TIM2->CCR2 = artDown;
      osDelay(50);
      artFlag = 0;
      artOK = 1;
    }
    osDelay(1);
  }
  /* USER CODE END StartArtTask */
}
```

Fig. 37: Tarea artTask.

4. **gripperTask.** Declarada con prioridad Normal2, con el objetivo de que su rutina sea ejecutada posterior a la rutina de la articulación, en caso de que lo mismo sea indicado por la tarea receiveJetson. La bandera asociada en este caso es gripperFlag, donde los valores le indican si la solicitud recibida es abrir (1) o cerrar (2) las tenazas del gripper. Una vez que ha sido ejecutada activa la bandera gripperOK. El código implementado se observa en la figura 38.

```
/* USER CODE END Header_StartGripperTask */
void StartGripperTask(void *argument)
{
  /* USER CODE BEGIN 5 */
  /* Infinite loop */
  for(;;)
  {
    if(gripperFlag == 1)
    {
      TIM2->CCR1 = gripperOpen;
      osDelay(50);
      gripperFlag = 0;
      gripperOK = 1;
    }
    else if (gripperFlag == 2)
    {
      TIM2->CCR1 = gripperClose;
      osDelay(2000);
      gripperFlag = 0;
      gripperOK = 1;
    }
    osDelay(100);
  }
  /* USER CODE END 5 */
}
```

Fig. 38: Tarea gripperTask.

5. **checkButton.** Tarea declarada con prioridad BelowNormal, su intención es leer de manera constante si entre las tenazas está sostenido un objeto, esto con la finalidad de que la Jetson Nano, una vez que ha recibido la información, decida el siguiente paso a ejecutar. Los valores brindados a la bandera objectHeld actúan de manera booleana, indicando si el objeto ha sido sostenido (1) o no (0). El código implementado se observa en la figura 39.

```
/* USER CODE END Header_StartCheckButton */
void StartCheckButton(void *argument)
{
  /* USER CODE BEGIN StartCheckButton */
  /* Infinite loop */
  for(;;)
  {
    if(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_2) == GPIO_PIN_RESET){
      objectHeld = 0;
    }
    else{
      objectHeld = 1;
    }
    osDelay(1);
  }
  /* USER CODE END StartCheckButton */
}
```

Fig. 39: Tarea checkButton.

6. **transmitJetson.** Última tarea en ejecutarse, por lo que se le asocia una prioridad Low. Es la encargada de realizar lecturas constantes de las banderas de confirmación (stopOK, artOK, gripperOK, objectHeld) y, con base en la interpretación de las mismas transmite el estado actual del gripper a la unidad Jetson Nano. Esta rutina almacenará los valores actuales de todas las banderas de confirmación en un arreglo de cuatro espacios y posteriormente las reiniciará todas. El código implementado se observa en la figura 40.

