



Reporte: Proyecto Final

Luis Alberto Fernández., Luis Dario Hinojosa., Luis
Felipe Flores.

Department: Computación

Course: TE3001B - Fundamentación de Robótica

Instructor: Aldo Ivan Aldecoa, PhD.

Date: June 18, 2022

Abstract

En el siguiente trabajo se explican a detalle los aspectos fundamentales de nuestra implementación del Puzzlebot, así como los problemas encontrados y las soluciones propuestas.

1 Introducción

Hoy en día la robótica se ha vuelto un pilar importante en cuestión a los avances tecnológicos y la satisfacción de necesidades que puede resolver para hacer más eficaz un proceso o una tarea determinada, abarcando muchas otras disciplinas como la programación, mecánica y electrónica, entre otros, siendo una ciencia que reúne muchos campos involucrados con innovación tecnológica, donde en la actualidad es un concepto muy amplio en búsqueda de la facilidad de la sociedad y la mejora continua por medio de robots inteligentes y autónomos. En este escrito se pretende explicar la interacción de un robot diferencial en búsqueda de autonomía y toma de decisiones por medio de sensores y actuadores para involucrarse más en la implementación de un sistema de control inteligente y robusto.(OpenWebinars 2017)

2 Marco Teorico

2.1 Robots Diferenciales

Los robots diferenciales usualmente son una plataforma móvil de tracción diferencial esta cuenta con dos pares de ruedas: dos ruedas de tracción que tienen acoplados dos motores DC, dos ruedas de estabilización que mantienen el balance del vehículo. La translación y rotación de este tipo de plataformas diferenciales se determinan por el movimiento independiente de cada una de las ruedas de tracción. La tracción diferencial se refiere al hecho de que el vector de movimiento del robot es la suma de los movimientos de las ruedas independientes. Las ruedas motrices generalmente se colocan a cada lado del robot y hacia el frente.

2.1.1 Odometría en Robots Diferenciales

La odometría en robots diferenciales es simplemente el sistema que permite localizar al robot en un entorno determinado; este es uno de los sistemas más usuales ya que se basa en el uso de ruedas de tracción diferencial. Las configuraciones de tracción diferencial, son muy populares y permiten calcular la posición

del robot a partir de las ecuaciones geométricas, que surgen de la relación entre los componentes del sistema de propulsión y de la información de los codificadores rotativos que comúnmente llevan acoplados a sus ruedas, este método de localización se conoce como odometría.

2.1.2 Control posicional y rotacional en Robots Diferenciales

En el control posicional y rotacional en los robots diferenciales permite de forma autónoma movimientos previamente planificados, en este mismo se pretende que el robot móvil siga un camino específico de forma autónoma. Este problema puede formularse como la obtención de las leyes de control que permitan estabilizar al robot sobre un punto de trabajo. En un problema de seguimiento de caminos explícitos se pretende que el error entre la posición deseada y la posición actual $q_d(t) - q(t)$ tienda a cero (0), manteniéndose acotadas las señales de control. En el control posicional: su objetivo es calcular las velocidades de las llantas izquierda y derecha (en función del error) necesarias para que el robot llegue a la posición deseada.

2.2 Visión Computacional

La Vision Artificial Computacional es un campo de la IA que permite que las computadoras y los sistemas obtengan información significativa como imágenes, videos o entradas visuales, de tal forma que a partir de esta información se tomen acciones y decisiones basados en estos datos. La visión artifical pretende funcionar con similaridad a la vista humana, entrena máquinas para procesar información de manera eficaz y efectiva, ejecutando análisis de datos una y otra vez hasta identificar diferencias para finalmente reconocer imágenes. Para la sección de aprendizaje y se tenga en ventaja la parte de experiencia, se toma en consideración el concepto de DeepLearning y una neuronal convolucional (CNN), de tal forma que el aprendizaje automático utiliza modelos algorítmicos que permite que una computadora se enseñe a si misma en la diferencia de que alguien programe para reconocer una imagen. De igual forma estos conceptos de Vision Artifial, IA y Deep Learning están estrechamente relacionados. (IBM 2022)

2.2.1 Operadores morfológicos (Gradiente y Erosión)

Las transformaciones morfológicas son algunas operaciones simples basadas en la forma de la imagen, que normalmente se aplican a imágenes binarias. Necesita dos entradas, una es nuestra imagen original, la segunda se llama elemento estructurante o núcleo (kernel) que decide la naturaleza de la operación. Estos son algunos ejemplos de operadores:

1.-Erosión: Similar a la convolución 2D, durante la erosión, un kernel se desliza a través de la imagen. El píxel de la imagen original (1 o 0) solo se considerará 1 si todos los píxeles de la ventana del kernel son 1; de lo contrario, se erosionará (se establecerá en 0). Por lo tanto, todos los píxeles cerca de los bordes de los objetos de la imagen se descartarán según el tamaño del núcleo. Como resultado, disminuye el grosor o el tamaño de los objetos en primer plano, es decir, disminuyen las áreas blancas de la imagen. Este procedimiento es útil para eliminar pequeños ruidos blancos, separar dos objetos conectados, etc.

2.-Gradiente: Es la diferencia entre la dilatación y la erosión de una imagen. El resultado se verá como el contorno del objeto. El gradiente morfológico resalta las transiciones bruscas entre niveles de grises de la imagen, estos usan elementos estructurales simétricos y dependen menos de la direccionalidad de los bordes.

2.2.2 Campos de color RGB y HSV

Un campo de color establece un conjunto de colores primarios a partir de los que mediante mezclas, se pueden obtener otros colores hasta cubrir todo el espectro visible. El modelo RGB está formado por los tres componentes de colores primarios aditivos, estos componentes son el azul, rojo y verde. Este modelo se utiliza cuando se representa color mediante haces de luz (pantallas o monitores). Un pixel en un monitor se representaría mediante tres subpíxeles o células: una roja, una verde y una azul, correspondiendo cada una a un LED o diodo emisor de luz del respectivo color. El modelo HSV o HSB (Hue, Saturation, Brightness – Tonalidad, Saturación, Brillo); deriva del espacio RGB y representa los colores combinando tres valores: el tono en sí (H), la saturación o cantidad de color (S) y el brillo del mismo (B). Estos valores suelen representarse en un diagrama circular (principal uso de este modelo). Estas tres magnitudes pueden

tener los siguientes valores:

- H (color en concreto): Valores de 0-360º. La gama cromática se representa en una rueda circular y este valor expresa su posición.
- S (Saturación): Valores de 0-100. De menos a más cantidad de color.
- B (Brillo). Valores de 0-100. De totalmente oscuro a la máxima luminosidad.

2.3 Redes Neuronales

Las redes neuronales artificiales son un modelo inspirado en el funcionamiento del cerebro humano. Uno de los objetivos principales para este modelo es aprender modificándose automáticamente, a si mismo de forma que puede llegar a realizar tareas complejas que no podrían ser realizadas mediante la clásica programación basada en reglas. El funcionamiento de las redes neuronales se asemeja al cerebro; estas redes reciben una serie de valores de entrada y cada una de estas entradas llega a un nodo llamado neurona. Las neuronas de la red están a su vez agrupadas en capas que forman la red neuronal. Cada una de las neuronas de la red posee a su vez un peso, un valor numérico, con el que modifica la entrada recibida. Los nuevos valores obtenidos salen de las neuronas y continúan su camino por la red. El alcance de las funciones de las redes neuronales es muy amplio, debido a su funcionamiento, son capaces de aproximar cualquier función existente con sus debidos detalles, estas a su vez son utilizadas principalmente para tareas de predicción y clasificación.

2.3.1 Redes Convolucionales (CNN)

Las redes convolucionales es un algoritmo específicamente diseñado para trabajar con imágenes, tomándolas como entrada, asignando importancia a ciertos elementos de la imagen para distinguirlos entre sí. Estas redes tienen como objetivo aprender a reconocer una diversidad de objetos dentro de imágenes, pero para ello necesitan entrenarse de previo con una cantidad importante de muestras un ejemplo puede ser que lean más de 10.000, de ésta forma las neuronas de la red van a poder captar las características únicas de cada objeto y a su vez, poder generalizarlo a esto es lo que se le conoce como el proceso de

"aprendizaje de un algoritmo". Con esto, la red va a poder reconocer un cierto tipo de célula porque ya la ha visto anteriormente muchas veces, pero no solo buscará celulas semejantes sino que podrá inferir imágenes que no conozca pero que relaciona y en donde podrían existir similitudes, y está es la parte inteligente del conocimiento.

