Implementación de un algoritmo de árboles de comportamiento y visión de computadora para el patrullaje autónomo de un robot tipo “skid steer” en un ambiente agrícola simulado en ROS

Ángel Yesid Ardila Ardila

Trabajo de Grado para optar al título de Ingeniero Mecánico

Director

Carlos Alberto Flórez Arias

Candidato a Doctor en Ingeniería Mecánica

Universidad Industrial de Santander

Facultad de Ingenierías Fisicomecánicas

Escuela de Ingeniería Mecánica

Bucaramanga

2024

Dedicatoria

Quisiera dedicar mi trabajo de grado, primero a mi familia quienes fueron un apoyo gigante durante toda mi carrera.

También, a mis amigos, a aquellos a quienes conocía desde tiempo, a Karen, a Mafe, a Angy y a Natalia, también a aquellos que conocí durante el transcurso de mi carrera, a Anny, a Anita, a Stef, a Jesús y a Sam. Todos ellos que han sido una parte importante de mi vida y me han apoyado de todas las formas posibles para que hoy en día este trabajo pueda estar terminado.

Por último le quisiera dedicar este trabajo a mi gato Totoro, con quien encontraba una escapatoria del estrés y la ansiedad.

Agradecimientos

Quisiera agradecer primero a mi familia por el apoyo durante el transcurso de mi carrera, en especial a mis hermanos Julio y Diana, quienes me apoyaron económicamente para poder cursar toda mi universidad sin ningún percance.

Además quisiera agradecer al semillero de robótica DicBot de la escuela de ingeniería mecánica, y al ingeniero Carlos Flórez, quien me enseñó las bases por las cuales fué posible realizar este trabajo de grado.

Tabla de Contenido

Lista de Figuras

Lista de Tablas

Lista de Apéndices

pág.

Glosario

Plugin Es un componente de software que añade un comportamiento específico a un programa ya existente. Estos pueden ser realizados por personas externas a el programa al que va dirigido.

framework Es un conjunto estandarizado de conceptos, prácticas y criterios para enfocar un tipo de problemática particular.

middleware Es un sistema de software que brinda comunicación entre aplicaciones.

Resumen

Título: Implementación de un algoritmo de árboles de comportamiento y visión de computadora para el patrullaje autónomo de un robot tipo "skid steer" en un ambiente agrícola simulado en ROS [*]

Autor: Ángel Yesid Ardila Ardila [**]

Palabras Clave: ROS2, Árboles de comportamiento, Visión de computadora, Robot "skid steer", simulador Gazebo

Descripción: El presente trabajo presenta una propuesta para la navegación autónoma de un robot en un ambiente agrícola. Para la validación del algoritmo se usó un ambiente virtual simulado en Gazebo y generado de forma procedural usando Python. los comportamientos de robot se programaron usando ROS2 en Python y C++, además, se incluyó la implementación de un árbol de comportamientos y de visión por computadora para la navegación. Finalmete se hace una comparación entre el modelo usado para la navegación autónoma y los modelos de odometría tradicionales.

---

[*] Trabajo de grado

[**] Facultad de Ingenierías Físico-Mecánicas. Escuela de Ingeniería Mecánicas. Director: Carlos Alberto Flólez Arias, Candidato a Doctor en Ingeniería Mecánica.

Abstract

Title: Implementation of a behavior tree algorithm and computer vision for the autonomous patroll of a robot type skid steer in an agricultural environment simulated in ROS [*]

Author: Ángel Yesid Ardila Ardila [**]

Keywords: ROS2, Behavior Trees, Computer vision, Skid Steer robot, Gazebo simulator.

Description: The present work presents a proposal for the autonomous navigation of a robot in an agricultural environment. To validate the algorithm is used a virtual enviroment simulated in Gazebo and and procedurally generated using Python. The robot behaviors were programmed using ROS2 in Python and C++, in addition, was included the implementation of a behavior tree and computer vision for navigation. Finally, a comparison is made between the model used for autonomous navigation and traditional odometry models.

---

[*] Bachelor Thesis

[**] Faculty of Physical-Mechanical Engineering. School of Mechanical Engineering. Director: Carlos Alberto Flólez Arias, Candidate for Doctor in Mechanical Engineering.

Introducción

El desarrollo agrario es un punto clave para las economías futuras, tanto a nivel nacional como mundial, se estima que para el año 2050 la población mundial supere los nueve mil millones de personas (Karkee y Zhang, 2021), llevando consigo un aumento de la demanda de bienes y servicios, principalmente el de alimentos. Por ello organizaciones como la ONU en la agenda 2030 plantea: "de aquí a 2030, duplicar la productividad agrícola y los ingresos de los productores" (CEPAL, 2018).

A nivel nacional, el sector agropecuario representa cerca del 6 % del PIB, y su crecimiento anual es del 2.3 %, debajo del promedio de la región que es del 2.6 % (Misión para la Transformación del Campo, 2015), debido a esto, y de acuerdo con los indicadores planteados en la agenda 2030, se debe aumentar la productividad del campo, una forma de lograrlo es la adopción de tecnologías agrícolas automatizadas Karkee y Zhang (2021).

El avance tecnológico actual ha conseguido que robots agrícolas puedan realizar labores como la labranza, la siembra, aplicación de productos químicos y cosecha (Karkee y Zhang, 2021) en grandes extensiones de tierra y con poca intervención humana. Para poder cumplir con las operaciones designadas la principal función solicitada por el robot es la capacidad de percibir su entorno, y encontrar formas de moverse en él (Karkee y Zhang, 2021), constituyendo así como uno de los núcleos centrales del robot la navegación autónoma.

Esta percepción que adquiere el robot resulta ser una de las partes más complejas de su funcionamiento (Karkee y Zhang, 2021), creciendo esta complejidad con los grados de libertad del

robot. El proceso para planear la trayectoria se lleva a cabo mediante adquisición y procesamiento de imágenes, es decir, visión de computadora, ya sea con percepción en 2D (visión monocular) o en 3D (visión binocular) (Karkee y Zhang, 2021), siendo la última una opción más robusta para el manejo del robot, pues se obtiene información de la profundidad.

Por ello es importante el desarrollo de algoritmos que permitan al robot cumplir con todos los aspectos que implica la navegación autónoma, como evitar la colisión y lidiar con información incompleta de sensores, esto para cumplir con los requerimientos que solicita las nuevas tecnologías agrarias.

1. Objetivos

Objetivo general

Implementar un algoritmo de árboles de comportamiento y visión de computadora para el patrullaje autónomo de un robot tipo “skid steer” en un ambiente agrícola simulado en ROS.

Objetivos específicos

Construir un ambiente virtual en Gazebo conformado por 8 hileras de plantas de tomate en un campo de 20mX10m para definir los comportamientos del robot.

Diseñar el árbol de comportamiento, programar los nodos necesarios para la navegación del robot en el ambiente virtual e integrar el modelo de visión de computadora.

Simular y analizar el desempeño del modelo de navegación mediante la medida del ángulo de navegación generado respecto a la trayectoria teórica.

2. Marco Referencial

2.1. Antecedentes

El desarrollo de robots autónomos ha tenido alta relevancia en investigaciones recientes alrededor del mundo, esto debido a que es un tema de importancia para la robótica moderna; el trabajo de Teh, Kit Wong, y Min (2021) implementa un algoritmo de Dijksta para planificar la ruta de un robot al interior de un edificio. En este trabajo usaron una cámara ESP23CAM y un sensor IMU para controlar el robot y así mismo actualizar la posición del robot dentro del mapa. El procesamiento de las imágenes se hizo mediante OpenCV en python y se realizó en dos etapas, la primera que convierte las imágenes en escala de grises y aplica un filtro y la segunda analiza las imágenes para detectar obstáculos mediante la transformada rápida de Fourier.

Por otra parte el trabajo de He, Cheng, Zheng, Sun, y Yu (2017) implementó un algoritmo de ubicación adaptado de Monte Carlo para lograr la localización del robot dentro del mapa. Este trabajo se desarrolló completamente en el framework de programación de robots ROS, usando paquetes de localización y mapeo simultaneo (SLAM) en 2D y se usó un sensor láser URG-04LX.

En la misma linea al trabajo anterior, Habibie, Nugraha, Anshori, Ma'sum, y Jatmiko (2017) realizaron la implementación del algoritmo de SLAM para un robot en un ambiente agrícola, todo dentro de un entorno de simulación. La simulación se realizó en Gazebo, que provee ROS, y el entorno constaba de un arreglo de árboles de dos tipos, árboles de manzanas y palmas aceiteras. El robot usado es de la marca Clearpath Robotics modelo Husky, equipado con un sensor láser LMS1xx y un sensor visual Kinect. Se implementaron una comparación de dos algoritmos de

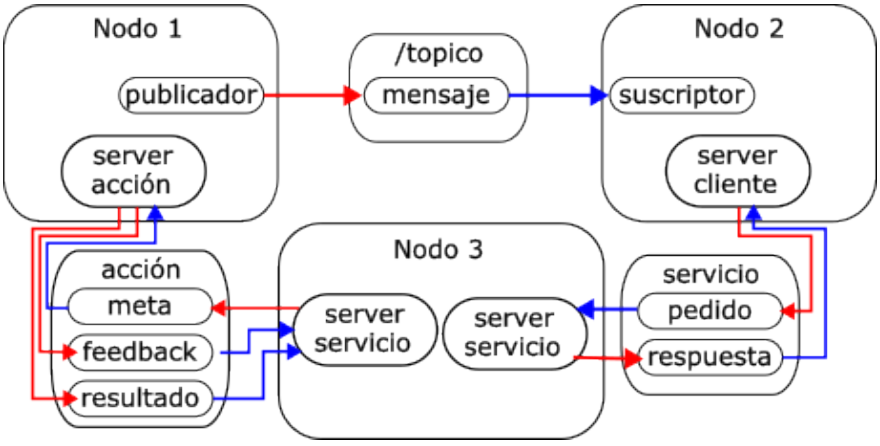
SLAM disponibles para entornos abiertos, esto se realizó adquiriendo datos del sensor láser, además que con el mismo se detectaba los árboles del entorno. Los datos del sensor visual se usaron para detectar frutos en los árboles mediante la técnica de detección de color y manchas.

Finalmente, los trabajos de Ahmadi, Nardi, Chebrolu, y Stachniss (2020) y Song, Xu, Yao, Liu, y Yang (2020) presentan dos soluciones para la navegación autónoma mediante visión de computadora, usando segmentación de imágenes logrando realizar la navegación en entornos agrícolas sin necesidad de algoritmos de SLAM y únicamente con cámaras RGB.

2.2. Marco Teórico

2.2.1. "Robot Operating Sistem". También conocido por sus siglas ROS o ROS2 en su segunda versión, es un "framework" de código abierto que permite el desarrollo de aplicaciones para robots. El ecosistema del software está dividido en tres caregorías: el "middleware", los algoritmos y las herramientas de desarrollo (Macenski, Foote, Gerkey, Lalancette, y Woodall, 2022), así mismo, el desarrollo de aplicaciones está basado en nodos, tópicos, servicios y acciones como se muestra en la imagen 1.