```
/* USER CODE END Header_StartTransmitJEtson */
void StartTransmitJEtson(void *argument)
{
  /* USER CODE BEGIN StartTransmitJEtson */
  /* Infinite loop */
  for(;;)
  {
    if (gripperOK == 1)
    {
      okFlags[0]= gripperOK;
      gripperOK = 0;
    }
    else
    {
      okFlags[0]= 0;
    }
    if (artOK == 1)
    {
      okFlags[1]= artOK;
      artOK = 0;
    }
    else
    {
      okFlags[1]= 0;
    }
    if (objectHeld == 1)
    {
      okFlags[2]= objectHeld;
    }
    else
    {-
      okFlags[2]= 0;
    }
    if (stopOK == 1)
    {
      okFlags[3]= stopOK;
      stopOK = 0;
    }
    else
    {
      okFlags[3]= 0;
    }
    osDelay(20100);
  }
  /* USER CODE END StartTransmitJEtson */
```

Fig. 40: Tarea transmitJetson.

Todas estas tareas se comunican de manera piramidal por medio de las banderas declaradas y modificadas durante el proceso de ejecución del código. Esta comunicación es interpretada de manera visual en el diagrama 41.

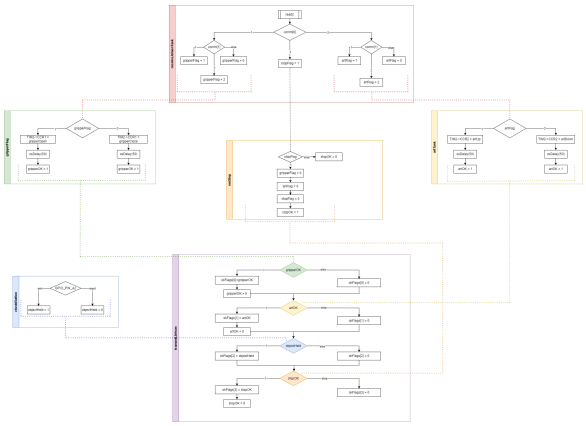

Fig. 41: Esquema de comunicación entre tareas.

k. Comunicación entre Jetson Nano y STM32 a través de rosserial

1. Implementación de FreeRTOS Comunicación STM-Jetson:

La comunicación entre la STM y Jetson se da de manera bidireccional a través del envío de paquetes de 2 bytes. La Jetson envía señales relacionadas con la detección visual de los ArUcos y navegación hasta su dirección destino, para proceder a la toma de los objetos, proceso que realiza la STM con la definición de tasks para cada acción.

La estructura se divide en dos bloques, el envío de las acciones y el de los ángulos, los cuales son identificados con el primer byte del buffer. Dentro del bloque de las acciones se establecen tanto los estados del Gripper (dirección 0xE0) como de la articulación de giro (dirección 0xE1), donde el segundo byte especifica el 'abrir'(0x01) o 'cerrar'(0x02) de Gripper y el 'subir'(0x01) o 'bajar'(0x02) de la articulación.

El segundo bloque, cumple la función de proporcionar una referencia para transmitir los datos actuales a la Jetson basándose en los ángulos recibidos Estos datos se utilizarán como base de referencia para el posterior proceso interno de toma de decisiones. Para ello, se emplean dos bytes para representar la dirección del primer bloque, conocido como "gripper". El primer byte tiene un valor de 0xC0, mientras que el segundo byte corresponde al valor del ángulo. Por otro lado, la estructura del bloque de articulación sigue un esquema similar, pero con una diferencia en el primer byte, que ahora es 0xC1.

El siguiente gráfico proporciona una representación visual de las direcciones asignadas a cada bloque.

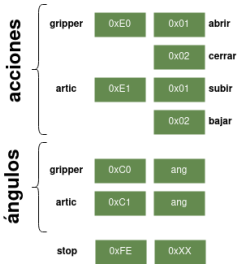

Fig. 42: Declaración de direcciones.

2. Esquema de comunicación FreeRTOS

La comunicación entre la STM y Jetson se da de manera bidireccional. Primeramente, la STM por medio de la función *receiveJetsonTask*, como su nombre lo menciona, recibe los datos desde la Jetson, tales como el estado de la detección de los ArUcos para proceder a la toma de los cubos y ejecutar las tareas especificadas en el esquema. Estos datos permiten el cambio de estados del Gripper entre los tres principales que se han mencionado con anterioridad, tales como la apertura, cerrado y comprobación de la toma del objeto. Al mismo tiempo, recibe los dos estados de la articulación, tales como subir o bajar, o en su aplicación final, girar o regresar al ángulo inicial.

Se han declarado un total de 6 tareas, 4 de las cuales se relacionan con el Gripper, la articulación y sus respectivos ángulos, mientras que los otros 2 abarcan la verificación de la toma del objeto y una llamada stop de emergencia.

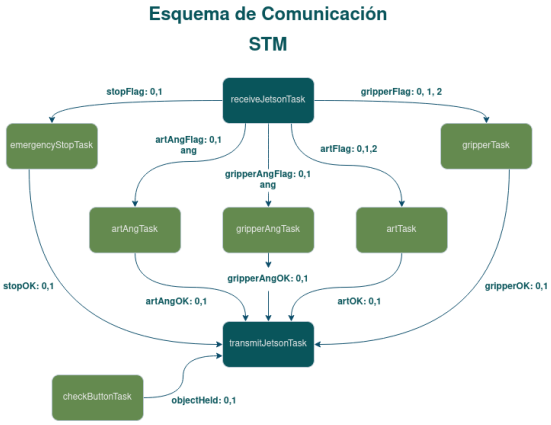

Fig. 43: Esquema de comunicación STM con freeRTOS.

Finalmente, por medio del envío de banderas OK con valores 0 o 1, los estados finales y ángulos tanto de la articulación como del Gripper, son enviadas a la Jetson por medio de la función *transmitJetsonTasks*. De esta manera, se puede proceder con las funciones relacionadas con la navegación y el resto de las acciones planeadas.

3. Diseño MBSD para el filtrado de señales

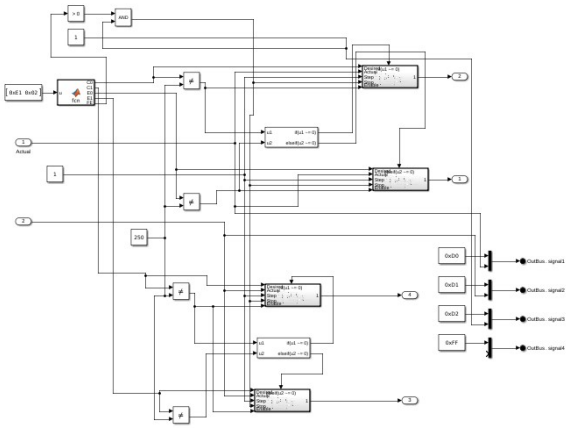

Fig. 44: MBSD en Simulink

4. Pruebas unitarias y Código de tareas FreeRTOS

En el siguiente link se encuentran las pruebas unitarias de las tareas y el proyecto completo implementando FreeRTOS.

Código en Github: Control de Gripper con freeRTOS.

l. RosSerialSTM32

Dentro de los paquetes que ofrece ROS, existe uno que posibilita la subscripción y publicación de tópicos a través de puertos seriales, llamado *rosserial*. Este paquete por sí solo, contiene un nodo que recibe de parámetro un puerto específico donde se estará comunicando con el dispositivo por comunicación serial. Sin embargo, se necesita de otro paquete llamado *rosserial_stm32*, el cual contiene de ejemplos y códigos necesarios para generar las librerías y archivos de encabezado que se necesitan incluir en el proyecto de STM32 que será compilado. Las librerías son de propósito general, por lo que deben ser adecuadas para el modelo específico del microcontrolador. En este caso, se busca realizar una transmisión de datos entre una tarjeta de desarrollo STM32 y una Jetson Nano. La Jetson Nano tiene puertos USB capaces de realizar recibir datos por comunicación serial, sin embargo la STM32F411CEU6 (el modelo específico utilizado), tiene varios puertos Rx y Tx programables para la transmisión asíncrona de datos. En este proyecto, se utilizaron los pines PB6 (Tx) y PB7 (Rx) para la comunicación USART.