2.3.2 Clasificación de imágenes por CNN

La clasificación de imágenes por CNN se ingresan en parte por tres diferentes secciones de extracción de características cada una con cinco capas: la primera sección utiliza un kernel de tamaño 3×3 ; la segunda sección utiliza un kernel de tamaño 5×5 , en la siguiente capa el kernel se reduce a 3×3 y se repite en las subsecuentes; en la tercera sección se inicia con un kernel de 7×7 , posteriormente en la siguiente capa se reduce a 5×5 , después a 3×3 ; y es así como se obtiene la información representativa de la imagen por cada kernel a lo largo de tres diferentes secciones. Finalmente, se utilizan tres capas completamente conectadas para la clasificación.

2.4 Sistemas de control tipo PID

Un controlador PID (proporcional integrativo derivativo); es el encargado de modular la señal de control del sistema de acuerdo al error entre el valor medido y el valor deseado. Este tipo de control es ampliamente utilizado en los procesos industriales, incluso cuando se presenten interferencias externas. Asimismo, resulta fácil de implementar y puede ser utilizado en todo tipo de hardware de manera eficiente debido a que no utiliza muchos recursos. La sintonización de un controlador PID consiste en la determinación de los parámetros, cuya única finalidad resulta en el comportamiento del sistema de control aceptable y robusto de conformidad con algún criterio de desempeño establecido.

2.4.1 Acción de control Proporcional

La acción de control proporcional produce una salida del controlador basada en el error presentado por el sistema; en otras palabras es en realidad un amplificador

con ganancia ajustable. Este control reduce el tiempo de subida, incrementa el sobretiro y reduce el error de estado estable.

2.4.2 Acción de control Diferencial

La acción de control diferencial predice el error y toma medidas oportunas para corregirlo. Esta acción reacciona a la velocidad de la entrada y ajusta la señal de salida, por lo que funciona en función de la tasa de cambio del error y se corrige antes de que aumente el error.

2.4.3 Acción de control Integrativa

La acción de control integrativa proporciona una salida del controlador proporcional al error acumulativo convertido en tiempo de respuesta. También se encarga de notificar a la salida que tan rápido hay que moverse cuando se produce un error.

2.5 OpenCV

OpenCV (Open Source Computer Vision) es una librería software open-source de visión artificial y machine learning. Esta provee una infraestructura para aplicaciones de visión artificial. La librería tiene más de 2500 algoritmos, que incluye algoritmos de machine learning y de visión artificial para usar. Estos algoritmos permiten identificar objetos, caras, clasificar acciones humanas en vídeo, hacer tracking de movimientos de objetos, extraer modelos 3D, encontrar imágenes similares,etc.También la podemos encontrar en el uso para aplicaciones como la detección de intrusos en vídeos, monitorización de equipamientos, ayuda a navegación de robots, inspeccionar etiquetas en productos,etc. OpenCV está escrito en C++, tiene interfaces en C++, C, Python, Java y MATLAB interfaces y funciona en Windows, Linux, Android y Mac OS.

2.6 TensorFlow

TensorFlow es una biblioteca de software de código abierto para computación numérica, que utiliza gráficos de flujo de datos. Los nodos en las gráficas representan operaciones matemáticas y los bordes de las gráficas representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos. Esto también es una gran plataforma para construir y entrenar redes neuronales, que permiten detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos. La arquitectura flexible de TensorFlow permite implementar el cálculo a una o más CPU o GPU en equipos de escritorio, servidores o dispositivos móviles con una sola API.

2.7 ROS

Robot Operating System (ROS) es un middleware robótico, es decir, una recopilación o un entorno de frameworks para el desarrollo de software de robots que provee servicios estándar para el control e interacción de dispositivos de bajo nivel, implementando funcionalidad en cuestión a abstracción de hardware, el paso de mensajes entre procesos y el mantenimiento de paquetes. Está basado en una arquitectura de nodos, donde en el procesamiento se puede recibir, mandar y multiplexar mensajes de sensores, implementación de controladores, planificaciones, actuadores, entre otras cosas. Este software está principalmente orientado para el uso en el sistema UNIX(Ubuntu-Linux) aunque también en otros sistemas operativos donde son experimentales. La estructura de ROS se basa en un nodo principal de coordinación, publisher y subscribers que manejan el flujo de datos, multiplexión de la información, creación y destrucción de nodos, nodos en multiprocesamiento, parámetros de servidor y testeо en simuladores para un mejor desempeño en pruebas físicas.

2.7.1 Tópicos

Los tópicos se entienden como una forma de comunicación publisher/suscriber diseñada como transmisión asíncrona de mensajes, donde un elemento publica mensajes y el otro los recibe o se suscribe. Para interactuar con los tópicos en ROS se utiliza Rostopic la cual es una herramienta y esta tiene funciones similares

a las de «`rosnode`» y otras que son propias de la naturaleza de los tópicos. Una de las funciones más comunes es «`rostopic list`» que nos muestra la lista de los tópicos abiertos.

2.7.2 Nodos

Un nodo es un archivo ejecutable dentro de un paquete ROS, utilizan una librería cliente para comunicarse con otros nodos, en estos mismos pueden publicar o suscribirse a un tópico.

2.7.3 Acciones

Las acciones se catalogan como las transmisiones que van a pasar los mensajes con objetivos de lo que se quiere lograr. Las acciones son utilizadas para controlar el tráfico de mensajes donde se centra en realizar un solo objetivo sin ninguna alteración de alguna otra tarea siendo más robusto al momento de realizar algo en específico, logrando un mejor control que el manejo de múltiples tareas.

2.7.4 Launch File

Un launch file es un archivo de lanzamiento que proporciona una forma conveniente de iniciar múltiples nodos, servicios y un master, así como otros requisitos de inicialización que se requieran, como la configuración de parámetros. Esto se puede hacer especificando el paquete en el que se encuentran los archivos seguido del nombre del archivo y así con todo lo que se requiera para iniciar un entorno determinado. Este tipo de archivos tienen una extensión de `.launch` y usan un formato XML específico, estos se pueden colocar dentro de un directorio de paquetes para organizar los archivos de lanzamiento donde el contenido del archivo launch debe estar en un par de etiquetas según el formato de XML.

2.7.5 Servidor de parámetros

El servidor se utiliza para almacenar variables en un espacio para todos los nodos, estos nodos usan este servidor para almacenar y recuperar parámetros en tiempo de ejecución. Este

2.7.6 OpenCV en ROS

OpenCV es una biblioteca madura y estable para el procesamiento de imágenes 2D, que se utiliza en una amplia variedad de aplicaciones. Gran parte de ROS utiliza sensores 3D y datos de nubes de puntos, pero todavía hay muchas aplicaciones que utilizan cámaras 2D y procesamiento de imágenes. Muy utilizado en implementaciones con ROS para robots diferenciales, donde resaltamos la parte de procesamiento de imágenes, siendo los frases los utilizados para operar con Open CV, ya sea con Python o C++.

```
x=self.bridge.cv2_to_imgmsg()  
self.bridge=cv_bridge.CVBridge()
```

CvBridge es una biblioteca de ROS que proporciona una interfaz entre ROS y OpenCV. CvBridge se puede encontrar en el paquete cvbridge en la pila visionopencv.

2.7.7 TensorFlow en ROS

El TensorFlow en ROS se ocupa para realizar un paquete simple para realizar el reconocimiento de imágenes, este mismo paquete contiene un nodo de ROS que se suscribe a las imágenes del controlador de la cámara web de ROS y realiza el reconocimiento de imágenes mediante las API de TensorFlow. El nodo dara como salida el objeto detectado y su probabilidad.