Figura 1. Esquema comunicación en ROS



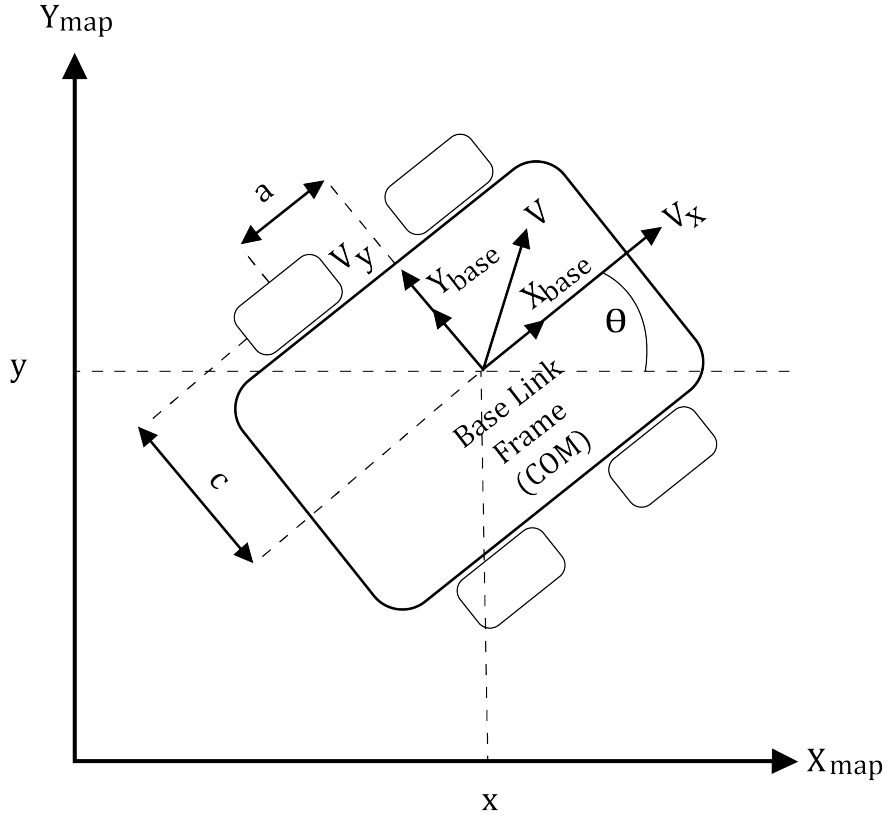
los tópicos son la forma más común de comunicarse en ROS, ellos transportan un mensaje proveniente de un publicador y cualquier nodo suscrito a tópico lo recibe de forma asíncrona. los servicios, a diferencia de los tópicos no funcionan de forma asíncrona, sino solo se envía una respuesta cuando se tiene un pedido. Por otra parte, las acciones son una forma de comunicación asíncrona, adicionando un método de control de metas mediante el feedback.

2.2.2. Robot "skid steer". Un robot autónomo es aquel que puede desarrollar la tarea para la cual fue diseñado sin necesidad de interacción humana, para ello el robot está compuesto de cuatro características: locomoción, percepción, cognición y navegación (Rubio, Valero, y Llopis-Albert, 2019). La locomoción es la que se encarga del movimiento del robot en sí, para controlar este aspecto hay que comprender el mecanismo por el cual se mueve el robot, es decir, su cinemática.

Así mismo, los robots se pueden clasificar dependiendo de su tipo de locomoción en robots estacionarios como los brazos manipuladores, robots voladores, robots acuáticos, y en robots terrestres, estos dividíos así mismos en robots con piernas, con orugas o con ruedas (Rubio y cols., 2019). En el caso de los robots "skid steer", son robots de cuatro ruedas, cada rueda activada independientemente por un motor, debido a esto, su cinemática puede resultar compleja, pero puede simplificarse a un robot de dos ruedas de tracción diferencial (Wen Zhu, Hill, Biglarbegian, Gadsden, y Cline, 2023).

En la figura 2 se muestra el esquema general de un robot "skid steer", podemos definir un vector para la posición del mismo como $q = [x, y, \theta]^T$ y, de la misma forma un vector para la velocidad como $\dot{q} = [\dot{x}, \dot{y}, \dot{\theta}]^T$, todo esto respecto a el origen inercial, y tomando como origen el centro

Figura 2. Esquema de un robot "skid steer"



de masa (COM). Entonces la velocidad del robot respecto al origen inercial estaría representada por la siguiente expresión:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix} \tag{1}$$

Debido a las restricciones propias del robot se puede asumir que $v_y = 0$ (Wen Zhu y cols.,

2023). Entonces, se tendría lo siguiente:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ -\sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v_x \\ \omega \end{bmatrix} \tag{2}$$

Por consiguiente, se obtiene la matriz mediante la cual se pude controlar la velocidad del robot como $\eta = [v_x, \omega]^T$.

2.2.3. Simulación de ambientes agrícolas. El uso de simuladores para el desarrollo de robots en contextos agrícolas ha demostrado poder aumentar la calidad de cultivos a gran escala de una manera más sustentable dependiendo menos de la mano de obra disponible (R. Shamshiri y cols., 2018); trabajos como el de Habibie y cols. (2017) ha demostrado la versatilidad y fiabilidad para poder probar y validar resultados por medio de simulaciones.

Los entornos de simulación disponibles son variados y cada uno de ellos ofrece ventajas propias, pero no todos ellos pueden ser usados para simulaciones agrícolas, debido a que un robot de este tipo interactúa con un ambiente dinámico diferente a otras aplicaciones industriales (R. Shamshiri y cols., 2018). Los principales entornos de simulación que pueden ser usados son los siguientes:

- Webots: Este es un simulador multiplataforma mayormente usado para el desarrollo de tractores autónomos. Puede ser usado con múltiples lenguajes de programación como C/C++, Python o Java, además que se puede integrar con herramientas como OpenCV para trata-

miento real de imágenes o SUMO o OSM para simulación tráfico y robots autónomos.

- Actin: Este simulador está diseñado para tareas como planificación de rutas, planificación de movimiento, prevención de colisiones y control de robots articulados, por ello su principal uso se da en la industria del transporte y en ocasiones muy limitadas en industria agrícola.

- Gazebo: Este es un simulador de código abierto integrado directamente en ROS, este posee un poderoso motor de físicas que puede ser usado para tareas complejas como interacción y movimiento de objetos además de ser usado ampliamente para simulaciones agrícolas, es por ello que es considerado por múltiples expertos como el mejor simulador de robots (R. Shamshiri y cols., 2018).

- RobotDK: Es un software de simulación offline, dedicado exclusivamente a robots industriales, contando con una biblioteca de modelos CAD de diversos robots comerciales.

- Morse: Este simulador es una aplicación completamente desarrollada en python, también tiene una complejidad mayor ya que no tiene una interfaz y todo tiene que ser programado desde scripts de python, tampoco tiene integrados algoritmos de planificación de movimiento, por ello es recomendado solo para desarrolladores experimentados.

- OpenRave: Es un simulador centrado en algoritmos de planificación de movimiento para robots industriales, siendo uno de los más estables y rápidos en cuando al cálculo de información cinemática y geométrica.

- OpenHRP3: Es un simulador de código abierto que gracias a su sistema basado en objetos

facilita los cálculos en simulaciones dinámicas, mejorando la portabilidad y mantenibilidad del proyecto.

- ARGoS: Este simulador está centrado en el desarrollo, control y comunicación en enjambres de robots.

2.2.4. Visión por computadora. La visión por computadora es una técnica por la cual se analizan imágenes para cumplir con tres tareas principales: clasificación de imágenes, segmentación de imágenes y la detección de objetos (Chollet, 2017). la clasificación de imágenes consta de clasificar una imagen entre un o más categorías, la segmentación trata de partir la imagen y cada partición representa una categoría y en la detección de objetos se dibujan rectángulos en la imagen (llamados "bounding boxes"); además de ello la visión de computadora se usa para más tareas, como comparación de imágenes, detección de puntos de interés, estimación de pose entre otros.

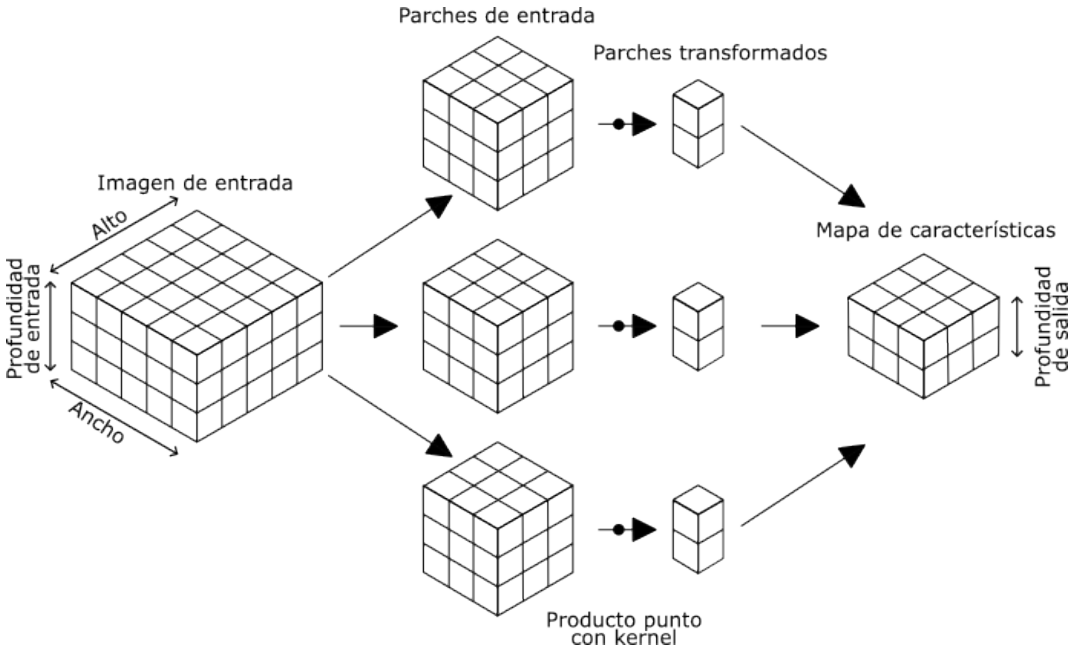
Esta técnica hace uso principalmente del redes neuronales convolucionales (convnets), así mismo, su principio matemático es la operación de convolución ilustrada en la figura 3. Esta operación funciona como un filtro para obtener los elementos característicos de la imagen, dando como resultado un mapa de características.

Matemáticamente la operación convolución está definida como:

$$f[m] * g[m] = \sum f[n]g[n-m] \tag{3}$$

donde $f[m]$ y $g[m]$ son las matrices de entrada.

Figura 3. Esquema de una operación de convolución



Por otra parte, actualmente existen librerías como Open Computer Vision (Opencv) enfocadas en aplicaciones de tiempo real que aplica operaciones sobre la imagen sin necesidad del uso de redes neuronales, ya que emplea algoritmos optimizados directamente sobre las imágenes.

En el caso específico de aplicaciones en agricultura, hay seis áreas importantes de aplicación (Tian, Wang, Liu, Qiao, y Li, 2020):

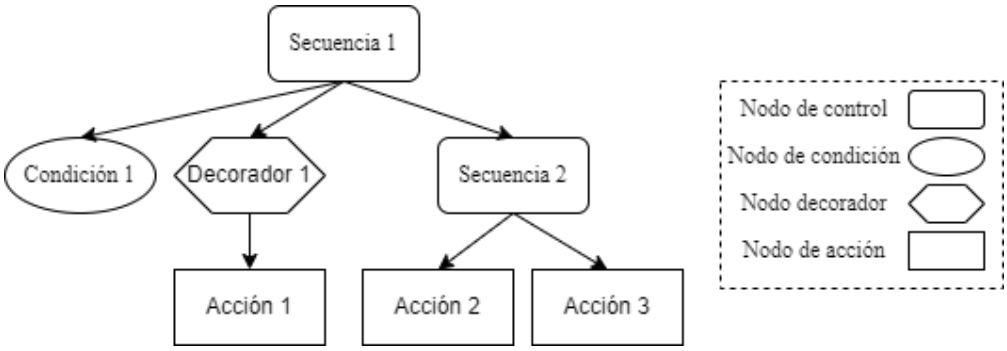
- Monitoreo de salud de plantas.
- Prevención de plagas y crecimiento de hierva.
- Cosecha de cultivos automática.
- Clasificación e inspección de calidad de productos agrícolas.
- Administración automática de granjas.

- Monitoreo de tierras de cultivo con vehículos aéreos no tripulados.