```
static void MX_USART1_UART_Init(void)
{

  /* USER CODE BEGIN USART1_Init 0 */

  /* USER CODE END USART1_Init 0 */

  /* USER CODE BEGIN USART1_Init 1 */

  /* USER CODE END USART1_Init 1 */
  huart1.Instance = USART1;
  huart1.Init.BaudRate = 57600;
  huart1.Init.WordLength = UART_WORDLENGTH_8B;
  huart1.Init.StopBits = UART_STOPBITS_1;
  huart1.Init.Parity = UART_PARITY_NONE;
  huart1.Init.Mode = UART_MODE_TX_RX;
  huart1.Init.HwFlowCtl = UART_HWCONTROL_NONE;
  huart1.Init.OverSampling = UART_OVERSAMPLING_16;
  if (HAL_UART_Init(&huart1) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN USART1_Init 2 */

  /* USER CODE END USART1_Init 2 */

}
```

Fig. 45: Configuración de la comunicación serial.

Ya con los pines habilitados por la comunicación serial, fue necesario agregar los archivos en las carpetas *Inc* y *Src* del proyecto, creados con el paquete de ROS mencionado previamente. Todos los archivos generados con el paquete de ROS, se encuentran en C++, por lo que si el proyecto se encuentra programado en lenguaje C, habrá que realizar algunas modificaciones de compatibilidad. Una vez modificados los archivos para que sean compatibles con el proyecto, se puede utilizar la notación de un nodo de C++ en ROS para comunicarse entre tópicos por medio del puerto serial. El código implementado en la tarjeta utilizada para el desarrollo del proyecto es el mostrado en la Figura 45.

```
ros::NodeHandle nh;

void req(const std_msgs::String& msg);

std_msgs::String str_msg;
ros::Publisher chatter("chatter", &str_msg);
ros::Subscriber<std_msgs::String> stm32_comms("gripper_action", &req);
std::string request="";
std::string gripState="";