2.8 Puzzlebot

El workspace proporcionado cuenta con todo lo necesario para cargar las funciones necesarias para hacer correr el puzzlebot, donde se tiene ya la implementación de los nodos de velocidad lineal y angular, así como también la integración de la cámara para realizar simulaciones el preprocesamiento de la imagen siendo para captar semáforos y el seguidor de línea negra, donde se tienen en consideración los siguientes paquetes:

- Puzzlebot Msgs
- Puzzlebot Navigation
- Puzzlebot Run
- Puzzlebot Vision

2.8.1 Características del Hardware

Microcontrolador ESP32:

- Microprocesador (CPU) LX6 de 32 bits de doble núcleo (o núcleo único) de Xtensa, que funciona a 160 o 240 MHz y tiene un rendimiento de hasta 600 DMIPS
- Wifi 802.11b/g/n HT40, banda base.
- Bluetooth: v4.2 BR / EDR y BLE
- Memoria de 520 KB SRAM
- Flash 4 MB incluido en el módulo WROOM32
- On-board PCB antenna
- Controlador de host SD / SDIO / CE-ATA / MMC / eMMC
- Controlador esclavo SDIO / SPI
- Coprocesador de ultra baja potencia(ULP)
- Sensor de efecto Hall
- 10 × sensores táctiles (GPIO de detección capacitiva)

- 32 kHz crystal oscillator
- 2 × interfaces I²S
- 2 × interfaces I²C
- 3 x UART
- 3 x SPI
- 12 x ADC input channels
- 2 x DAC
- Mando a distancia por infrarrojos (TX / RX, hasta 8 canales)
- Ethernet interfaz MAC con DMA dedicado y IEEE 1588 Precision Time Protocol apoyo
- OpenOCD debug interface with 32 kB TRAX buffer
- SDIO master/slave 50 MHz
- OTP de 1024 bits, hasta 768 bits para clientes
- Todas las funciones de seguridad estándar IEEE 802.11 son compatibles, incluidas WFA, WPA / WPA2 y WAP
- Aceleración de hardware criptográfico: AES , SHA-2 , RSA , criptografía de curva elíptica (ECC), generador de números aleatorios (RNG).

Protocolos de comunicación soportados:

- I2C
- I2S
- SPI
- UART

Raspberry Pi Camera V2:

- Este se conecta para dispositivos con un cable de cinta de 15 pines, para la interfaz serial de 15 pines de cámara MIPI (CSI-2).
- Para ocupar la cámara existe una integración con OpenCV y de Tensor-FlowLite desde la librería libcamera.

- Esta librería (libcamera) soporta los formatos JPEG, JPEG, PNG, YUV420 y RGB888 para imágenes; para videos: raw h.264 (acelerado) y MJPEG.
- Resolución de imágenes de 8 Megapíxeles, 3280 x 2464
- Resolución de Video 1080p30, 720p60, y 640x480p90
- Peso (Cámara + cable): 3.4g
- 32 bytes de memoria programable una sola vez incorporada (OTP)

NVIDIA Jetson Nano:

- Su conectividad es en Gigabit Ethernet, M.2 Key E
- Su voltaje recomendado es desde 4.75 y 5.25V.
- Su corriente debe ser mayor a 3A.
- Voltaje en los pines con un aproximado de: ± 0.5 V.
- Puertos: GPIO, I2C, I2S, SPI, UART

Especificaciones de procesador y GPU:

- 48KB L1 instruction cache (I-cache) por núcleo.
- 32KB L1 data cache (D-cache) por núcleo.
- 4GB de memoria y 16GB de almacenamiento.
- 2D BLIT.
- Arquitectura ARMv8 de 4 núcleos.
- Compresión de color en 2D.
- Color render constante SM bypass
- Salidas de vídeo: DMI 2.0 o DP 1.2 — eDP 1.4 — DSI (1 x 2) 2 simultáneos
- 2x, 4x, 8x MSAA con compresión a color.
- Procesamiento máximo de imagen 1400Mpix/s (hasta 24MP sensor)
- Conexión cámara de 12 vías (3 x 4 o 4 x 2) MIPI CSI-2 DPHY 1.1 (18 Gbps)
- MIPI CSI 2.0 hasta 1.5Gbps (por carril)

2.8.2 Características del Software

En las principales características para el software se utilizó TensorFlow, ROS, Phyton, OpenCV para así crear una red neuronal, con esto se fue creando una capa de neuronas las cuáles recibían inputs o una serie de valores y posteriormente esto daba como objetivo montar una capa de una red neuronal en el cuál solo necesitábamos saber el número de neuronas en la capa y el número de neuronas de la capa anterior para así crear nuestra red, y fue como poco a poco se fue estructurando la capa.

2.9 Deep Learning

DeepLearning es una rama de Machine Learning, que se basa en entrenamientos para que realice tareas como las que hace el ser humano, haciendo reconocimiento del habla, identificación de imágenes o hacer predicciones, de tal forma organizar los datos a través de ecuaciones predefinidas y así proponer un modelo para que aprenda por su cuenta propia reconociendo patrones mediante el uso de capas que permitan filtrar características pertinentes. Es una de las bases de la Inteligencia Artificial, donde en conjunto va mejorando la capacidad de clasificar, reconocer, detectar y describir conjuntos de datos. Se necesita mucho poder computacional para resolver la complejidad de Deep Learning debido a la naturaleza iterativa de los métodos de algoritmos para el aprendizaje de la información, donde la complejidad aumenta por el número de capas y por la gran cantidad de volúmen de datos que se necesita para entrenar una red neuronal. En relación al puzzlebot, se necesitará la aplicación de una red neuronal para procesar señales de tráfico y clasificar estas señales durante el trayecto del puzzlebot. (SAS 2021)

3 Descripción del Reto

El reto consiste en el armado de un carrito autónomo seguidor de línea y señales de tránsito, formado por un kit Puzzlebot otorgado por Manchester Robotics, en colaboración con la Universidad de Manchester 1824 y NVIDIA, de tal forma emplear herramientas de Computer Visión, DeepLearning, progra-

mación en Python, empleando la arquitectura de ROS para estructurar nodos y tópicos, haciendo más fácil la ejecución por partes, simulando una navegación autónoma de vehículos, siendo un robot móvil a pequeña escala, así como el diseño y simulación de algoritmos de procesamiento, de aprendizaje profundo para finalmente probar en una escala real y en el puzzlebot.



Figure 1: Socioformadores

3.1 Características del Reto

Se proporcionó un kit para armar el carrito de Manchester Robotics, donde se tiene en cuenta la siguiente lista de materiales para lograr el armado del puzzlebot, siendo lo siguiente:

1. NVIDIA Jetson Edition por Manchester Robotics 2GB
2. HACKER BOARD para motores del puzzlebot
3. Raspberry PI Camera V2
4. Motores con Encoders
5. LCD con I²C
6. Módulo Wifi/Bluetooth
7. Fuente de poder con salida de 3A y 5V.
8. KIT de desarrollo para el armado del Puzzlebot

Para el buen proceso del armado del puzzlebot se toma en cuenta los siguientes puntos para conocer más del entorno de trabajo tanto de hardware y software:



Figure 2: Materiales pertinentes

1. Armado de robot y puesta en marcha.
2. Entorno de ROS: Teleoperación.
3. Control con ROS: Navegación autónoma.
4. OpenCV: Detección de color.
5. OpenCV: Seguimiento de carriles.
6. Tensorflow: Shape recognition.
7. Pruebas en entorno real

3.2 Trayectoria a Resolver

El reto consiste en conducir de forma autónoma el PuzzleBot en una pista predefinida. La pista consta de diferentes piezas de MDF ensambladas juntas para formar una forma de "b". El PuzzleBot debe seguir un pre-ruta definida obedeciendo los semáforos y de la pista. Se tiene pensado con la implementación de semáforos, señales de tráfico, donde el puzzlebot tiene que seguir una línea del carril y con un punto de intersección para definir que trayectoria, de tal forma que avanza el puzzlebot en línea recta llegando a un punto de intersección, donde encuentra la señal del semáforo y la indicación ya sea RED o GREEN prosigue a entrar en la parte del cuadrado donde tenemos 3 curvas hasta volverse a encontrar otra vez con la misma intersección y otro semáforo para indicar el giro final a la

derecha y llegar al mismo punto inicial y consecutivamente detenerse viendo la señal de tráfico "Stop".