Además de estos usos, recientemente la visión por computadora ha sido usada para navegación autónoma. En el trabajo de Ahmadi y cols. (2020) se usa segmentación para seguir una linea de cultivos, de forma similar el trabajo de Song y cols. (2020) usa visión por computadora para obtener el camino a seguir por un robot entre dos lineas de cultivos.

2.2.5. Árboles de comportamiento. Un árbol de comportamiento (BT) es un modelo de codificación para el control de un sistema (Rico, 2022). Este modelo está compuesto por nodos que son activados por una acción llama tick, y pueden retornar tres estados: éxito, falla, o corriendo. en la figura 4 se ve la estructura general de un árbol de comportamientos, dependiendo de la posición del nodo en la estructura se puede categorizar como nodo raíz, padre, hijo o hoja (Iovino, Scukins, Styrud, Ögren, y Smith, 2022). El nodo raíz es el que está en la parte más alta del diagrama, y a partir de él aparecen los nodos hijos, así mismo estos nodos hijos pueden ser hojas, si no tienen nodos hijos, o pueden ser nodos padres.

Figura 4. Esquema general de un árbol de comportamiento



También la figura 4 ilustra los cuatro tipos de nodos principales que pueden existir: los nodos de control, de condición, decoradores y de acción (Rico, 2022).

- Nodos de control: Estos se encargan de de transmitir el tick a sus hijos, su valor de retorno depende del tipo de nodo de control que puede ser ya sea de secuencia, retroceso o decorador. En la tabla 1 se detalla la clasificación de los nodos de control según Rico (2022).
- Nodos de condición: retornan éxito o falla dependiendo de la condición que tiene pero nunca retornan corriendo.
- Nodos decoradores: son nodos de control con un solo hijo.
- Nodos de acción: son nodos de tipo hoja, y retornan el estado de la acción que deben ejecutar.

Los árboles de comportamiento, debido a la ventaja que suponen para el control de sistemas han sido usados principalmente en robótica para manipulación y navegación tanto terrestre como aérea y subacuática (Iovino y cols., 2022). En el caso de la navegación terrestre, pueden ser usados para el control de patrullaje en campo (Rico, 2022), en este caso se puede añadir trayectorias semánticas (Iovino y cols., 2022) añadiendo buena respuesta ante obstáculos y fallas en el camino.

Tabla 1
Tipos de nodos de control en árboles de comportamientos

		Valor retornado por los hijos		
Tipo de nodo de control		Falla	Éxito	Corriendo
Secuencia	Sequence	Retorna falla y reinicia la secuencia	Transmite el tick, retorna éxito si no hay más hijos	Retorna corriendo y vuelve a activar el tick
	ReactiveSequence	Retorna falla y reinicia la secuencia	Transmite el tick, retorna éxito si no hay más hijos	Retorna corriendo y reinicia la secuencia
	SequenceStar	Retorna falla y vuelve a activar el tick	Transmite el tick, retorna éxito si no hay más hijos	Retorna corriendo y vuelve a activar el tick
Retroceso	Fallback	Transmite el tick, retorna fallo si no hay más hijos	Retorna Éxito	Retorna corriendo y vuelve a activar el tick
	ReactiveFallback	Transmite el tick, retorna fallo si no hay más hijos	Retorna Éxito	Retorna corriendo y reinicia la secuencia
Decoradores	InverterNode	Retorna Éxito	Retorna Falla	Retorna Corriendo
	ForceSuccessNode	Retorna Éxito	Retorna Éxito	Retorna Corriendo
	ForceFailureNode	Retorna Falla	Retorna Falla	Retorna Corriendo
	RepeatNode(N)	Retorna Falla	Retorna Corriendo N veces y retorna corriendo	Retorna Corriendo
	RetryNode(N)	Retorna Corriendo N veces y retorna Falla	Retorna Éxito	Retorna Corriendo

3. Metodología

3.1. Configuración del Mundo en Gazebo

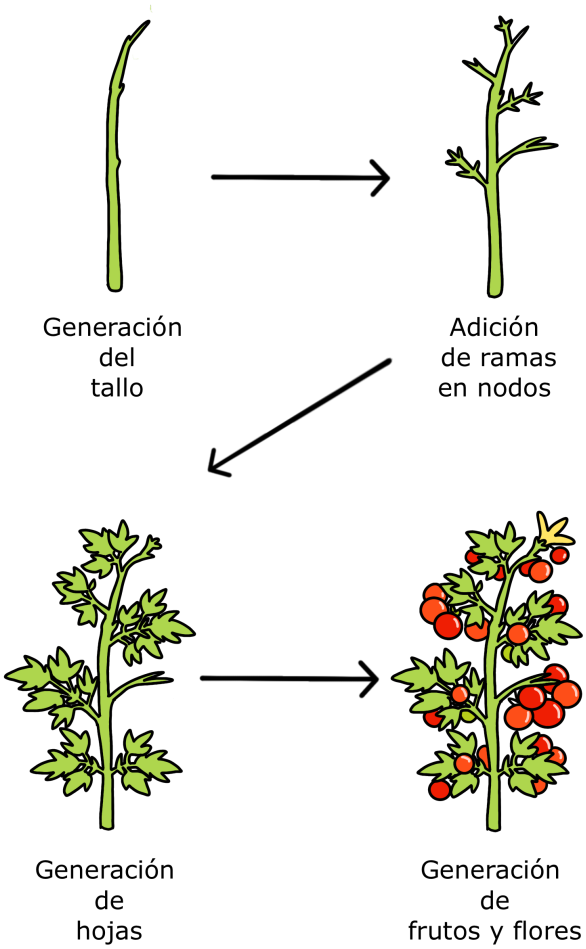
Para la generación del mundo se siguieron principalmente tres procesos, el primero es la generación de los modelos de las plantas de tomate, el segundo es la creación en conjunto del arreglo de plantas y el tercero es la inclusión del modelo del robot a usar.

3.1.1. Generación de plantas de tomate. Las plantas de tomate en cuanto a su morfología, crecen entre unos 60 a 180 cm, sus partes principales son el tallo, las hojas, el fruto y las flores. El tallo se compone de una única rama que crece en dirección vertical, de este desprenden las ramas; las hojas están atadas a las ramas que pueden crecer hasta los 45 cm, el fruto tiene forma de esfera con diámetros entre los 1.5 a 7.5 cm, finalmente las flores conformadas por 5 pétalos de color amarillo y tienen un tamaño de 2 cm de lado a lado (Mangal, 2020).

En la figura 5 se representa la planta de tomate con sus partes y el procedimiento para generarla; el tallo se puede construir mediante la unión de diferentes cilindros, cada uno con una pequeña desviación y de este desprenden las ramas y hojas. Las hojas son planos deformados posicionados en las terminaciones de las ramas, los frutos son esferas puestas en grupos al final de ciertas ramas en la parte superior y el modelo de las flores se generaría en las ramas ubicadas en la punta del tallo.

3.1.2. Construcción del mundo. Para la construcción del mundo se deben integrar todos los modelos de las plantas generados con un plano que representa el terreno y exportarlos al simulador Gazebo, para ello, se debe representar los modelos en un formato de descripción de

Figura 5. Modelo de generación de plantas de Tomate



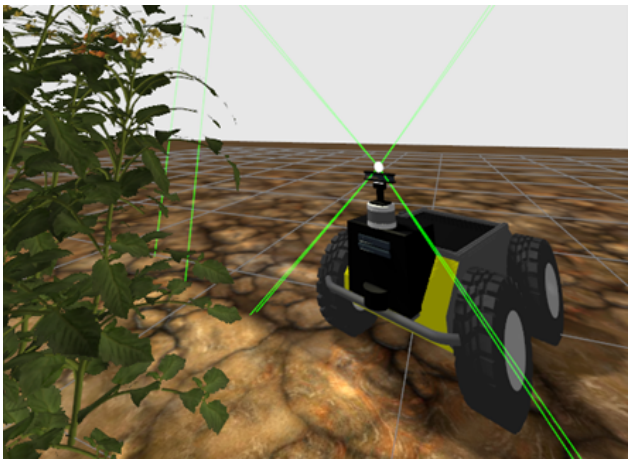
simulación (extensión .SDF), este formato contiene la ruta a la malla del modelo, el nombre del modelo y las coordenadas tanto de translación como de rotación que tendrá el modelo en el mundo.

Además, en este mismo formato se incluyen los "plugins" para cargar el motor de físicas del simulador.

3.1.3. Modelo del Robot. Para la simulación se usó el modelo del robot husky de la marca crealpath mostrado en la imágen 6. Este cuenta con tres cámaras RGBD modelo realsense realsense d435i instaladas en la parte frontal del robot, y se usará para la simulación la que está en

la parte más superior.

Figura 6. Robot husky



Para incluir el modelo del robot en el mundo se importa directamente desde Fuel, que es la biblioteca de modelos de robots propios en Gazebo.

3.2. Diseño del árbol de comportamientos e implementación de nodos

El árbol de comportamientos se puede dividir en dos grupos de nodos de control, el primero, que se encarga de simular el comportamiento de la batería, y el segundo, se encarga de controlar la navegación autónoma.

3.2.1. Modelado de la batería del robot. El desgaste de la batería depende del gasto energético que consumen los motores, directamente ligado a la velocidad, y a un gasto pasivo consecuencia de los componentes electrónicos del robot, entonces, el decaimiento del nivel de carga se encontraría de la siguiente forma:

$$\Delta_b = (v \times D_l + \varepsilon) \times \delta t \tag{4}$$

donde Δ_b es el porcentaje de batería que cayó en el instante δt, v es la velocidad y D_l y ε son constantes de decaimiento. A partir de la ecuación anterior se puede encontrar el nivel de batería actual:

$$BL_t = BL_{t-1} - \Delta_b \tag{5}$$

Donde BL_t es el nivel de batería actual y BL_{t-1} es el nivel anterior.

3.2.2. Nodos de control de navegación. Para controlar la navegación autónoma se debe tomar en cuenta la secuencia de acciones necesarias para poder recorrer todo el espacio, es decir, el cambio entre hileras de plantas y la navegación a través de hileras.

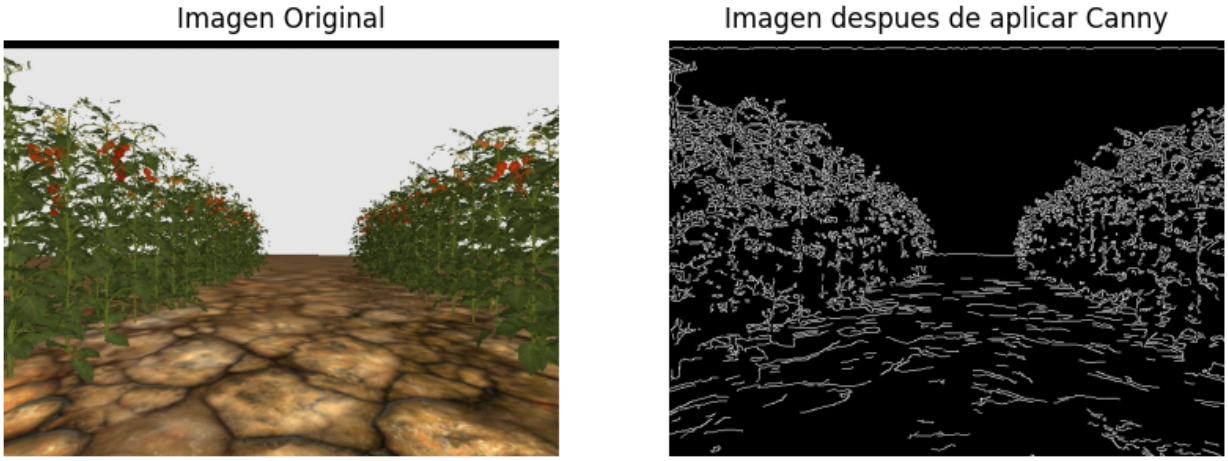
La lógica detrás de la navegación atravesando las hileras le corresponde al modelo de visión por computadora, mientras que el cambio entre hileras, se cuenta con información de la distancia entre ellas y se puede considerar que la forma de cambiar entre ellas es constante, por lo tanto se puede cambiar de hilera sin necesidad del uso de un modelo de visión.