void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart){
  nh.getHardware()->flush();
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart){
  nh.getHardware()->reset_rbuf();
}
/* USER CODE END PV */
```

Fig. 46: Creación del nodo de ROS en C++.

En esa sección de código, se declaran las variables necesarias para el correcto funcionamiento de un nodo de ROS y pueda ejecutar acciones de publicación y subscripción a tópicos. En este caso, se declara un publicador llamado *chatter* que publica mensajes de tipo String en un tópico del mismo nombre. El tópico *chatter* es donde la STM32 va a actualizar el último estado del brazo robótico, es decir, subir el brazo, bajar el brazo, cerrar el agarre o abrir el agarre. Por otra parte, también se declara el tópico *gripper_action*, en el que la Jetson Nano estará publicando las acciones que tiene que llevar a cabo el gripper, las cuales son las mismas 4 acciones descritas anteriormente. Esta comunicación es posible gracias al nodo *serial_node*, el cual se encarga de leer los datos recibidos por el puerto serial y publicar esos datos en sus respectivos tópicos. El diagrama de comunicación entre los nodos de la Jetson Nano y la STM32 se encuentra en la Figura 47.

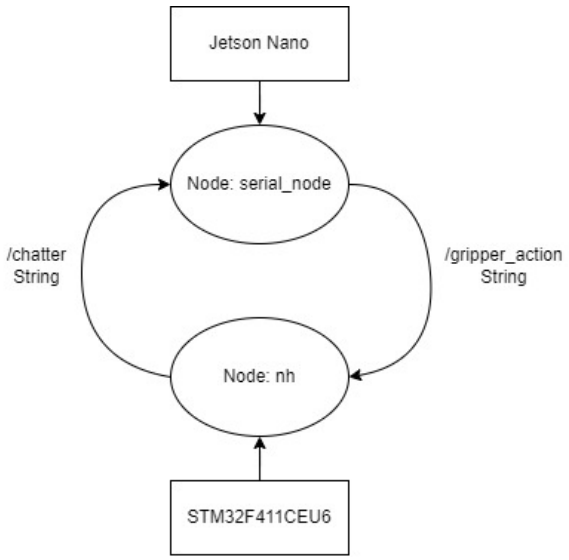

Fig. 47: Diagrama de comunicación con tópicos.

Dentro de la programación de la STM32, existen tres funciones que son fundamentales para el funcionamiento correcto de la comunicación entre nodos. La primera es la función *setup()*, que se encarga de inicializar el nodo *nh* y de añadir los subscriptores y publicadores, es decir, de

establecer la comunicación con los tópicos *chatter* y *gripper_action*. En segunda, se encuentra la función que se encarga de publicar el estado del gripper en el tópico *chatter*, la cual está catalogada como *jetsonResponse()*. Por último, se encuentra la función *req()* que está constantemente obteniendo cualquier actualización del tópico *gripper_action*, y en dado caso, guardar la acción solicitada por la Jetson Nano en su respectiva variable. Las funciones descritas se pueden encontrar en la Figura 48.

```
void setup(void)
{
  nh.initNode();
  nh.advertise(chatter);
  nh.subscribe(stm32_comms);
}


void jetsonResponse(void)
{
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    const char* str = gripState.c_str();
    str_msg.data = str;
    chatter.publish(&str_msg);
    nh.spinOnce();
}

void req(const std_msgs::String& msg){
    request = msg.data;
}
```

Fig. 48: Funciones del nodo nh.

Finalmente, existe la tarea *StartReceiveJetson* que se encarga de modificar las banderas de las otras tareas con base en los mensajes recibidos del tópico *gripper_action*. Por cuestiones de practicidad para las pruebas, en esta ocasión se utilizaron caracteres clave para identificar el tipo de acción que solicitaba la Jetson Nano a través del tópico. Las 5 instrucciones que se determinaron con sus respectivos caracteres clave son: levantar el brazo, con el comando "au"; bajar el brazo, con el comando "ad"; cerrar el agarre, con el comando "gc"; abrir el agarre, con el comando "go"; y finalmente, detener el gripper, con el comando "s". La programación para la modificación de las banderas con base en el comando recibo por medio del tópico *gripper_action* se puede visualizar en la Figura 49.

En la tarea *StartReceiveJetson* solo se cambian los valores de las banderas con las que otras tareas ya se encargan de hacer sus respectivos procesos. Es por eso, que la función *jetsonResponse()* se llama cada vez que se ejecuta el comando obtenido a través del tópico *gripper_action* en su respectiva tarea. Además, previo a su llamada, también se modifica el valor de la variable *gripState* que se publica en el tópico *chatter* donde el nodo de la STM32 publica el estado actual del gripper. Por ejemplo, en la Figura 50 se puede observar cómo es llamada la función *jetsonResponse()* cada vez que se realiza el movimiento hacia arriba o abajo del brazo robótico.

V. Resultados Finales

A continuación, se exhiben los resultados obtenidos mediante la presentación de dos vídeos ilustrativos. El primer vídeo representa la respuesta al desafío planteado por Manchester Robotics, mientras que el segundo responde al reto de la clase y se expande en la implementación física del modelo previamente presentado. Cabe destacar que estos vídeos condensan los resultados obtenidos y sintetizan de