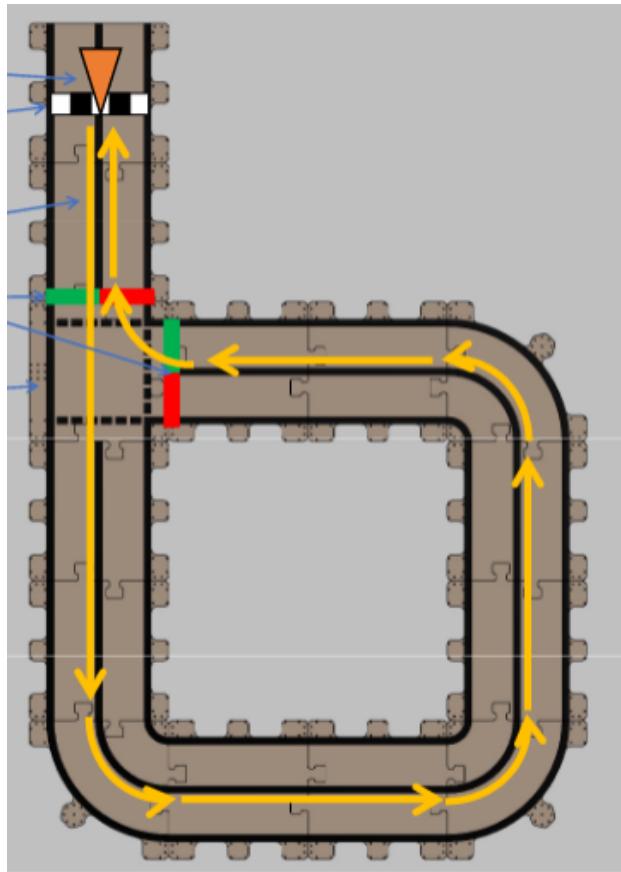


Figure 3: Trayectoria Puzzlebot

3.3 Condiciones de Manejo

Las condiciones de manejo implica que el puzzlebot en ningún momento se tiene que salir de carril, así como también mantener el seguimiento de línea, respetar las luces de los semáforos, respetar los señalamientos de stop, right signal, acelera un poco más, up signal, ahead only, y el punto de intersección para avanzar y terminar el trayecto. Un semáforo es ubicado en la parte de enmedio y otro de manera que el puzzlebot con la cámara lo detecte en la orilla del camino, ayudándose del punto de intersección para realizar los cambios correspondientes.

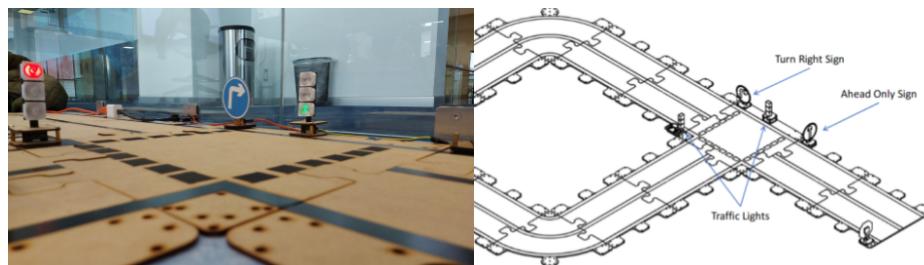


Figure 4: Semáforos pista

3.3.1 Semáforo

En la parte de semáforos se toma en cuenta uno en la parte del inicio para avanzar a la parte de la curva y otro al llegar al punto de intersección una vez completado el trayecto, donde el puzzlebot debe de ser capaz de detenerse en el cambio de luces.

3.3.2 Señales de Tráfico

Las señales de tráfico que se consideran en la pista son los siguientes:



Figure 5: Señales de tránsito

3.3.3 Seguimiento de carril

El seguimiento del carril se hace conforme a la detección de línea negra de la pista, donde el robot se maneja de manera autónoma y debe de seguir el trayecto marcado por la línea negra de la pista.

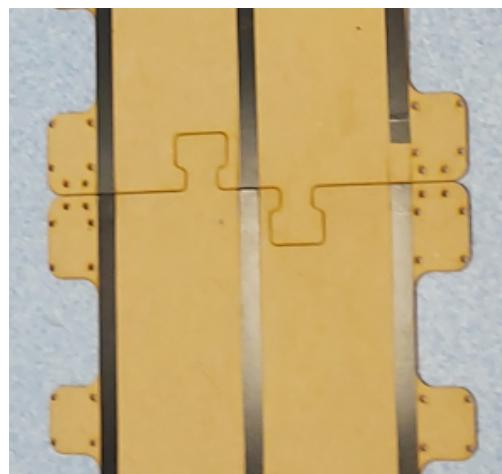


Figure 6: Segmentación Pista

3.3.4 Intersecciones

En la parte de intersecciones se toman en consideración líneas en cuadros negros con espacios determinados para dar a entender que es una intersección, siendo de gran ayuda para determinar otro tipo de trayectoria con el puzzlebot, donde la cámara se encarga de determinar dicho espacio con la detección de contornos.

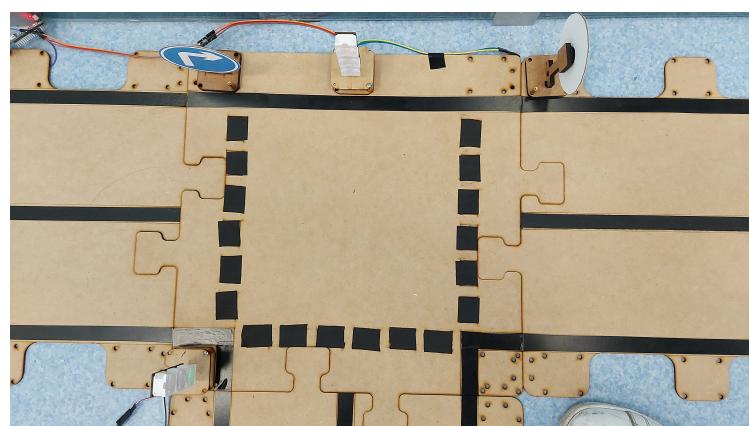


Figure 7: Intersección Pista

4 Odometria:

En robots diferenciales, es común usar la odometría de un robot para llevar a cabo el control posicional de un robot. Se tiene que hacer unos ajustes para obtener una estimación de la posición del robot mediante la información desplegada en los tópicos del mismo.

1. Coordenadas (x,y) posición en 2 ejes: El punto en donde esta el robot.
2. Ángulo de orientación: Hacia donde se desea que el robot termine viendo al finalizar la trayectoria.

El robot incorpora 2 tipos de movimiento, el control seguidor de línea y el control posicional por coordenadas objetivo. El segundo se vale de la odometría para estimar la posición (pares de coordenadas) y orientación.

Las ecuaciones de odometria son las siguientes:

$$x(t_{i+1}) = x(t_i) + R \frac{\omega_R + \omega_L}{2} \cdot \cos(\theta(t_i)) \cdot dt \quad (1)$$

$$y(t_{i+1}) = y(t_i) + R \frac{\omega_R + \omega_L}{2} \cdot \sin(\theta(t_i)) \cdot dt \quad (2)$$

$$\theta(t_{i+1}) = \theta(t_i) + R \frac{\omega_R - \omega_L}{B} \cdot dt \quad (3)$$

Estas ecuaciones estiman la posicion y orientacion del robot usando las velocidades angulares de cada una de las llantas (estas son provistas por la plataforma puzzlebot), e informacion de la geomtria del robot

4.1 Convección atan2 de orientación

: Para el control de orientación del robot, se hizo uso de esta convención de ángulos.TODO: EXPLICAR MAS A DETALLE.

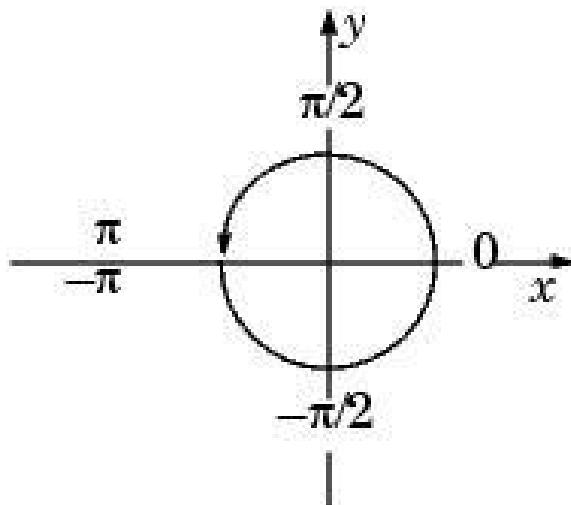


Figure 8: Convección atan2

5 Control Posicional:

El algoritmo requiere de 2 argumentos:

1. Coordenadas (x,y) objetivo: El punto hacia donde se desea desplazar el robot
2. Ángulo de orientación: Hacia donde se desea que el robot termine viendo al finalizar la trayectoria.