3.2.3. Implementación de modelo de visión de computadora. Como se había propuesto anteriormente, se usará un modelo de visión por computadora para realizar la navegación entre hileras, el proceso general para lograr este objetivo es el siguiente:

1. Obtención de la imagen: se captura la imagen del entorno por medio de la cámara en el robot.

2. Aplicación de la función para la detección de bordes: Una vez se obtiene la imagen, a esta se le aplica la función “Canny”, que aplica un filtro para que de la imagen original se obtenga una nueva donde solo estén los bordes de los objetos presentes como se muestra en la imagen

Figura 7. Ejemplo función Canny



7. Esta función aplica primero un filtro Gaussiano para eliminar ruido en la imagen, este está dado por la siguiente expresión:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{6}$$

La ecuación anterior da como resultado una matriz con la cual se aplica la operación convolución con la imagen obtenida. Seguidamente se vuelve a aplicar la operación convolución con el operador de Sober, definido como lo siguiente:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * A$$

$$G_y = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} * A$$

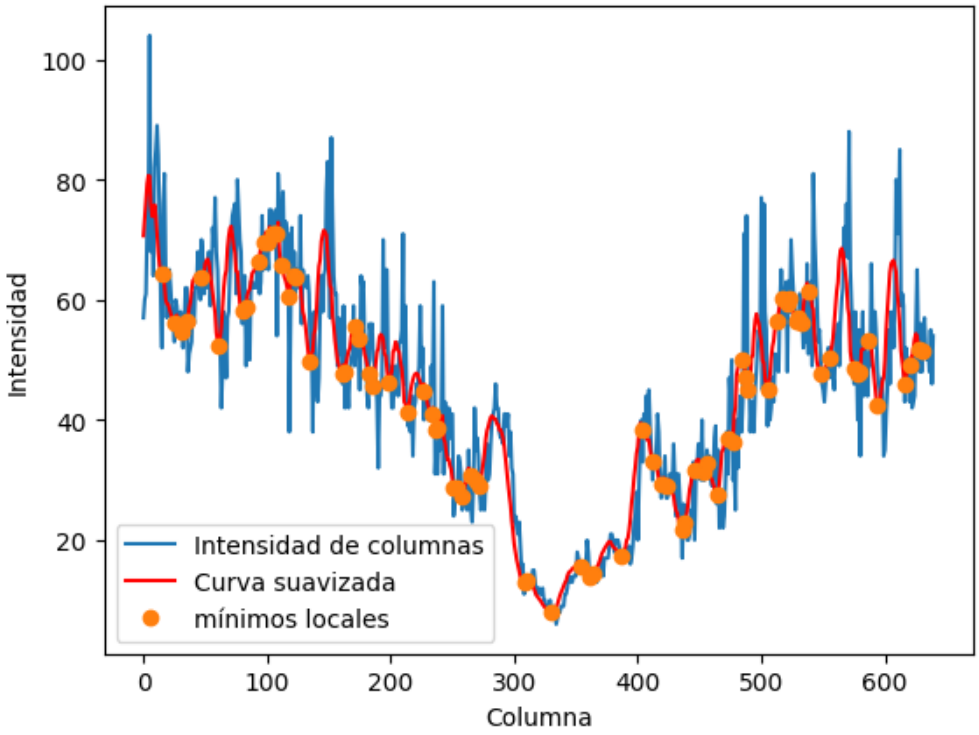
$$G = \sqrt{G_x^2 + G_y^2} \tag{7}$$

Donde A es la imagen de entrada y G es la salida.

3. Cálculo de la línea de navegación: Se se observa figura 7, en la imagen de la derecha donde solo se presentan los bordes, se puede identificar que existe una mayor densidad de bordes sobre las plantas y una menor densidad sobre el camino, entonces, si se obtiene la intensidad de los bordes por columna en la imagen y se grafica se obtiene lo siguiente:

   Como se observa en la figura 8, existe una correlación entre la intensidad de las columnas y el ángulo de navegación, pues en al curva existe un valle que corresponde con el camino. Entonces para poder calcular la ruta, se filtran los mínimos locales que corresponden con el camino y se calcula el punto medio entre ellos, entonces el ángulo de navegación vendría

Figura 8. Intensidad de bordes por columna



por lo siguiente:

$$dx = x - 0.5W \tag{8}$$

$$dy = 0.25H \tag{9}$$

$$\theta = \arctan(\frac{dx}{dy}) \tag{10}$$

Donde x es la distancia obtenida por el cálculo del punto medio del camino, W es el ancho total de la imagen, H es el alto total de la imagen y θ el ángulo de navegación. Notese

también que aplicando el método anterior se puede asegurar si el robot aún está entre dos hileras de plantas o si ya salió de las hileras, pues si la mayoría de mínimos locales pasan por el filtro se puede decir que la cámara del robot ya no está viendo plantas, por lo tanto ya debe realizar la acción de cambiar de hilera.

4. Conversión del ángulo de navegación en velocidad angular: ya obtenido el ángulo al cual el robot debe rotar, se debe obtener una velocidad angular y así, mediante la ecuación 2 controlar el robot.

   Esto se realiza de la siguiente forma, donde la velocidad angular está dada por:

$$\omega = \frac{d\theta}{dt} \tag{11}$$

   la anterior ecuación se puede representar como $\omega = k\theta$ donde θ es el valor del ángulo calculado anteriormente y $k = 1/t$, además, debido a las condiciones en las que navega el robot, cuando el ángulo $\theta = 0$ entonces $\omega = 0$, además como el robot puede rotar a una velocidad angular máxima entonces si $\omega = \omega_{max}$ entonces $\theta = \theta_{max}$, si decimos que estas dos variables varían de forma lineal, entonces:

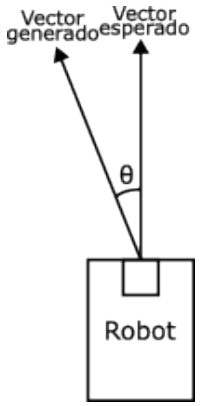
$$k = \frac{\omega_{max}}{\theta_{max}}$$

y la velocidad angular entonces sería lo siguiente:

$$\omega = \frac{\omega_{max}}{\theta_{max}}\theta \tag{12}$$

3.3. Simulación y evaluación del desempeño del robot

Figura 9. Visualización del ángulo de desviación



Para evaluar el desempeño del comportamiento del robot dentro de la simulación se medirá el error en las medidas de los vectores de velocidad generados.

La figura 9 muestra la diferencia de los ángulos entre el vector de velocidad esperado y el vector generado por el tratamiento de imagen. Se puede encontrar la desviación estándar de la medida de los ángulos de la siguiente forma:

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n}(\theta_i - \overline{\theta})^2}{n}} \tag{13}$$

Donde θ_i es el calor del ángulo en un momento dado y $\overline{\theta}$ es el promedio de todos los ángulos. De igual forma, para medir el error y la exactitud, se usará el error promedio absoluto

(MAE) que se calcula según lo siguiente:

$$MAE = \frac{\sum_{i=1}^{n} |\theta_i|}{n} \tag{14}$$

$$ACC = 1 - MAE \tag{15}$$

4. Resultados

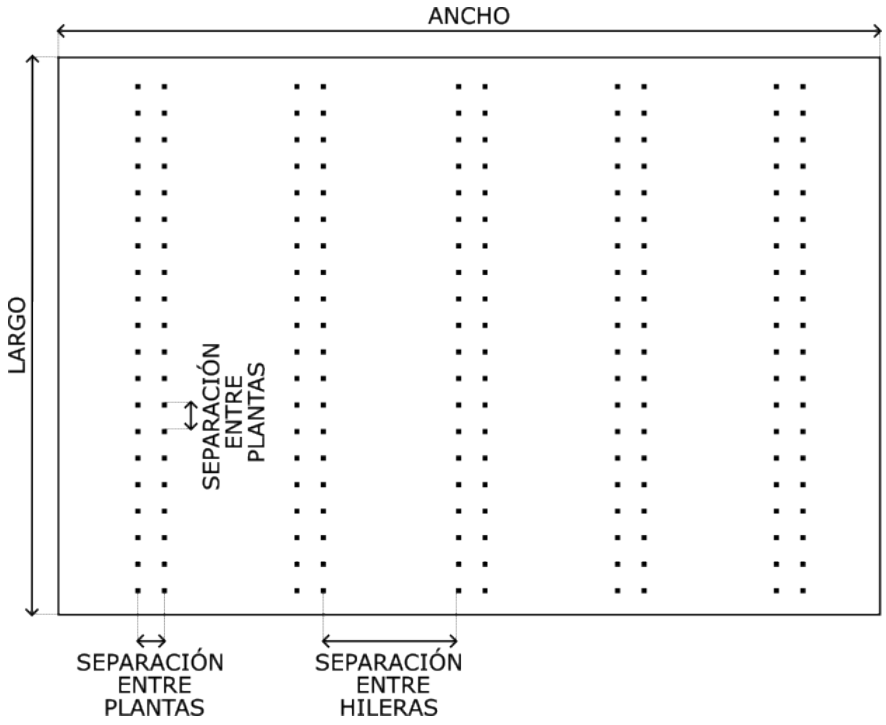
4.1. Simulación del entorno en Gazebo

La generación del mundo se realizó en 3 pasos, primero se generaron los modelos de las plantas, después, a cada modelo se le asigna uno de los modelos una coordenada en el mundo, y finalmente se crea un mundo en un archivo de extensión SDF. Para este propósito se usó el repositorio de github Polgár (2020), este genera plantas de forma procedimental de tal forma que los modelos generados son diferentes entre si.

La figura 10 muestra el esquema general de la forma en que se genera el mundo a partir de unos datos de entrada. Debido a que la implementación de la generación del mundo de Polgár está configurada para ROS1, solo se usó la parte relacionada con la generación de modelos en sí y se migró a ROS2, así mismo, el modelo se modificó de tal manera que cada linea de plantas de tomate estuviera conformada por parejas de plantas como se muestra en la figura 10.

Los datos de entrada del modelo son el número de filas, el largo de las filas, la separación entre hileras y la separación entre plantas, todo en metros. El número de filas es 8 y el largo de las mismas es de 10 metros, para establecer la separación entre filas se usó lo siguiente:

Figura 10. Esquema del mundo



$$L_f = \frac{A - L_p}{N_f - 1} - L_p \tag{16}$$

Donde L_f es la longitud entre filas, A el ancho total, que es de 20 metros, N_f es el número de hileras que ya se fijó en 8 y L_p es la distancia entre plantas, esta distancia se tiene que establecer al tanteo, pues de ella depende la densidad de los cultivos, si la distancia entre plantas es muy alta los cultivos no representarán una plantación real, pero si la distancia es muy pequeña el número de modelos se dispara siguiendo la siguiente ecuación:

$$N_p = \frac{2N_f L}{L_p} \tag{17}$$

Donde N_p es el número de plantas y L es la longitud total de la fila.