```
void StartReceiveJetson(void *argument)
{
  /* USER CODE BEGIN StartReceiveJetson */
  /* Infinite loop */
  for(;;)
  {
        switch (request[0]){
            case 'g':
                if (request[1] == 'o')
                    {
                        gripperFlag = 1;
                    }
                else if (request[1] == 'c')
                    {
                        gripperFlag = 2;
                    }
                else
                    {
                        gripperFlag = 0;
                    }
                break;
            case 'a':
                if (request[1] == 'u')
                    {
                        artFlag = 1;
                    }
                else if (request[1] == 'd')
                    {
                        artFlag = 2;
                    }
                else
                    {
                        artFlag = 0;
                    }
                break;
             case 's':
                    stopFlag = 1;
                    break;
            default:
                    gripperFlag = 0;
                    artFlag = 0;
                    stopFlag = 0;
                    break;
              }
            osDelay(200);
  }
  /* USER CODE END StartReceiveJetson */
}
```

Fig. 49: Código de la tarea *StartReceiveJetson*.

manera audiovisual el enfoque más refinado de todas las secciones anteriormente descritas.
Manchester: Presentación de reto en simulación
Bloque: Presentación de reto en Fisico
Código en Github: Implementacion de Puzzlebot

VI. Questions about Software/Methods in Robotics/Digital Systems Engineering Challenges

1.- ¿Requiere que los datos de entrada sean preprocesados de manera automatizada o manual?

Considerando que nuestro sistema opera como un robot destinado a la detección de obstáculos, la percepción visual de objetos y el control de diversos estados, resulta pertinente mencionar que todos los datos ingresan al sistema a través de sensores y se mantienen en forma de datos sin procesar. Por consiguiente, en relación a los sensores, es necesario aplicar automáticamente algoritmos con distintas funcionalidades para procesar dichos datos, dado el considerable volumen de valores y relaciones que conforman estos datos.

```
void StartArtTask(void *argument)
{
  /* USER CODE BEGIN StartArtTask */
  /* Infinite loop */
  for(;;)
  {
      if(artFlag == 1)
          {
              TIM2->CCR2 = artUp;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
              gripState = "artUp";
              jetsonResponse();
          }
      else if (artFlag == 2)
          {
              TIM2->CCR2 = artDown;
              osDelay(50);
              artFlag = 0;
              artOK = 1;
              gripState = "artDown";
              jetsonResponse();
          }