Dados los datos de entrada el control posicional se divide en tres etapas:

1. Control de Dirección: El robot gira hasta llegar a una orientación donde puede desplazarse directamente hacia las coordenadas objetivo.
2. Control Posicional: El robot se desplaza a una velocidad lineal constante

y una velocidad angular modulada mediante control PD hacia la posición objetivo

3. Control de orientación: Una vez que el robot llega a las coordenadas objetivos, se orienta con respecto al tercer input (ángulo deseado) para terminar viendo en una dirección específica.

Detalles de la Implementación:

6 Implementación del sistema de odometría y manejo punto a punto,

Este sistema permite llevar al robot a una posición y orientación específica. La posición y angulo iniciales se guardan en memoria en el lugar donde esta el robot al momento en el que se corren los nodos correspondientes, y se basa el algoritmo de control posicional (Ver Marco Teórico).

De esta manera se utiliza la posición al inicio de la acción, para cruzar una intersección, y se calcula el punto deseado y su orientación para salir de la intersección y volver al nodo de seguimiento de línea.

7 Explicación del Funcionamiento:

El funcionamiento del sistema de navegación se puede explicar mediante la siguiente máquina de estados:

Para el primer estado, es necesario orientar el robot con respecto a las coordenadas objetivo. Es decir, se tiene que orientar hacia donde tiene que ir. Esto se hace girando con respecto a una referencia que se obtiene mediante la función atan2 de la librería math de python. Dicha función recibe como argumento un par de coordenadas (X,Y) y regresa el angulo en el que el robot debe tener que orientar para ir a dicha posición. Las posiciones de los argumentos son la salida de las ecuaciones (1) y (2). Asimismo, el angulo de referencia para comparar con la orientación objetivo está dado por la ecuación (3).

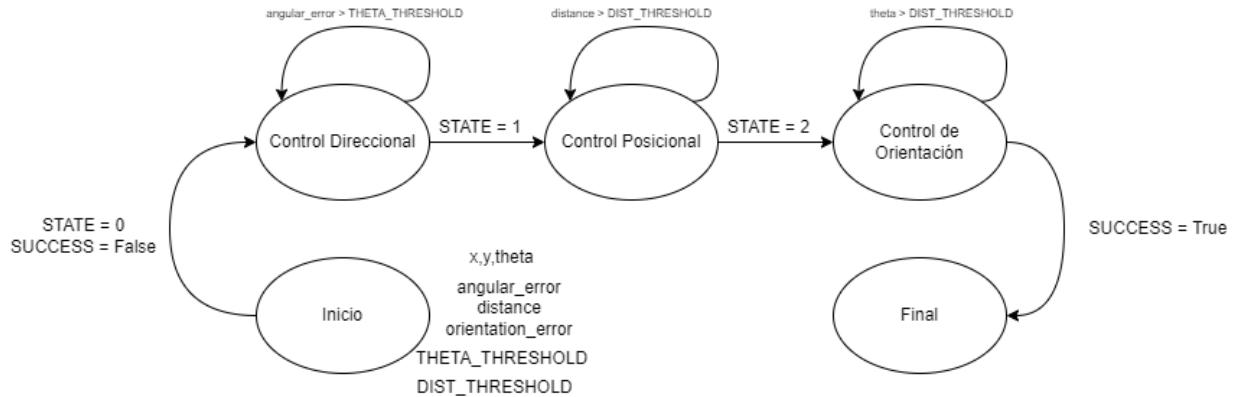


Figure 9: Máquina de estados Nodo de Navegación ??

Lo que se hace es definir un umbral de error en orientación ($\pi * 10/180$) y hacer que el robot gire hasta que el error sea menor a dicho umbral. El error se computa mediante el valor absoluto de la diferencia del ángulo de orientación actual del robot y el ángulo objetivo provisto por la función atan2.

Una vez que el robot esta orientado, se avanza con una velocidad lineal constante hasta la posición deseada. Notese que se sigue computando el error angular como en el primer estado. Esto se debe a que este error sirve como la entrada de un controlador proporcional derivativo para modular el giro del robot (ver Implementación en ROS).

Nuevamente, es necesario computar un error para determinar cuándo se tiene que parar el robot. Esto se hace calculando la distancia euclídea entre la posición actual (resultado de las ecuaciones (1) y (2)) con la posición objetivo (otro par de coordenadas adicionales que se publican a una acción de ROS). Cuando la distancia euclídea es menor a un umbral de distancia, se pasa al siguiente estado.

El tercer estado es un control de orientación similar al del primer estado. La diferencia es que se hace con respecto al ángulo de orientación objetivo que se recibe como tercer argumento del algoritmo.

7.1 Implementación en ROS:

Este sistema consta de los siguientes componentes de ROS:

7.1.1 Nodo de Odometria:

Este nodo se encarga de suscribirse a los tópicos de velocidad angular de las llantas del robot ($/wr$ y $/wl$), computar la orientación y la posición actual del robot (usando las ecuaciones (1), (2), y (3)), y publicarlas a un tópico para ser usadas por el nodo de navegación.

Para el parámetro de θ se considera un pequeño detalle de conversión de ángulos con el fin de que los angulos esten en la convección atan2. La conversión del circulo unitario clásico a la convención atan2 se hace de la siguiente manera:

- Caso 1

$$position\theta > \pi$$

Se compensa con:

$$position\theta = position\theta - 2 * \pi$$

- Caso 2

$$position\theta < -\pi$$

Se compensa con:

$$position\theta = position\theta + 2 * \pi$$

7.1.2 Nodo de Navegación

Para el diferencial de tiempo, se inicializan dos variables en cero donde una guarda el tiempo presente y la otra el tiempo de la iteracion pasada. El diferencial de tiempo esta dado por la diferencia entre el tiempo presente y pasado.

```
last_time = current_time  
current_time = rospy.get_time()  
dt = current_time - last_time
```

La implementación de las ecuaciones de Odometria es la siguiente:

```
position.x = x+radius*((self.wr+self.wl)/(2.0)) * cos(theta)*dt  
position.y = y+radius *((self.wr+self.wl)/(2.0)) * sin(theta)*dt  
position.theta = theta+radius*((self.wr-self.wl)/(base))*dt
```

Considerando todo esto, se hace el correcto envío y monitoreo de los datos en el script de odometría. En la siguiente imagen se considera el punto inicial de reposo donde relativamente "x" y "y" tienen valores cercanos a 0 por considerar una coordenada en (0,0).

7.1.3 Accion para asignar objetivo

Una vez que el sistema de odometria funciona y puede ser personalizado usando el archivo de configuración adecuado, se crea un nuevo script encargado de la navegación del robot con base de la implementación de un servidor para Acciones de ROS. Este script funcionará a partir de la implementación de Acciones de ROS, lo cual permitirá que el robot se traslade a un punto objetivo mientras se monitorea el estado del robot durante esta trayectoria, así como el resultado de la acción implementada. La acción a implementar estará definida por la siguiente estructura:

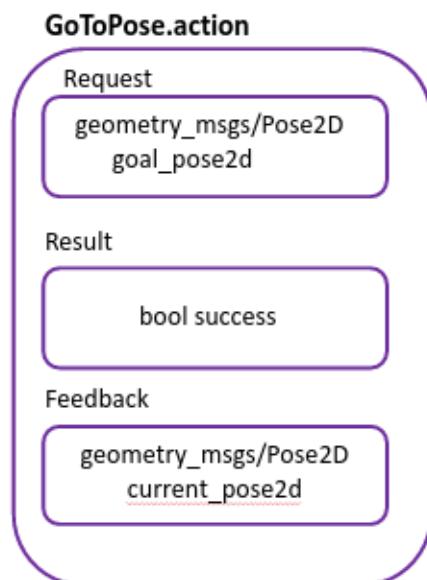


Figure 10: Acción GoToPose

Para modificar la acción se requiere ir al paquete de PuzzlebotMsgs donde se tiene que modificar el archivo GoToPose.action para definir adecuadamente la acción de tal forma que también se tiene que actualizar el contenido de los archivos de configuración CMakeLists.txt y package.xml para poder compilar la acción que se va a definir y así poder generar los mensajes adecuados, donde se utiliza el comando de catkin make para realizar el procedimiento de compilación.