La generación de los modelos de las plantas se realizó usan scripts de python y blender, mostrado en el anexo 1, para el terreno, se usó un plano rectangular sobre el cual se generarán las plantas, todo unificado en un archivo luego cargado por gazebo. Los valores usados para la generación del mundo son los siguientes:

- Número de hileras: 8
- Plantas por hilera: 33
- Separación entre hileras: $2.5[m]$
- Separación entre plantas: $0.3[m]$
- Número total de plantas: 528

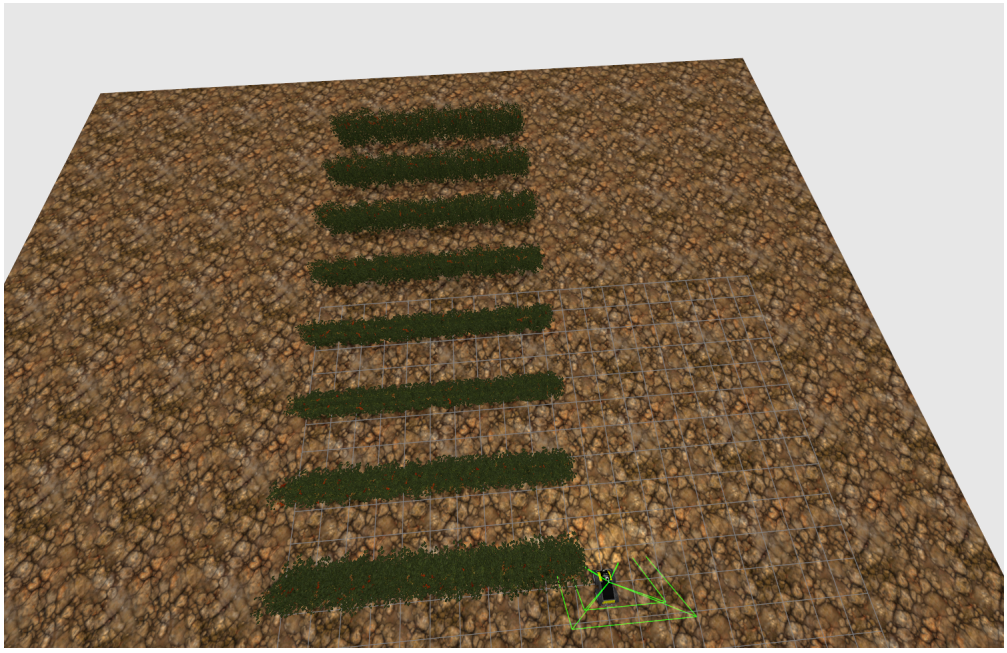
En la figura 11 se muestra un mundo ya generado dentro de gazebo, la versión del simulador que se está usando es ignition gazebo garden en el sistema operativo Ubuntu 22.04.

4.2. Árbol de comportamientos y nodos de control

La construcción del árbol de comportamientos se realizó en la aplicación Groot, esta permite diseñar grafo de forma gráfica y posteriormente exportarlo para que el paquete de ROS2 lo pueda integrar a un nodo, como se muestra en el anexo 3.

la figura 12 muestra el grafo del árbol de comportamientos diseñado para el proyecto, los nodos y sus características son los siguientes:

Figura 11. Mundo generado en Gazebo



- Root: Es el nodo raíz de árbol, este se encarga de transmitir el tick a todos los nodos siguientes.

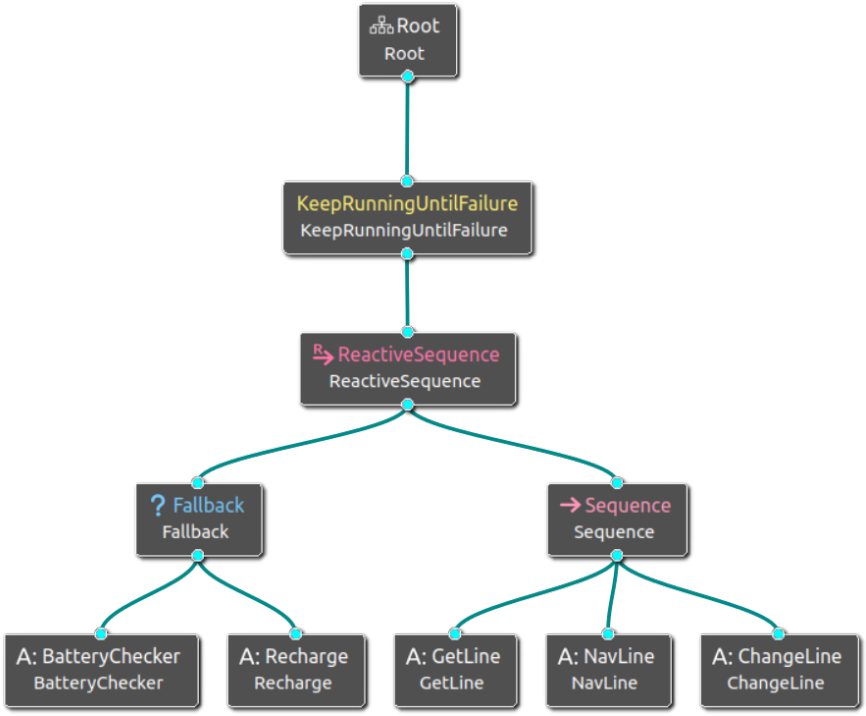
- KeepRunningUntilFailure: Este nodo es de tipo decorador, garantiza que el árbol de comportamientos siga funcionando hasta que ocurra un error.

- ReactiveSequence: Es un nodo de control que funciona como se describe en la tabla 1.

- Fallback: Este nodo de control se encarga de manejar el modelo de carga y descarga de la batería, funciona como se muestra en la tabla 1

- BatteryChecker: Este nodo de tipo acción se encarga de revisar el nivel de carga de la batería, retorna éxito si la batería se encuentra sobre el nivel mínimo y retorna falla si se encuentra por debajo del nivel mínimo que se estableció en 10 %, para esto el nodo tiene integrado las

Figura 12. Árbol de Comportamientos



ecuaciones 4 y 5.

- Recharge: Es un nodo de acción que se activa cuando el nivel de batería en bajo, una vez se activa detiene cualquier acción que el robot esté realizando el tiempo necesario hasta que la batería se recarga.

- Sequence: Este nodo controla los nodos de la parte de la navegación autónoma, se comporta como se explica en la tabla 1.

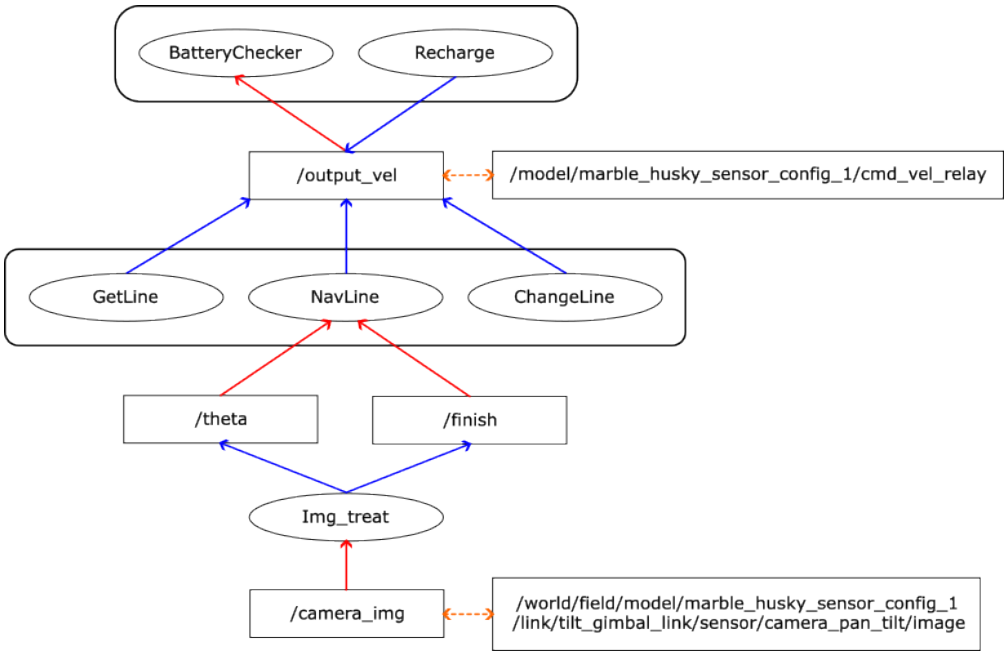
- GetLine: Es un nodo de acción que se encarga de detectar la línea de plantas y movilizar el robot hasta ella, debido a que este nodo solo posiciona el robot en la primera hilera solo se activa una vez.

- NavLine: Acá se realiza la navegación mediante visión aplicando la ecuación 12, con $\omega_{max} = 0.5[rad/s]$ y $\theta_{max} = \pi/2[rad]$.Una vez que el robot detecta que ya no está entre dos filas de plantas el nodo se desactiva.

- ChangeLine: Una vez que termina la navegación entre la lineas debido a las dimensiones en que la cámara toma fotografías aún falta una distancia por navegar entra las lineas, este nodo se encarga de terminar ese pequeño tramo y posicionar el robot para que pueda entrar a la siguiente linea.

4.3. Resultados del despliegue de la simulación y navegación

Figura 13. Grafo computacional



Una vez lanzado todos los nodos de ROS para el control del robot, se consiguió el grafo computacional mostrado en la imagen 13 y en el anexo 4 incluyendo los nodos del simulador. Se

puede observar que adicionalmente a los nodos implementados en el árbol del comportamientos se adicionó un nodo independiente denominado Img_treat descrito en profundidad en el anexo 2; este nodo se encarga de leer las imágenes tomadas por la cámara y procesarlas para luego enviar dos mensajes, el primero de ellos llamado theta es que el lleva la información del ángulo de giro, y el segundo, finish es el que lleva la información para la activación y desactivación del nodo. También del grafo computacional se puede observar que todos los nodos del árbol de comportamientos publican un mensaje al tópico output_vel, ya que este es el controla directamente el movimiento del robot.

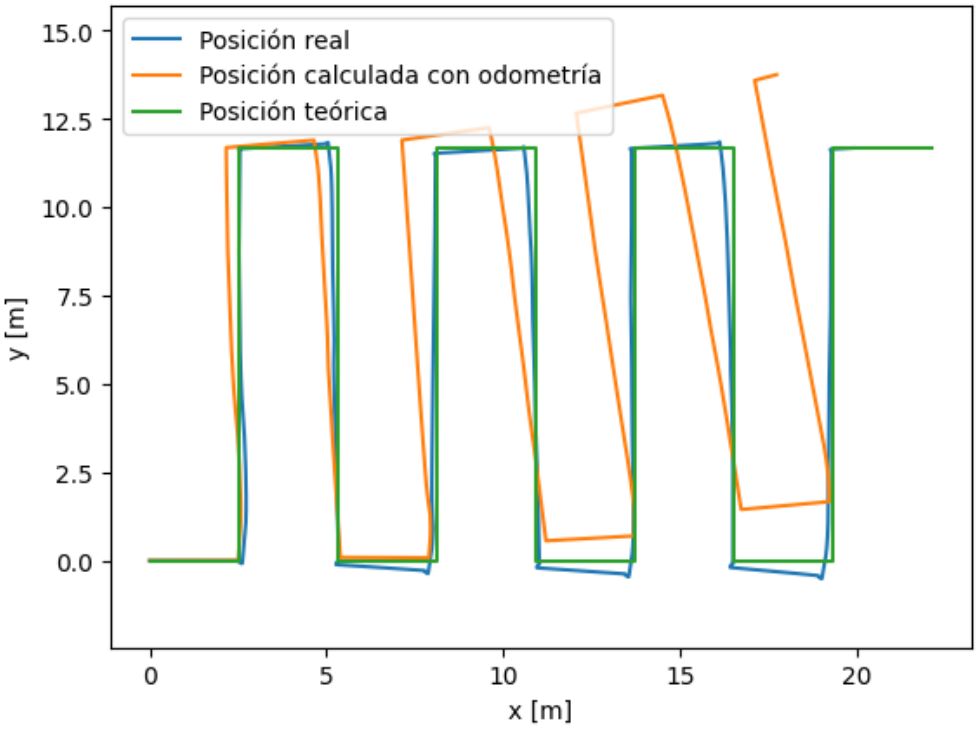
La simulación se realizón en un equipo de la marca Dell con procesador Intel Core i7-11850H y 32 gigas de RAM y los resultados del ángulo de navegación son los siguientes:

Tabla 2
Resultados obtenidos para el ángulo de desviación

	Fila 1	Fila 2	Fila 3	Fila 4	Fila 5	Fila 6	Fila 7	Total
σ	0.1011	0.123	0.185	0.0869	0.106	0.146	0.151	0.134
MAE	0.0747	0.0828	0.0922	0.0673	0.0733	0.0860	0.0815	0.0798
ACC	0.925	0.917	0.907	0.932	0.926	0.913	0.918	0.920

Como se observa en la tabla 2, se analizó fila por fila los valores de desviación estándar, error absoluto medio y de exactitud. Para la desviación estándar se obtuvo un mínimo de 0.08 y un máximo de 0.18 y en total de 0.134. Además, el error absoluto medio siempre fue menor al 10 %, siendo el máximo en la fila 3, así mismo, la exactitud siempre se mantuvo por encima del 90 %, estos resultados se ven reflejados en la imagen 14, donde se comparan las posiciones del robot durante la simulación mediante varios métodos.