      osDelay(100);
  }
  /* USER CODE END StartArtTask */
}
```

Fig. 50: Ejemplo de tarea con llamada a la función *jetsonResponse()*.

Dicha magnitud dificulta su interpretación manual, a excepción de la identificación de señales y cambios de estado, que sí pueden ser abordados de manera manual debido a la calibración de los datos, lo cual permite su observación directa.

2.- ¿Requiere acceso a bases de datos o funciones externas?

Debido a que en la parte de navegación se tiene una de tipo reactiva y local para esta parte no se necesita tener un almacenamiento de los datos, así como un acceso hacia estos. Para la parte de visión se tiene que tomar en cuenta el tipo de ArUcos a trabajar, ya que estos mediante funciones externas tenemos la capacidad de generar e identificar los diferentes objetos. Sin embargo, no es necesario recurrir a accesos de bases de datos, a menos de que en un futuro se incorporen localizaciones de alta demanda o extensión, como lo puede ser en un caso de almacenes en ejemplos reales.

3.- ¿El programa puede ser aplicado a otras áreas?

El programa consiste en la navegación a través de entornos desconocidos, implementando la evasión de obstáculos y la localización del robot a través del procesamiento de imagen. En este caso se generó el caso a partir de almacenes industriales que utilizan robots autónomos, pero sin duda dentro de cualquier ámbito autónomo es sumamente importante el procesamiento del entorno para evasión de obstáculos y según la necesidad, se añadirían los complementos, como Gripper, manos, botones o cualquier herramienta necesaria para automatizar una tarea. Por ello, podemos concluir que este mismo programa podría implementarse para la navegación, detección de obstáculos y automatización de tareas, independientemente del objetivo que se desee lograr.

4.- ¿Se puede determinar el desempeño de unas partes con cambios en otras?

Al dividir tareas, podemos generar planes de mejora, generando cambios en los algoritmos de navegación individualmente, el cual puede sustentarse de la detección de objetos a través del LiDAR y la cámara. La navegación podría depender completamente

5.- ¿Podemos evaluar su comportamiento?

Prueba de recogida de cajas: Realice pruebas para evaluar la capacidad del robot para recoger las cajas de ArUcos con la pinza. Evaluar la precisión y la eficiencia del robot para recoger y soltar cajas, así como su capacidad para manejar diferentes tipos de cajas y posiciones de descarga.
Pruebas de navegación: Verificar con qué tipo de obstáculos y entornos se comporta de mejor manera, materiales, formas y distribuciones, junto con la precisión y la capacidad de planeación durante la navegación.
Pruebas de visión: Evaluar la cercanía hacia los diferentes objetos, que tan centrado se encuentra el objeto desde la perspectiva de la cámara y distancia hacia las pinzas del efector final. Recursos: Verificación de cantidad de energía utilizada y comparación de cantidad recorrida en comparación a la necesitada, junto con trayecto más corto.

6.- ¿Qué tan transparente es el método?

Para que un método sea transparente tiene que cumplir con ciertos criterios, dentro de el nuestro se considera que se categoriza como transparente por los siguientes motivos: El codigo esta abierto, el proyecto que tenemos está en desarrollo público por lo cual las personas pueden tener la capacidad de analizar, revisar y utilizar el codigo para sus proyectos mismos, en cuanto a informes y disponibilidad de documentación detallada, se tiene una dentro de los códigos, de este mismo informe y cada uno de los aspectos mayores durante el proceso, no sin olvidar mencionar la cantidad de recursos audiovisuales expuestos durante este proyecto mostrando el avance del mismo. El único aspecto que tal vez no esté transparente es el de los datos ya que no realizamos obtención o uso de bases de datos o conjuntos externos, sin embargo en términos de transparencia la accesibilidad de documentación, pruebas e informes es bastante clara teniendo una posibilidad de evaluación bastante amplia.

7.- ¿Cuál es el alcance del método?

El alcance de este método es la navegación autónoma a través de entornos desconocidos, evasión de obstáculos y la detección de objetos utilizando un sensor LiDAR, visión computacional y el procesamiento de imagen. Este proyecto permite al robot la percepción de su entorno y la toma de decisiones basadas en la información capturada por sus sensores. Puede tener potencial aplicación en áreas como: Almacenes, permitir la navegación de los robots en la gestión de inventario,

recogida de productos y transporte de mercancías dentro de la bodega ayuda a economizar tiempo y enfocarse en tareas más complejas; otra aplicación es en la Automatización industrial, pues en entornos industriales, el método puede utilizarse para dotar a los robots de la capacidad de interactuar y manipular objetos en un entorno de producción. Esto incluye la detección y toma de objetos utilizando técnicas de visión por computadora y la evasión de obstáculos para garantizar la seguridad en el entorno de trabajo.

8.- ¿Se pueden extender sus capacidades?

En cuanto a las capacidades del robot es posible ser extendidas implementado en cualquier ámbito donde sea necesario el procesamiento del entorno, ya sea mapeo o evasión de obstáculos. Además, permite la posibilidad de añadir complementos según la necesidad específica, por ejemplo, si se requiere que el robot pueda manipular objetos, se podrían integrar funciones de agarre integrando un gripper como actuador. Al tiempo, los algoritmos de navegación pueden ser complementados para que pueda explorar entornos más complejos con facilidad, permitiendo que la ruta sea eficiente para el mapeo del entorno.