7.1.4 Controlador

En la implementación del controlador para el seguimiento de línea, se implementó un script llamado puzzlebot edges de python en base al análisis de la línea de detección donde ya tenemos la información de leftEdge y rightEdge con todo el desglose del primer gradiente, segundo gradiente y el shif que se hace para comparar y obtener el data de cada uno de los lados. Esta información se utiliza para crear dos suscribers y manejar la información y crear un nuevo publisher del error que va a corregir en el nodo de navegación. De tal forma quedando de la siguiente forma:

```
# Subscribers
self.leftEdgeSubscriber = rospy.Subscriber("/leftEdge", ...
Float32MultiArray, self.leftEdgeCallback)
self.rightEdgeSubscriber = rospy.Subscriber("/rightEdge", ...
Float32MultiArray, self.rightEdgeCallback)

# Publishers
elf.angularErrorPub = rospy.Publisher("/angularError", ...
Float32, queue_size = 10)
```

Ya una vez definido con los datos que se van a trabajar se toma en cuenta un diccionario de python para mostrar la información de izquierda, derecho y el centro que este captando la cámara, de tal forma crear un nodo y así recibir el line edges y el line position para los 3 casos.

```
def edgeModulation(self):
    line_edges_tuple = []
```

```

filtered_line_edges_tuple = {
    "left" : {
        "line_edges" : None,
        "line_position" : None
    }, "center" : {
        "line_edges" : None,
        "line_position" : None
    }, "right" : {
        "line_edges" : None,
        "line_position" : None
    }
}

```

Esto toma base del análisis de los edges donde ya con la información que se tiene de las matrices left and right se hace un publisher para generar el angular error, esto procesado por medio de tuplas, donde a grandes rasgos se recorren los arrays de los edges para determinar valores pertinentes, y estos valores agregarlos a nueva tupla, utilizando la función lambda para después ponerlos en orden de la función, quitando la información innecesaria y tener un nuevo conjunto de datos más filtrados. Y así tener una nueva tupla de izquierdo y derecho donde a partir de ello se genera el error angular, teniendo lo siguiente:

```

if filtered_line_edges_tuple["center"]["line_position"]...
== None:
    self.currentLinePos == self.previousLinePos
    self.angularError = np.nan
else:
    self.currentLinePos = filtered_line_edges_tuple["center"]...
    ["line_position"]
    self.angularError = filtered_line_edges_tuple["center"]...
    ["line_position"]

```

En esta sección si no se hace la detección de línea no se corrige, en caso contrario si se hace la detección del Angular Error deja de enviar NAN de tal forma enviar un valor de corrección donde este va a ser traspasado en el nodo de navegación, para poder corregir el ángulo de giro en base al line detection y al

análisis de los edges.

Una vez calculado el Angular error en el script de los Edges se hace un suscriber en el nodo de navegación para pasar el valor de la corrección del Angular error, teniendo la siguiente sentencia:

```
self.angularSub = rospy.Subscriber("/angularError", ...  
Float32, self.angularErrorCallback)
```

Finalmente este valor irá en la corrección de la velocidad angular donde ayudará a mejorar el trayecto, a la corrección de la línea y los edges, el nodo de Navegación utilizará un suscriber y a través de una función de callback se hará el uso del dato del Angular Error para disponerlo en la implementación del controlador

7.1.5 Controlador PD

Finalmente, el error calculado pasa al nodo de navegación que se subscribe al tópico "/angularError" quedando de la siguiente forma:

```
self.angularSub = rospy.Subscriber("/angularError", Float32,
```

Para esta implementación la velocidad angular es constante, y la velocidad se ajusta mediante un controlador PD, tomando en cuenta las siguientes consideraciones:

```
def angularErrorCallback(self, msg):  
    angularError = msg.data  
    angularErrorAbs = abs(angularError) if...  
    not math.isnan(angularError) else self.pastAngularErrorAbs  
    controlAngularSpeed = kp * angularErrorAbs + ...  
    (kd * (angularErrorAbs - self.pastAngularErrorAbs))...  
    / angularTempDiff
```

De tal forma que apartir de la función angularErrorCallback se implementa en

el controlador, utilizando la variable de Angular error igualado como msg.data para utilizarlo en el control de la velocidad Angular. Para el control de Izquierda a Derecha se considera un THRESLHOLD de tal forma que las vueltas se logren concretar y le de tiempo al robot de seguir con su trayecto, donde se tiene lo siguiente:

```
if not math.isnan(angularError):
    self.counterNAN = 0
    if self.counter >= COUNTER_THRESHOLD and...
        self.counterNAN <= COUNTER_THRESHOLD:
            self.pastAngularErrorAbs = angularErrorAbs
            self.pastAngularError = angularError
    else:
```

En la anterior sentencia se tiene en consideración el giro del puzzlebot de la pista donde eventualmente puede que mientras gira encuentra otra línea, cuando se encuentran NAN se mantiene unos momentos girando, de tal forma que se tiene un COUNTER THRESHOLD para mantener haciendo la acción hasta volver al camino del giro y pueda volver a encontrar la línea, de tal manera pueda volver a encontrar seguidamente los valores pertinentes, solo el robot parará cuando ya encuentre seguidamente puros NAN en un tiempo determinado.

La ecuación del controlador depende del error absoluto, donde se considera la distancia que se está de la recta y la distancia respecto al momento anterior. El valor absoluto presenta un problema para corregir, donde si se considera este valor como el error se corregiría para un solo lado, por tanto se declara un factor para que en la fórmula del controlador este pueda corregir para ambos lados, siendo lo siguiente:

```
if self.pastAngularError > 0:
    factor = -1.0
elif self.pastAngularError < 0:
    factor = 1.0
```

Este factor pregunta por el error angular si es mayor o menor a 0 y dependiendo

del valor del pastAngularError será la corrección del puzzlebot, para finalmente corregir el Angular Speed, mientras que la velocidad lineal constante.

```
cmd_vel.angular.z = factor * controlAngularSpeed if...  
controlAngularSpeed <= 0.1 and controlAngularSpeed...  
>= -0.1 else 0.1 * factor  
cmd_vel.linear.x = 0.1
```

1.-¿Cómo implementaste tu sistema de control de manejo autónomo con base en las posiciones obtenidas?

Se hizo en base a calcular el error Angular y con el sistema de odometría que toma en cuenta la geometría del robot, distancia entre ruedas y los movimientos que se ejecutan con las diferencias del tiempo actual y previo. Esto ayuda a generar un sistema basado en la corrección de la velocidad angular y proponer una velocidad lineal en donde se permita hacer la corrección del trayecto.

2.-Ajuste del sistema de control de velocidad lineal y angular, elección de constantes de control (K_p , K_d) Las constantes se hizo en base a la explicación anterior, donde a través de un suscriber escrito en el sistema de navegación al nodo de edges se hace la corrección en base al edge de la línea que detecta la cámara del robot. De tal forma que las ganancias nos dice el error de la distancia que está de la recta y la distancia respecto al momento anterior respecto a nuestra ganancia derivativa. Por tanto sucede el ajuste autónomo a medida que avanza el puzzlebot.

3.- Problemáticas Implementación de propuesta final del controlador autónomo

- Un gran problema fue la parte de las vueltas de la pista y que volvería a tener un trayecto adecuado.
- La corrección del valor del Error absoluto en un solo sentido.
- La implementación de THRESHOLD del tiempo y distancia para llegar a concretar la vuelta y volver a la línea para seguir con el trayecto de la línea negra.
- Conocimiento Empírico para ir haciendo pruebas.

Al final de abordar todas las problemáticas todo resultó para mejorar el proceso de navegación.