Figura 14. Posición del robot durante la simulación



En la imagen 14 se comparan la posición real del robot con la posición teórica perfecta y con la estimación de la posición calculada con odometría. Como se observa, la más cercana a la posición teórica es la posición real descrita por el robot usando el método de navegación visual, presentando pequeñas desviaciones en las posiciones en las que el robot gira, pero corrigiendo la posición, de tal manera que siempre se alinea con la posición teórica.

Por otra parte, la estimación de la posición mediante odometría se ve claramente desviada

a medida que aumenta la distancia recorrida.

5. Conclusiones

- Se logró la construcción del ambiente virtual, automatizando la generación de modelos de plantas de tomate de forma procedural, también incluyendo el modelo del terreno y del robot obteniendo un archivo unificado listo para su simulación.

- El uso del árbol de comportamientos como método de control permitió dividir el algoritmo de navegación en una serie de tareas individuales, en conjunto con ROS2 como framework para la programación del comportamiento del robot se consiguió que el uso del modelo de visión por computadora no esté dentro del propio árbol de comportamientos, facilitando además que algoritmo que se usó con visión de computadora tradicional se pueda modificar y plantear uso de redes neuronales sin la necesidad de cambiar el propio árbol de comportamientos.

- El modelo de navegación presentado permitió cumplir con la navegación del entorno virtual manteniendo un error promedio del 7.98 %, sin embargo este modelo es dependiente de condiciones propias del entorno como la separación entre hileras. Además, el modelo presentado cumple mejor su tarea comparado con modelos de cálculo de odometría, que se pueden ver desviados por deslizamiento en las ruedas y son sensibles al tiempo de la simulación.

- El uso de simulación para validación de algoritmos de navegación ha demostrado sus ventajas en cuanto a costos y tiempo, pues se logró la simulación de un entorno virtual que representa de forma realista una plantación de tomate y usando modelos de robots comerciales usados en estos espacios. Sin embargo, el costo computacional de este tipo de simulaciones

suele ser alto, es por ello que es importante tener un equipo especializado y de gran potencia para poder visualizar resultados en tiempo real.

Referencias Bibliográficas

Ahmadi, A., Nardi, L., Chebrolu, N., y Stachniss, C. (2020). Visual servoing-based navigation for monitoring row-crop fields. En *2020 IEEE international conference on robotics and automation (ICRA)* (pp. 4920–4926). (ISSN: 2577-087X) doi: 10.1109/ICRA40945.2020.9197114

CEPAL, N. (2018, enero). Agenda 2030 y los Objetivos de Desarrollo Sostenible: una oportunidad para América Latina y el Caribe. Descargado 2023-04-10, de `https://repositorio.cepal.org/handle/11362/40155.4` (Accepted: 2017-04-28T14:21:49Z Publisher: CEPAL)

Chollet, F. (2017). *Deep Learning with Python*. Manning Publications Company. (Google-Books-ID: Yo3CAQAACAAJ)

Habibie, N., Nugraha, A. M., Anshori, A. Z., Ma'sum, M. A., y Jatmiko, W. (2017, diciembre). Fruit mapping mobile robot on simulated agricultural area in Gazebo simulator using simultaneous localization and mapping (SLAM). En *2017 International Symposium on Micro-NanoMechatronics and Human Science (MHS)* (pp. 1–7). (ISSN: 2474-3771) doi: 10.1109/MHS.2017.8305235

He, Z., Cheng, L., Zheng, W., Sun, M., y Yu, Q. (2017, agosto). Indoor intelligent patrol robot based on ROS architecture. En *2017 2nd International Conference on Advanced Robotics and Mechatronics (ICARM)* (pp. 294–298). doi: 10.1109/ICARM.2017.8273177

Iovino, M., Scukins, E., Styrud, J., Ögren, P., y Smith, C. (2022, agosto). A survey of Beha-

vior Trees in robotics and AI. *Robotics and Autonomous Systems*, *154*, 104096. Descargado 2023-04-27, de https://www.sciencedirect.com/science/article/pii/S0921889022000513 doi: 10.1016/j.robot.2022.104096

Karkee, M., y Zhang, Q. (Eds.). (2021). *Fundamentals of Agricultural and Field Robotics*. Cham: Springer International Publishing. Descargado 2023-04-06, de https://link.springer.com/10.1007/978-3-030-70400-1 doi: 10.1007/978-3-030-70400-1

Macenski, S., Foote, T., Gerkey, B., Lalancette, C., y Woodall, W. (2022, mayo). Robot Operating System 2: Design, architecture, and uses in the wild. *Science Robotics*, *7*(66), eabm6074. Descargado 2022-06-02, de https://www.science.org/doi/10.1126/scirobotics.abm6074 doi: 10.1126/scirobotics.abm6074

Mangal, V. (2020). *Technical description of a tomato plant – writing for the sciences portfolio.* Descargado 2023-10-09, de https://englisportfolio.commons.gc.cuny.edu/technical-description-of-a-tomato-plant/

Misión para la Transformación del Campo, D. (2015). *Diagnóstico Económico del Campo Colombiano (Informe de la Misión para la Transformación del Campo)* (Inf. Téc.). Bogotá D.C.: DNP.

Polgár, A. (2020). *Random crop field generator for ignition gazebo.* Descargado de https://github.com/azazdeaz/fields-ignition.git

Rico, F. M. (2022). *A Concise Introduction to Robot Programming with ROS2*. Boca Raton: Chapman and Hall/CRC. doi: 10.1201/9781003289623

R. Shamshiri, R., Hameed, I. A., Pitonakova, L., Weltzien, C., Balasundram, S. K., J. Yule, I., ...

Chowdhary, G. (2018). Simulation software and virtual environments for acceleration of agricultural robotics: Features highlights and performance comparison. *15-31*. Descargado 2023-04-27, de https://ntnuopen.ntnu.no/ntnu-xmlui/handle/11250/2595480 (Accepted: 2019-04-25T11:08:26Z Publisher: Chinese Society of Agricultural Engineering) doi: 10.25165/j.ijabe.20181104.4032

Rubio, F., Valero, F., y Llopis-Albert, C. (2019, marzo). A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, *16*(2), 1729881419839596. Descargado 2023-04-27, de https://doi.org/10.1177/1729881419839596 (Publisher: SAGE Publications) doi: 10.1177/1729881419839596

Song, Y., Xu, F., Yao, Q., Liu, J., y Yang, S. (2020). Navigation algorithm based on semantic segmentation in wheat fields using an RGB-d camera. *Information Processing in Agriculture*, S2214317322000488. Descargado 2023-05-24, de https://linkinghub.elsevier.com/retrieve/pii/S2214317322000488 doi: 10.1016/j.inpa.2022.05.002

Teh, C. K., Kit Wong, W., y Min, T. S. (2021, junio). Extended Dijkstra Algorithm in Path Planning for Vision Based Patrol Robot. En *2021 8th International Conference on Computer and Communication Engineering (ICCCE)* (pp. 184–189). doi: 10.1109/ICCCE50029.2021.9467157

Tian, H., Wang, T., Liu, Y., Qiao, X., y Li, Y. (2020, marzo). Computer vision technology in agricultural automation —A review. *Information Processing in Agriculture*, *7*(1), 1–19. Descargado 2023-04-27, de https://www.sciencedirect.com/science/article/

pii/S2214317319301751 doi: 10.1016/j.inpa.2019.09.006

Wen Zhu, C., Hill, E., Biglarbegian, M., Gadsden, S. A., y Cline, J. A. (2023, abril). Smart agriculture: Development of a skid-steer autonomous robot with advanced model predictive controllers. *Robotics and Autonomous Systems*, *162*, 104364. Descargado 2023-04-30, de https://linkinghub.elsevier.com/retrieve/pii/S0921889023000039 doi: 10.1016/j.robot.2023.104364

Apéndices

Apéndice A. Código para la generación del mundo

```
import numpy as np
from cookiecutter.main import cookiecutter
import collections
from pathlib import Path
import json
import shutil


SEED = 178
WORLD_NAME = 'segmentation_world' #NAME OF SDF OUTPUT FILE
MODEL_NAME_PREFIX = 'tomato' #NAME OF MODEL IN SDF FILE
 #PATH OF GENERATE MODELS AND SDF FILE
OUT_PATH = Path(Path.cwd() / '../generated/test_05').resolve()
#PATH OF BLENDER MODEL, DONT CHANGE
MODEL_TEMPLATE = Path(Path.cwd() / '../templates/tomato_model').resolve()
#PATH OF SDF TEMPLANTE, DONT CHANGE
WORLD_TEMPLATE = Path(Path.cwd() / '../templates/tomato_world').resolve()
ROW_COUNT = 3 #NUMBER OF ROWS
ROW_LENGTH = 20 #NUMBER OF PLANTS PER ROW
```

```
ROW_DIST = 2.5 #DISTANCE BETWEEN ROWS
CROP_DIST = 0.3 #DISTANCE BETWEEN PLANTS IN A ROW

shutil.rmtree(OUT_PATH, ignore_errors=True)
np.random.seed(SEED)

# helper class to build the markers.json
class Markers:
    markers = []
    last_id = 0

    @staticmethod
    def next_id():
        Markers.last_id += 1
        return Markers.last_id

    @staticmethod
    def reset():
        Markers.markers = []
```

```
@staticmethod
def add_plant(x, y, z):
    id = Markers.next_id()
    Markers.markers.append({
        'marker_type': 'PLANT',
        'id': id,
        'translation': [x, y, z]
    })
    return id


@staticmethod
def add_fruit(x, y, z, plant_id):
    id = Markers.next_id()
    Markers.markers.append({
        'marker_type': 'FRUIT',
        'id': id,
        'translation': [x, y, z],
        'plant_id': plant_id
    })
```

```
        return id


    @staticmethod
    def dumps():
        return json.dumps(Markers.markers, indent=4)


#DOUBLE THE NUMBER OF ROWS
ROW_COUNT *= 2
LL = ROW_LENGTH * CROP_DIST
print(ROW_COUNT, LL)


models = {'list': []}
Markers.reset()
#LOOP FOR GENERATE THE MODEL AND LINK IT TO THE SDF FILE
for x in range(ROW_COUNT):
    for y in range(ROW_LENGTH):
        model_name = 'tomato_{}'.format(x * ROW_LENGTH + y)


        cookiecutter(str(MODEL_TEMPLATE),
            output_dir=str(OUT_PATH),
```

```
        overwrite_if_exists=True,
        no_input=True,
        extra_context={'world_name': WORLD_NAME,
                       'model_name': model_name})
    if x%2 ==0:
        x_pos, y_pos, z_pos = x * (ROW_DIST + CROP_DIST)*0.5,
                              y * CROP_DIST, 0
    else:
        x_pos, y_pos, z_pos =
        (x-1) * (ROW_DIST + CROP_DIST)*0.5 + CROP_DIST,
        y * CROP_DIST, 0


    models['list'].append({
        'model': model_name,
        'name': model_name,
        'pose': '{} {} 0 0 0 0'.format(x_pos, y_pos)
    })
    x_pos += np.random.uniform(-0.1, 0.1)
    y_pos += np.random.uniform(-0.1, 0.1)
```

```
seed = np.random.randint(10000)
dir = (OUT_PATH / WORLD_NAME / model_name).resolve()
dir_blender = (Path.cwd() / '../blender').resolve()
blend = str(dir_blender / 'tomato_gen.blend')
script = str(dir_blender / 'tomato_gen.py')
! blender $blend --background --python $script --
--model_dir $dir --seed $seed


plant_id = Markers.add_plant(x_pos, y_pos, z_pos)
with open(dir / 'markers.json') as markers_file:
    plant_markers = json.load(markers_file)
    for marker in plant_markers:
        if marker['marker_type'] == 'FRUIT':
            Markers.add_fruit(
                marker['translation'][0] + x_pos,
                marker['translation'][1] + y_pos,
                marker['translation'][2] + z_pos,
                plant_id
            )
```

```
cookiecutter(str(WORLD_TEMPLATE),
             output_dir=str(OUT_PATH),
             overwrite_if_exists=True,
             no_input=True,
             extra_context={'world_name': WORLD_NAME, 'models': models})

with open(OUT_PATH / WORLD_NAME / 'markers.json', 'w') as outfile:
    json.dump(Markers.markers, outfile, indent=4, sort_keys=True)
```

El código fue adaptado directamente del repositorio de Polgár (2020).