9.- ¿Qué tan fácil es para los usuarios utilizarlo ?

Todo el desarrollo del proyecto se encuentra dentro de un repositorio de GitHub, sin embargo, solo por contar con el código, no significa que ya sea posible replicar el proyecto, ni siquiera con una documentación detallada. Para que un usuario pueda replicar este proyecto, debe tener: conocimiento en física y matemáticas para comprender la teoría detrás de la programación del robot; programación orientada a objetos en Python, porque en ese lenguaje se desarrollaron los nodos de ROS; conocimientos de ROS, que fue le middleware utilizado para facilitar la programación del robot; familiaridad con entornos Linux, porque la mayoría de la programación del robot tiene que ser ejecutada forzosamente en sistemas operativos Linux; y finalmente, tiene que haber programado sistemas embebidos en lenguaje C, porque así está desarrollado el funcionamiento de la STM32. Como se puede observar, no son pocos los conocimientos necesarios para entender por completo el proyecto, por lo que en cuestión de facilidad para usar este proyecto, sí se requiere de otro desarrollo enfocado en facilitar su uso.

10.- ¿Cómo responde el sistema si las entradas son organizadas de otra manera, que contengan ruido o que vengan incompletas ?

Depende de cuál sea el ruido que y cuáles son las variables de entrada que serán alteradas. Por ejemplo, si dentro de las tareas de FreeRTOS cambiamos la prioridad de las tareas encargadas de mover las aritculaciones del gripper, en realidad no afectaría mucho el funcionamiento, porque igualmente las tareas para el movimiento de articulaciones dependen de la modificación de banderas que se encuentra en otra tarea principal. Por otro lado, si se modificara el orden de prioridad de la tarea *StartReceiveJetson*, probablemente si provocaría algún mal funcionamiento del proyecto, ya que esta tarea es de las más prioritarias, al ser la encargada de actualizar las banderas de las otras. En cuanto a las entradas con ruido o incompletas, es el mismo caso, si por ejemplo, colocamos al robot en una superficie con mucho derrape, probablemente experimente un mayor error en su odometría, provocando que su posición sea más inexacta. Sin embargo, en este proyecto se utilizan los ArUcos para que el robot vuelva a calibrar su posición en el espacio, por lo que, dependiendo en qué medida el piso afecte la posición del robot, en realidad no debería afectar mucho el funcionamiento del Puzzlebot. Por otro lado, si uno de los ArUcos estuviera incompleto, el robot perdería totalmente su posición, debido a que no podría identificar el ArUco y, por lo tanto, tampoco podría calibrar su posición.

VII. Reflexión y comentarios finales

Fernando. Cuéllar Martínez

Dentro del contexto de aprendizaje y contenido en el bloque, he experimentado un crecimiento tanto académico como profesional en diversos aspectos. Este crecimiento abarca desde el descubrimiento de nuevos métodos de trabajo, liderazgo y comunicación, hasta la adquisición de nuevas habilidades en áreas como algoritmos, integración de software y hardware, planificación de componentes para proyectos y cumplimiento de requisitos establecidos. Considero que esta diversidad de conocimientos y enfoques contribuye a la formación integral de un ingeniero y profesional, capacitado para abordar no solo desafíos específicos, como el planteado en el proyecto Puzzlebot, sino también para aplicar estos conocimientos en diferentes áreas. Es fundamental identificar cómo podemos aplicar cada uno de estos aspectos y conceptos clave, como el reconocimiento de objetos mediante la visión computacional, el filtrado de imágenes o el uso de técnicas como Kalman para mejorar nuestros movimientos y estados. Este proyecto me ha enseñado también la importancia de la interrelación entre la implementación digital y física, así como los desafíos que surgen al traducir conceptos y procesos entre estos dos dominios. Una estructura adecuada y bien diseñada puede facilitar significativamente este proceso. Sin duda, trabajar en la modelación cinemática, la programación de visión, la evasión de obstáculos y la lógica del proyecto ha generado un perfil profesional completo, capaz de incorporarse a diferentes áreas y de ampliar las capacidades del proyecto en sí, abriendo así posibilidades para intereses futuros y nuevas aplicaciones.

Jorge Antonio. Villegas Hernández

El proyecto de esta ocasión fue verdaderamente retador, debido a que tuvimos que aplicar todos los conocimientos adquiridos durante la carrera. El objetivo del proyecto era programar el Puzzlebot que se nos fue proporcionado, para lograr que haga una rutina de evasión de obstáculos y manipulación de objetos, utilizando visión computacional para aproximarse a los objetos con códigos ArUcos y un brazo robótico programado generado totalmente por el equipo y adaptado al Puzzlebot. Por lo tanto, se tuvieron que aplicar los conocimientos de sistemas embebidos y de robótica de todos los semestres anteriores. El brazo robótico fue programado en una tarjeta STM32, por lo que reforzamos y reaprendimos conocimientos de semetres pasados para lograr la implementación de rutinas para manipulación de ob-

jetos. Por otra parte, retomamos el uso de ROS para el funcionamiento de sistemas robóticos en tiempo real y lograr una navegación autónoma del modelo que se nos fue proporcionado. Y la parte que considero fue la más divertida y con más aprendizajes, fue la integración de ambas partes para alcanzar los resultados obtenidos.