7.1.6 Archivo de Configuración

Para la configuración del servidor de parámetros de ROS se utiliza el archivo de configuración puzzlebotNavigation.ymal ubicado en el paquete de navegación en la carpeta de config, donde se utiliza la sintaxis adecuada para archivos YAML, en este archivo se define parámetros como el radio, la base entre las llantas del robot y el rate de las lecturas de odometría. YAML es un lenguaje de serialización de datos que suele utilizarse en el diseño de archivos de configuración. Al crear un archivo YAML, es necesario asegurarse de que sea válido y siga las reglas sintácticas. Uno de los usos más comunes es la creación de archivos de configuración. Se recomienda utilizar YAML en lugar de JSON para escribir los archivos de configuración porque es un lenguaje más fácil de comprender, aunque ambos pueden usarse de manera indistinta en la mayoría de los casos. (RedHat 2021)

¿Cómo ajustar el comportamiento de mis nodos haciendo uso del sistema de parámetros en ROS? El comportamiento de los nodos puede ser ajustado tomando parámetros pertinentes y definiéndolos en el archivo de configuración YAML, donde podemos pasar variables como el RATE de la odometría, las ganancias, la base de las llantas, el radio de las llantas, los THRESHOLD, entre otras variables, un ejemplo de sintaxis puede ser el siguiente considerando solamente algunos parámetros:

```
puzzlebot_navigation:  
  parameters:  
    base: 0.191  
    radius: 0.05  
    odometry_pub_rate: 50
```

8 Notas generales sobre la Obtención de Imágenes en Sistemas de Visión Computacional:

Para el correcto funcionamiento del robot, fue necesario hacer uso de métodos de visión computacional para dos tareas:

- Detección de Luces de Tráfico:
- Detección de Señales de Tráfico:
- Detección de línea central de la carretera.

8.1 Obtención y publicación de imágenes:

En estos algoritmos es común tener que transformar mensajes de imagen de ROS a formatos que pueden ser procesados por OpenCV. De la misma manera fue necesario publicar imágenes con propósitos de Debuggeo.

Nótese que todos los nodos que llevan procesamiento de imágenes, detección de características, o tareas de visión computacional en general ocupan exactamente la misma metodología. En vista de que solo cambian los nombres de las variables y los tópicos a los que se publican, el proceso de obtención de y publicación de imágenes únicamente se explica en esta sección. A lo largo del documento los autores se referirán a esta sección para la obtención de imágenes se fuese necesario.

8.1.1 Obtención de Imágenes:

El proceso de obtención de imágenes se realiza mediante el objeto Bridge de ROS y un subscriber de tipo Image. Es necesario instalar la dependencia correspondiente:

```
sensor_msgs
```

El código para recibir mensajes de ROS es el siguiente:

```

...
self.bridge = cv_bridge.CvBridge()
self.image = None
self.rawVideoSubscriber = rospy.Subscriber(camera_topic
    ,Image,self.imageCallback)
...
def imageCallback(self,msg):
    self.image = self.bridge.imgmsg_to_cv2(msg
        ,desired_encoding="bgr8")

```

Se requiere instanciar un objeto Bridge, un placeholder para la imagen que será recibida (es importante guardar la imagen porque siempre se requiere preprocesar), un subscriptor a un tópico que publique mensajes de tipo Image (lógicamente, el suscriptor debe de ser del mismo tipo de dato) y una función callback.

El tópico estará publicando imágenes constantemente. Dichas imágenes en formato de mensaje de ROS serán recibidas por el tópico que llamará a la función callback cuando lleguen nuevos mensajes. La función usa el objeto Bridge para convertir la imagen a un formato que pueda entender OpenCV.

8.2 Publicación de Imágenes:

Para publicar imágenes, se requiere de una estructura similar a aquella para la obtención de imágenes:

- Tópico para publicar imágenes: Se recomienda cargarlo desde un archivo de configuración.
- Publicador: Debe ser del tipo Image.
- Bridge: Será utilizado para convertir imágenes de OpenCv (arreglos de numpy) a mensajes tipo Image de ROS.

```

# Nombre de Tópico
OUTPUT_IMAGE_TOPIC =

```

```

"/puzzlebot_vision/line_detection/edges_detection_image"
...
# Objeto CvBridge
self.bridge = cv_bridge.CvBridge()

# Publicador
self.edgesImagePub =
    rospy.Publisher(OUTPUT_IMAGE_TOPIC, Image, queue_size=10)
...
# Convertir OpenCv -> ROS
output = self.bridge.cv2_to_imgmsg(binarized)
#Publicar Tópico
self.edgesImagePub.publish(output)

```

8.2.1 Sobre la publicación de imágenes y el número de canales a publicar:

Constantemente, se encontraron problemas al momento de publicar una imagen con una codificación deseada como se muestra a continuación:

```
output = self.bridge.cv2_to_imgmsg(binarized, encoding="bgr8")
```

Nos dimos cuenta que esto se debia a que siempre se usaba la codificación bgr8 la cual asume que se publica una imagen de 3 canales de color. El error se da cuando se intenta codificar una imagen en escala de grises con un formato hecho para una imagen con tres canales de color. Como en nuestra implementación se publican ambos tipos de imágenes (ya sea para propósitos inherentes al funcionamiento o para debuggear). Entonces, alternamos entre publicar las imágenes con formato o sin formato según los canales de color:

```
output = self.bridge.cv2_to_imgmsg(binarized)
```

9 Detección de semáforos:

Este nodo se encarga de la detección de líneas de tráfico rojas o verdes captando las imágenes de la cámara. Donde a partir de la colocación de las esquinas de semáforos, van cambiando los colores, mientras el puzzlebot se desplaza por la pista.

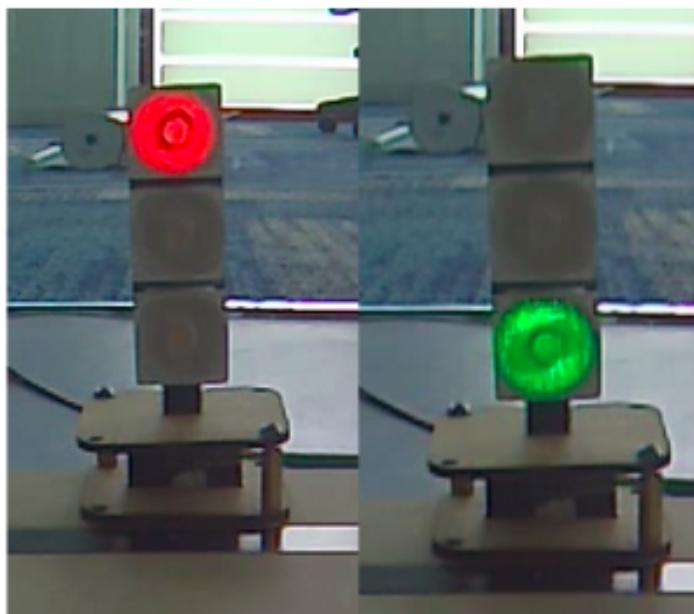


Figure 11: Semaforos Pista

Primeramente se hace un preprocessamiento de la imagen de la cámara, donde se hace un factor de reescalamiento a partir de height y width, haciendo un resize de la imagen, para después aplicar un Gaussian Blur para resaltar aún más los bordes y quitar un poco el ruido de la imagen en la búsqueda de detección de los semáforos.

```
height = int(float(img.shape[0]) * ...  
(float(self.img_scale_factor)/100.00))  
width = int(float(img.shape[1]) * ...  
(float(self.img_scale_factor)/100.00))  
size = (width, height)  
img = cv2.resize(img, size)  
#img = cv2.rotate(img, cv2.ROTATE_180)
```

```
img = cv2.GaussianBlur(img, (3, 3), 0)
```

Después de implementar una función para extraer pixeles de acuerdo a un canal de tal forma hacer un tipo GRAYSCALE de cada escala de colores de RGB, donde solo nos interesa el color rojo, amarillo y verde, a través de un threshold se asignan la búsqueda de pixeles blancos, resaltando en una imagen binarizada la búsqueda de colores por canal, de tal forma resaltar en blanco la forma circular, quedando de la siguiente manera:

```
img_b, img_green, img_red = cv2.split(img)
th, redPixels = cv2.threshold(img_red, 220, 255, 0)
th, greenPixels = cv2.threshold(img_green, 220, 255, 0)
```