Apéndice B. Nodo de implementación de visión por computadora en ROS

```
import rclpy
from rclpy.node import Node
from rclpy.qos import qos_profile_sensor_data
from sensor_msgs.msg import Image
from std_msgs.msg import Float64, Bool
from cv_bridge import CvBridge
import cv2
import numpy as np
from scipy import signal


#CREATE A NODE CLASS
class Img_Node_CV(Node):
    def __init__(self):
        super().__init__('img_node')
        #SUBSCRIPTION TO THE IMG TOPIC AND
        #PUBLISHES THETA AND FINISH TOPICS
        self.img_sub = self.create_subscription(Image,
        "/camera_img", self.img_callback, qos_profile_sensor_data)
        self.theta_pub = self.create_publisher(Float64, "/theta", 100)
```

```
        self.finish_pub = self.create_publisher(Bool, "/finish", 100)
        self.cv_bridge = CvBridge()


    def img_treatment(self, img):
        #OPEN CV INTEGRATION WITH ROS APLIYING CANNY FUNCITON
        edges = cv2.Canny(img, 100, 200, None, 3, cv2.DIST_L2)
        normalized = cv2.normalize(edges, None, alpha=0, beta=1,
        norm_type=cv2.NORM_MINMAX)
        column_intensity = normalized.sum(axis=0)
        window_size = 9
        window = np.ones((window_size,)) / window_size
        smoothed = np.convolve(column_intensity, window, mode="valid")
        indices = signal.argrelmin(smoothed)[0]
        mins = smoothed[indices]
        x_filter = indices[smoothed[indices]<20]
        if x_filter.shape[0]>0:
            x_3 = (x_filter[0] + x_filter[-1])*0.5
        else:
            x_3 = 0
        return x_3, np.mean(mins)
```

```
def img_callback(self, img_msg):
    #READ THE IMAGE WITH OPENCV BRIDGE
    img = self.cv_bridge.imgmsg_to_cv2(img_msg, "bgr8")
    #APPLY CANNY AND OTHER OPERATIONS ANG GET THE VECTOR DATA
    x, status = self.img_treatment(img)
    height = img.shape[0]
    width = img.shape[1]
    dy = 0.25*height
    dx = x - 0.5*width
    theta = np.arctan2(dx,dy)
    #PUBLIS THE DATA
    theta_msg = Float64()
    theta_msg.data = float(theta)
    finish_msg = Bool()
    if(status < 20):
        finish_msg.data = True
    else:
        finish_msg.data = False
    self.theta_pub.publish(theta_msg)
    self.finish_pub.publish(finish_msg)
```

Apéndice C. Nodo de implementación del árbol de comportamientos

Archivo de cabecera construyendo la clase el nodo BatteryChecker:

```
#ifndef NAV_CONTROL__BATTERYCHECKER_HPP_
#define NAV_CONTROL__BATTERYCHECKER_HPP_
#include <string>
#include <vector>
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "behaviortree_cpp_v3/bt_factory.h"
#include "geometry_msgs/msg/twist.hpp"
#include "rclcpp/rclcpp.hpp"

namespace nav_control{

class BatteryChecker : public BT::ConditionNode
{
    public:
        explicit BatteryChecker(
            const std::string & xml_tag_name,
            const BT::NodeConfiguration & conf
        );
```

```
        BT::NodeStatus tick();
        static BT::PortsList providedPorts(){
            return BT::PortsList({});
        };
        void
        vel_callback(const geometry_msgs::msg::Twist::SharedPtr msg);
        const float DECAY_LEVEL = 0.5;
        const float EPSILON = 0.01;
        const float MIN_LEVEl = 10.0;

    private:
        void update_battery();
        rclcpp::Node::SharedPtr node_;
        rclcpp::Time last_reading_time_;
        geometry_msgs::msg::Twist last_twist_;
        rclcpp::Subscription<geometry_msgs::msg::Twist>
        ::SharedPtr vel_sub_;
};
}
#endif
```

Archivo de cabecera construyendo la clase el nodo Recharge:

```
#ifndef NAV_CONTROL__RECHARGE_HPP_
#define NAV_CONTROL__RECHARGE_HPP_
#include <string>
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "behaviortree_cpp_v3/bt_factory.h"
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
namespace nav_control{
class Recharge : public BT::ActionNodeBase{
    public:
        explicit Recharge(
            const std::string & xml_tag_name,
            const BT::NodeConfiguration & conf
        );
        void halt();
        BT::NodeStatus tick();
        static BT::PortsList providedPorts(){
            return BT::PortsList({});
        };
```

```
    private:
        int counter_;
        rclcpp::Node::SharedPtr node_;
        rclcpp::Publisher<geometry_msgs::msg::Twist>
        ::SharedPtr vel_pub_;
};
}
#endif
```

Archivo de cabecera construyendo la clase el nodo GetLine:

```
#ifndef NAV_CONTROL__GETLINE_HPP_
#define NAV_CONTROL__GETLINE_HPP_
#include <string>
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "behaviortree_cpp_v3/bt_factory.h"
#include "geometry_msgs/msg/twist.hpp"
#include "std_msgs/msg/bool.hpp"
#include "rclcpp/rclcpp.hpp"
namespace nav_control{
class GetLine : public BT::ActionNodeBase{
    public:
```

```
        explicit GetLine(
            const std::string & xml_tag_name,
            const BT::NodeConfiguration & conf
        );
        void halt();
        BT::NodeStatus tick();
        static BT::PortsList providedPorts(){
            return BT::PortsList({});
        };
    private:
        rclcpp::Node::SharedPtr node_;
        rclcpp::Publisher<geometry_msgs::msg::Twist>
        ::SharedPtr vel_pub_;
        rclcpp::Time start_time_;
        static bool status_;
};
}
#endif
```

Archivo de cabecera construyendo la clase el nodo NavLine:

```
#ifndef NAV_CONTROL__NAVLINE_HPP_
```

```
#define NAV_CONTROL__NAVLINE_HPP_

#include <string>

#include <fstream>

#include "behaviortree_cpp_v3/behavior_tree.h"

#include "behaviortree_cpp_v3/bt_factory.h"

#include "rclcpp/rclcpp.hpp"

#include "geometry_msgs/msg/twist.hpp"

#include "std_msgs/msg/bool.hpp"

#include "std_msgs/msg/float64.hpp"

namespace nav_control{

class NavLine : public BT::ActionNodeBase{

    public:

        explicit NavLine(

            const std::string & xml_tag_name,

            const BT::NodeConfiguration & conf

        );

        void halt();

        BT::NodeStatus tick();

        static BT::PortsList providedPorts(){

            return BT::PortsList({});
```

```
        };
        void theta_callback(
        const std_msgs::msg::Float64::SharedPtr theta);
        void status_callback(
        const std_msgs::msg::Bool::SharedPtr status_node);
        const float THETA_MAX = 1.57079;
        const float OMEGA_MAX = 0.5;
    private:
        rclcpp::Node::SharedPtr node_;
        rclcpp::Publisher<geometry_msgs::msg::Twist>
        ::SharedPtr vel_pub_;
        rclcpp::Subscription<std_msgs::msg::Float64>
        ::SharedPtr theta_sub_;
        rclcpp::Subscription<std_msgs::msg::Bool>
        ::SharedPtr status_sub_;
        std_msgs::msg::Float64 last_theta_reading;
        std_msgs::msg::Bool last_status_reading;
        std::ofstream CSVfile;
        static int n_line_;
};
```

```
}

#endif
```

Archivo de cabecera construyendo la clase el nodo ChangeLine:

```
#ifndef NAV_CONTROL__CHANGELINE_HPP_
#define NAV_CONTROL__CHANGELINE_HPP_
#include <string>
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "behaviortree_cpp_v3/bt_factory.h"
#include "geometry_msgs/msg/twist.hpp"
#include "std_msgs/msg/bool.hpp"
#include "rclcpp/rclcpp.hpp"
namespace nav_control{
class ChangeLine : public BT::ActionNodeBase{
    public:
        explicit ChangeLine(
            const std::string & xml_tag_name,
            const BT::NodeConfiguration & conf
        );
        void halt();
        BT::NodeStatus tick();
```

```
        static BT::PortsList providedPorts(){
            return BT::PortsList({});
        };
        const double PI = 3.141592;
        const double D1 = 2.5;
        const double D2 = 2.8;
        const double D3 = 0.5;
    private:
        rclcpp::Node::SharedPtr node_;
        rclcpp::Time start_time_;
        rclcpp::Publisher<geometry_msgs::msg::Twist>::SharedPtr vel_pub_;
        static int direction_;
};
}
#endif
```

Archivo definiendo los métodos de la clase en el nodo BatteryChecker:

```
#include <string>
#include <iostream>
#include <algorithm>
#include "nav_control/BatteryChecker.hpp"
```

```
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "geometry_msgs/msg/twist.hpp"
#include "rclcpp/rclcpp.hpp"
namespace nav_control{
    using namespace std::chrono_literals;
    using namespace std::placeholders;
    BatteryChecker::BatteryChecker(
        const std::string & xml_tag_name,
        const BT::NodeConfiguration & conf)
        : BT::ConditionNode(xml_tag_name, conf){
        config().blackboard->get("node", node_);
        vel_sub_ = node_->create_subscription
        <geometry_msgs::msg::Twist>(
            "/output_vel", 100, std::bind(
            &BatteryChecker::vel_callback, this, _1)
        );
        last_reading_time_ = node_->now();
    }
```

```
void BatteryChecker::vel_callback(
const geometry_msgs::msg::Twist::SharedPtr msg){
    last_twist_ = *msg;
}
void BatteryChecker::update_battery()
{
    float battery_level;
    if(!config().blackboard->get("battery_level", battery_level)){
        battery_level = 100.0f;
    }
    float dt = (node_->now() - last_reading_time_).seconds();
    last_reading_time_ = node_->now();
    float vel = sqrt(
        last_twist_.linear.x*last_twist_.linear.x +
        last_twist_.angular.z*last_twist_.angular.z
    );
    battery_level = std::max(
    0.0f, battery_level-(vel*dt*DECAY_LEVEL)- EPSILON*dt);
    config().blackboard->set("battery_level", battery_level);
}
```

```
    BT::NodeStatus BatteryChecker::tick(){
        update_battery();
        float battery_level;
        config().blackboard->get("battery_level", battery_level);
        std::cout <<"Battery Level: "<<battery_level<< std::endl;
        if(battery_level<MIN_LEVEl){
            return BT::NodeStatus::FAILURE;
        }
        else{
            return BT::NodeStatus::SUCCESS;
        }
    }
}
#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory){
    factory.registerNodeType<nav_control::BatteryChecker>("BatteryChecker")
}
```