María Fernanda. Hernández Montes

A lo largo del desarrollo de este bloque se nos presentó una amplia variedad de retos y tareas que nos brindaron la posibilidad de ampliar nuestros conocimientos en algunas áreas así como profundizar en otras, pues el proyecto final consistía de la implementación e integración de sistemas complejos por sí mismos dentro de su propio enfoque. Es por esto que, de manera personal, experimentar el proceso de solución de una tarea con este nivel de dificultad me ha resultado tan enriquecedor como retador, ya que me ha expuesto al aprendizaje y reaprendizaje de contenido que será de gran valor e impacto tanto a nivel personal como académico y profesional. Dado que el objetivo final era integrar diversos sistemas unitarios con la finalidad de programar un robot de tipo Puzzlebot con la finalidad de que el mismo fuera capaz de navegar de manera autónoma dentro de un entorno variable y de interactuar con los objetos a su alrededor, pude profundizar en temas como visión computacional, sistemas embebidos y programación de sistemas robóticos en tiempo real. Si bien es importante identificar la manera y nivel en los cuales cada ámbito puede impactar en el resultado final, de igual manera considero que es de vital importancia reconocer aquellos que tienen un impacto profundo a nivel individual en el rubro profesional, algo que me sucedió dentro del uso de microcontroladores para el diseño de sistemas embebidos, pues tuve la oportunidad de afinar mis habilidades en el área al desarrollar el control del gripper que utilizamos con la finalidad de que nuestro robot fuera capaz de levantar y soltar objetos en el entorno físico de navegación

Melissa. Montemayor Riojas

Los aprendizajes en este reto fueron altamente variados, desde el desarrollo de los algoritmos, la comprensión de cómo es que funcionan y el cómo implementarlos en el Puzzlebot. En esta sección, descubrí lo que era cada algoritmo y la lógica detrás del porqué un Bug sería más óptimo que otro. Después, la lectura de ArUcos fue igualmente un recordatorio de cómo procesar imágenes y la toma de decisiones según lo que la cámara obtenía de información. Posteriormente, la mayor oportunidad de aprendizaje fue la comprensión del Filtro de Kalman, el cual me permitió comprender cómo el implementar un filtro para trabajar con señales más limpias nos permitió obtener resultados aún mejores. Por ello, considero que la implementación del filtro fue esencial para la obtención de movimientos más fluidos y observamos tanto digital como físicamente el funcionamiento y los beneficios de implementarlo. Finalmente, el diseño del Gripper, crear las piezas y modificarlas para la adaptación al Puzzlebot, las consideraciones digitales y físicas al trabajar con los URDF y STL me permitió comprender cómo se hacían los modelos para Rviz y Gazebo el cual me permitió entender que los aprendizajes no solamente se cierran a los Puzzlebots, sino que se podría trabajar con cualquier modelo robótico y aprender a generar simulaciones con nuevos retos.

Melissa. Vélez Martínez

Este proyecto integrador me ha brindado una perspectiva interesante sobre la automatización y la mejora de la calidad en la industria logística, principalmente en almacenes o cadenas de suministro. En mi experiencia, considero que aprendí muchas cosas nuevas sobre los temas aplicados en el proyecto. En las pruebas realizadas con el Puzzlebot, pude notar la importancia de la precisión y la incertidumbre en la robótica, tal es el caso de los ArUcos y su uso para determinar mejor la posición actual en el mapa, así como el uso de los bugs en la navegación y las ventajas, dificultades y consideraciones que tenía cada uno. Es importante tener en cuenta la incertidumbre y buscar formas de reducirla, como fue la aplicación del filtro de Kalman en estos aspectos con el fin de obtener resultados más confiables. Esto me lleva a apreciar el desafío de la complejidad en el desarrollo de sistemas robóticos avanzados, pues se requiere de un enfoque multidisciplinario que abarque desde la visión y la percepción del entorno hasta el control y la toma de decisiones.

REFERENCES

[1] M. Zohaib, S. M. Pasha, N. Javaid, and J. Iqbal, “Iba: Intelligent bug algorithm – a novel strategy to navigate mobile robots autonomously,” in *Communication Technologies, Information Security and Sustainable Development*, F. K. Shaikh, B. S. Chowdhry, S. Zeadally, D. M. A. Hussain, A. A. Memon, and M. A. Uqaili, Eds. Cham: Springer International Publishing, 2014, pp. 291–299.

[2] M. Serna Hernández, A. Sánchez, and L. Puebla, ““navegación de robots móviles utilizando los algoritmos bug extendidos” tesis profesional,” 2018. [Online]. Available: https://repositorioinstitucional.buap.mx/bitstream/handle/20.500.12371/7420/683018T.pdf?

[3] planificación de trayectoria ci-2657 robótica prof. kryscia ramírez benavides. [Online]. Available: https://www.kramirez.net/Robotica/Material/Presentaciones/PlanificacionRuta.pdf

[4] M. Pascual, “Bug0, bug1, bug2 explained,” 2016. [Online]. Available: www.slideshare.net/luisalfredomoctezumapascual/bug1-y-bug2

[5] S. David and L. Beltrán, 2017. [Online]. Available: repository.libertadores.edu.co/bitstream/handle/11371/1441/leonsergio2017.pdf?

[6] Z. Hager, Dodds, and H. Choset, “6-735, howie choset with slides from g.” [Online]. Available: https://www.cs.cmu.edu/~motionplanning/lecture/Chap2-Bug-Alg_howie.pdf