Archivo definiendo los métodos de la clase en el nodo Recharge:

```
#include <iostream>
#include <string>
```

```
#include <set>
#include "nav_control/Recharge.hpp"
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
namespace nav_control{
    Recharge::Recharge(
        const std::string & xml_tag_name,
        const BT::NodeConfiguration & conf
    )
    : BT::ActionNodeBase(xml_tag_name, conf), counter_(0){
        config().blackboard->get("node", node_);
        vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>
        ("/output_vel", 100);
    }
    void Recharge::halt(){
    }
    BT::NodeStatus Recharge::tick(){
        std::cout<<"Recharge node active"<< counter_ << std::endl;
        geometry_msgs::msg::Twist vel_msg_;
```

```
            vel_msg_.linear.x = 0;
            vel_msg_.angular.z = 0;
            vel_pub_->publish(vel_msg_);
            if (counter_++ < 50)
            {
                return BT::NodeStatus::RUNNING;
            }
            else{
                counter_ = 0;
                config().blackboard->set<float>("battery_level", 100.0f);
                return BT::NodeStatus::SUCCESS;
            }
        }
}
#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory){
    factory.registerNodeType<nav_control::Recharge>("Recharge");
}
```

Archivo definiendo los métodos de la clase en el nodo GetLine:

```
#include <iostream>
```

```
#include <string>
#include <set>
#include "nav_control/GetLine.hpp"
#include "std_msgs/msg/bool.hpp"
#include "behaviortree_cpp_v3/behavior_tree.h"
namespace nav_control{
    using namespace std::placeholders;
    bool GetLine::status_ = false;
    GetLine::GetLine(
        const std::string & xml_tag_name,
        const BT::NodeConfiguration & conf
    )
    : BT::ActionNodeBase(xml_tag_name, conf){
        config().blackboard->get("node", node_);
        vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>
        ("/output_vel", 100);
    }
    void GetLine::halt(){
    }
    BT::NodeStatus GetLine::tick(){
```

```
if(status_){
    return BT::NodeStatus::SUCCESS;
}

std::cout<<"Get␣Line␣Node␣Active"<< std::endl;
if (status() == BT::NodeStatus::IDLE){
start_time_ = node_->now();
}

geometry_msgs::msg::Twist vel_msg;
auto elpased = node_->now() - start_time_;

if (elpased < rclcpp::Duration(10.2415, 0)){
    if (elpased < rclcpp::Duration(5.1, 0)){
        vel_msg.linear.x = 0.5;
        vel_pub_->publish(vel_msg);
        return BT::NodeStatus::RUNNING;
    }
    else if (elpased < rclcpp::Duration(8.2415, 0)){
```

```
                vel_msg.angular.z = 0.5;
                vel_pub_->publish(vel_msg);
                return BT::NodeStatus::RUNNING;
            }
            else{
                vel_msg.linear.x = 0.5;
                vel_pub_->publish(vel_msg);
                return BT::NodeStatus::RUNNING;
            }
        }
        else{
            status_ = true;
            return BT::NodeStatus::SUCCESS;
        }
    }
}
#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory){
    factory.registerNodeType<nav_control::GetLine>("GetLine");
}
```

Archivo definiendo los métodos de la clase en el nodo NavLine:

```
#include <iostream>
#include <string>
#include <set>
#include <algorithm>
#include <fstream>
#include "nav_control/NavLine.hpp"
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "rclcpp/rclcpp.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "std_msgs/msg/bool.hpp"
#include "std_msgs/msg/float64.hpp"
namespace nav_control{
    using namespace std::placeholders;
    int NavLine::n_line_ = 0;
    NavLine::NavLine(
        const std::string & xml_tag_name,
        const BT::NodeConfiguration & conf
    )
    : BT::ActionNodeBase(xml_tag_name, conf){
```

```
    config().blackboard->get("node",node_);
    vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>
    ("/output_vel", 100);
    theta_sub_ = node_->create_subscription<std_msgs::msg::Float64>
    (
        "/theta", 100, std::bind(
        &NavLine::theta_callback, this, _1));
    status_sub_ = node_->create_subscription<std_msgs::msg::Bool
    ( "/finish", 100, std::bind(
    &NavLine::status_callback, this, _1));

}
void
NavLine::theta_callback(
const std_msgs::msg::Float64::SharedPtr theta){
    last_theta_reading = *theta;
}
void
NavLine::status_callback(
const std_msgs::msg::Bool::SharedPtr status_node){
```

```
        last_status_reading = *status_node;
    }
    void NavLine::halt(){
    }
    BT::NodeStatus NavLine::tick(){
        std::cout<<"Nav Line Node Active" <<std::endl;
        geometry_msgs::msg::Twist vel_msg;
        if (!last_status_reading.data)
        {
            vel_msg.angular.z = -1 *
                                last_theta_reading.data *
                                OMEGA_MAX / THETA_MAX;
            vel_msg.linear.x = 0.5;
            vel_pub_->publish(vel_msg);
            std::string File_Name = "theta_data_";
            File_Name += std::to_string(n_line_);
            File_Name += ".csv";
            CSVfile.open(File_Name, std::ios::out | std::ios::app);
            CSVfile << last_theta_reading.data <<"\n";
            CSVfile.close();
```

```
            return BT::NodeStatus::RUNNING;
        }
        else{
            n_line_++;
            return BT::NodeStatus::SUCCESS;
        }
    }
}
#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory){
    factory.registerNodeType<nav_control::NavLine>("NavLine");
}
```

Archivo definiendo los métodos de la clase en el nodo ChangeLine:

```
#include <iostream>
#include <string>
#include <set>
#include "nav_control/ChangeLine.hpp"
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "geometry_msgs/msg/twist.h"
```

```
#include "rclcpp/rclcpp.hpp"
namespace nav_control{
    int ChangeLine::direction_ = -1;
    ChangeLine::ChangeLine(
        const std::string & xml_tag_name,
        const BT::NodeConfiguration & conf
    )
    : BT::ActionNodeBase(xml_tag_name, conf){
        config().blackboard->get("node", node_);
        vel_pub_ = node_->create_publisher<geometry_msgs::msg::Twist>
        ("/output_vel", 100);
    }
    void ChangeLine::halt(){
    }
    BT::NodeStatus ChangeLine::tick(){
        std::cout<<"Change Line Node Active"<< std::endl;
        if (status() == BT::NodeStatus::IDLE)
        {
            start_time_ = node_->now();
        }
```

```
geometry_msgs::msg::Twist vel_msg;
auto elpased = node_->now() - start_time_;

if (elpased < rclcpp::Duration(2*(D1+D2+D3+PI), 0)){

    if(elpased < rclcpp::Duration(2*D1, 0)){
        vel_msg.linear.x = 0.5;
        vel_pub_->publish(vel_msg);
        return BT::NodeStatus::RUNNING;
    }
    else if (elpased < rclcpp::Duration(2*D1+PI, 0)){
        vel_msg.angular.z = direction_*0.5;
        vel_pub_->publish(vel_msg);
        return BT::NodeStatus::RUNNING;
    }
    else if (elpased < rclcpp::Duration(2*(D1+D2)+PI, 0)){
        vel_msg.linear.x = 0.5;
        vel_pub_->publish(vel_msg);
        return BT::NodeStatus::RUNNING;
```

```
            }
            else if (elpased < rclcpp::Duration(2*(D1+D2+PI), 0)){
                vel_msg.angular.z = direction_*0.5;
                vel_pub_->publish(vel_msg);
                return BT::NodeStatus::RUNNING;
            }
            else{
                vel_msg.linear.x = 0.5;
                vel_pub_->publish(vel_msg);
                return BT::NodeStatus::RUNNING;
            }
        }
        else{
            direction_ *= -1;
            return BT::NodeStatus::SUCCESS;
        }
    }
}
#include "behaviortree_cpp_v3/bt_factory.h"
BT_REGISTER_NODES(factory){
```

```
    factory.registerNodeType<nav_control::ChangeLine>("ChangeLine");
}
```

Archivo creando el nodo en ROS para la implementación de los diferentes nodos del árbol de comportamientos:

```
#include <string>
#include <memory>
#include "behaviortree_cpp_v3/behavior_tree.h"
#include "behaviortree_cpp_v3/bt_factory.h"
#include "behaviortree_cpp_v3/utils/shared_library.h"
#include "behaviortree_cpp_v3/loggers/bt_zmq_publisher.h"
#include "ament_index_cpp/get_package_share_directory.hpp"
#include "rclcpp/rclcpp.hpp"

int main(int args, char * argv[]){
    rclcpp::init(args, argv);

    auto node = rclcpp::Node::make_shared("nav_control_node");
    BT::BehaviorTreeFactory factory;
    BT::SharedLibrary loader;
    factory.registerFromPlugin(loader.getOSName(
```

```
"nav_control_battery_checker_bt_node"));
factory.registerFromPlugin(loader.getOSName(
"nav_control_recharge_bt_node"));
factory.registerFromPlugin(loader.getOSName(
"nav_control_get_line_bt_node"));
factory.registerFromPlugin(loader.getOSName(
"nav_control_nav_line_bt_node"));
factory.registerFromPlugin(loader.getOSName(
"nav_control_change_line_bt_node"));
std::string pkpath = ament_index_cpp::get_package_share_directory(
"nav_control");
std::string xml_file = pkpath +
                        "/behavior_tree_xml/nav_control_bt.xml";

auto blackboard = BT::Blackboard::create();
blackboard->set("node", node);
BT::Tree tree = factory.createTreeFromFile(xml_file, blackboard);
auto publisher_zmq = std::make_shared<BT::PublisherZMQ>(
tree, 10, 2666, 2667);
rclcpp::Rate rate(10);
```

```
    bool finish = false;


    while (!finish && rclcpp::ok())
    {
        finish = tree.rootNode()->executeTick() ==
                 BT::NodeStatus::SUCCESS;
        rclcpp::spin_some(node);
        rate.sleep();
    }
    rclcpp::shutdown();
    return 0;
}
```

Además, todo el código está disponible en el siguiente repositorio en github:

https://github.com/angeloob11/VisualNavigation.git

Apéndice D. Arquitectura de la simulación en conjunto con ROS2

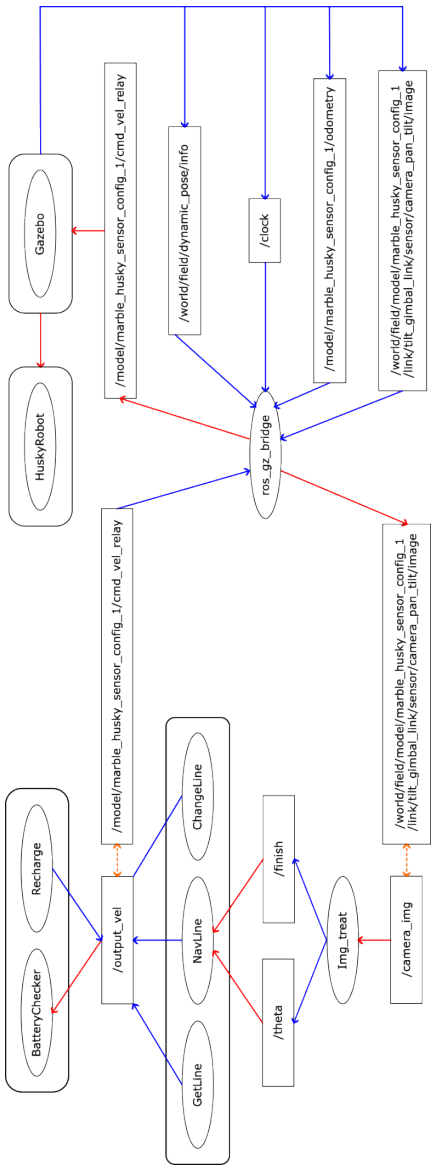


Figura 15. Grafo computacional en conjunto con el simulador Gazebo