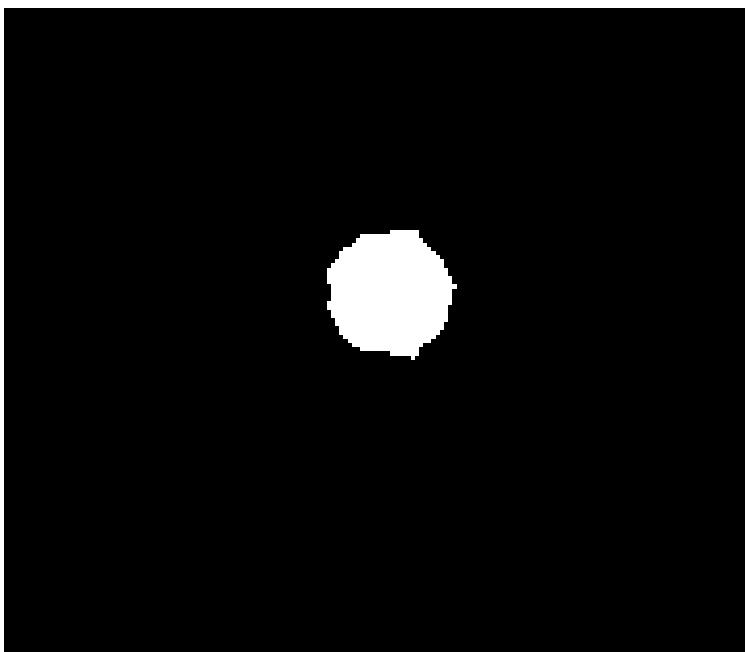


Figure 12: Imagen Binarizada por canal

Una vez esto se aplica la búsqueda de contorno, donde estos se dibujarán sobre la imagen original y sobre la máscara de threshold para obtener una lista de contornos detectados en el semáforo, con la función cv.findContours que no ayudará a extraer en el frame actual cada contorno y cv.boundingRect a dibujar rectángulos en pequeñas secciones de la imagen de cada contorno obtenido, así como guardar toda esta información en un arreglo.

```
cnt, hierarchy = cv2.findContours...
```

```

(mask, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
rect = []
e = 5
imagec=img.copy()
for c in cnt:
    x, y, w, h = cv2.boundingRect(c)
    if(w>0) and (h>0):
        cv2.rectangle(imagec, (x, y), (x+w, y+h),...
(255,0, 0), 2)
        rect.append(img[y:y+h, x:x+w])

```

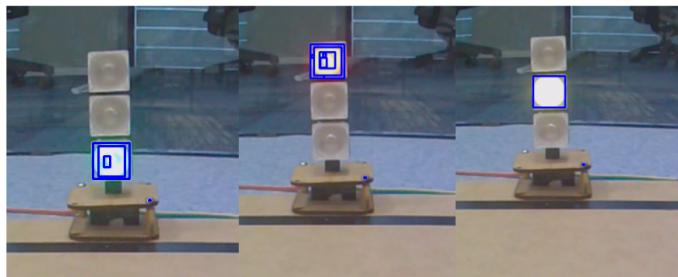


Figure 13: Rectángulos encontrados por canal

A partir de las imágenes segmentadas en rectángulos aplicar la detección de pixeles sobre esta lista de valores del nuevo arreglo con los rectángulos dibujados, donde se hace una función para recibir el canal de color de búsqueda y el arreglo de información. Se utiliza el formato HSV para detectar de mejor manera el canal de color teniendo diferentes THRESHOLD Lower and Upper haciendo mejor la detección de colores. Para el color rojo se requirió 2 partes de threshold por los bordes que se tienen del HSV donde se empieza del inicio del borde y al final del borde, donde se definió de la siguiente forma:

```

if(channel == 1):
    for c in rect:

        c = cv2.cvtColor(c, cv2.COLOR_BGR2HSV)
        # H, S, V
        red_min1 = np.array([0, 70, 110])

```

```

red_max1 = np.array([15, 255, 255])

red_min2 = np.array([170, 70, 110])
red_max2 = np.array([180, 255, 255])

mask1 = cv2.inRange(c, red_min1, red_max1)
mask2 = cv2.inRange(c, red_min2, red_max2)
mask = mask1 + mask2
counterRED+=np.count_nonzero(mask)
rospy.loginfo(counterRED)

```

Y para el verde solo se usó un apartado de THRESHOLD, donde de igual forma se recorre el arreglo con las imágenes segmentadas con rectángulos dibujados y para después pasar a la búsqueda de pixeles verdes, definiéndose de la siguiente manera:

```

if(channel == 1):
    for c in rect:

        c = cv2.cvtColor(c, cv2.COLOR_BGR2HSV)
        # H, S, V
        red_min1 = np.array([0, 70, 110])
        red_max1 = np.array([15, 255, 255])

        red_min2 = np.array([170, 70, 110])
        red_max2 = np.array([180, 255, 255])

        mask1 = cv2.inRange(c, red_min1, red_max1)
        mask2 = cv2.inRange(c, red_min2, red_max2)
        mask = mask1 + mask2
        counterRED+=np.count_nonzero(mask)
        rospy.loginfo(counterRED)

```

Finalmente lo que interesa de la sección de semáforos es enviar banderas de detección para usarlos en otros tópicos que utilicen las señales de semáforo permitiendo hacer un buen proceso de la máquina de estados.

```
self.found_red = True  
self.found_red = False  
self.found_green = True  
self.found_green = False
```

10 Implementación del sistema de detección y seguimiento del carril:

10.1 Obtención de las imágenes;

Para la obtención de las imágenes se crea un subscriber al tópico:

`"/video/source/image_raw"`

con la metodología explicada en la sección: Notas generales sobre la Obtención de Imágenes en Sistemas de Visión Computacional

10.2 Sistema de Detección de líneas de carril con Visión:

El algoritmo de detección de líneas consiste, a grandes rasgos, en localizar las posiciones de los centros de las líneas de la carretera que usará el robot para navegar por las calles. Decir, para una imagen de 200 pixeles, por ejemplo, el algoritmo deberá localizar el píxel donde se haya la línea central y publicarla como medición para que el sistema de navegación sitúe la línea central alrededor del pixel 100 ($error = 0$) mediante la corrección de la trayectoria del robot.

10.2.1 Preprocesamiento de las Imágenes:

El primer paso para la detección de líneas es la obtención de las imágenes (véase las secciones anteriores). Posteriormente, es necesario realizar el preprocesamiento pertinente. Dicho preprocesamiento se explica a continuación:

1. Convertir la imagen en escala de grises: Esto es importante para simplificar

la identificación de líneas. Buscamos cambios en los valores de la imagen y esto se aprecia mejor en un único canal de color.

```
gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

2. Escalar la imagen: Este paso es crucial, porque el enfoque propuesto conlleva muchas operaciones matriciales cuya ejecución tomará más tiempo en función del tamaño de la imagen. Al reescalar la imagen se reduce el tiempo de ejecución de estas operaciones y garantiza una respuesta más rápida por parte del robot. Para esta implementación de uso un factor de escalamiento del 40%.

```
width = int(gray.shape[0]*scale/100)
height = int(gray.shape[1]*scale/100)
gray = cv2.resize(gray,(height,width))
```

3. Rotar la imagen: Esto se hace porque se cambió la orientación de la cámara gracias al nuevo lente y a que en la posición anterior se captaron tonos de blanco que resultaban poco favorables para el algoritmo de detección de líneas.

```
gray = cv2.rotate(gray, cv2.ROTATE_180)
```

4. Aplicar Filtro Gaussiano: Este filtro tiene como propósito aumentar los ruidos de alta frecuencia caracterizados cambios abruptos en la escala de color (ej. 0 → 255). Esto amplifica los gradientes, y facilita tanto el filtrado de ruido como la detección de bordes.

```
gray = cv2.GaussianBlur(gray,(11,11),0)
```

5. Erosión y Dilatación: Esto se hace con el propósito de normalizar la información en la imagen. Es decir. El objetivo es eliminar píxeles que no guardan relación con sus vecinos y se deben a posibles efectos indeseados de luz o fallos de la cámara. Facilita el filtrado de ruido.

```
gray = cv2.erode(src=gray, kernel=(9,9) ,iterations=1)
gray = cv2.dilate(src=gray, kernel=(7,7) ,iterations=1)
```

10.2.2 Obtención de una Región de Interés:

Una vez que se tiene la imagen preprocesada, es importante tener en consideración que no toda la información captada es pertinente. Para esta tarea,

únicamente nos interesa la zona inferior de la imagen que es la porción del framque contiene el camino. Es por esto por lo que se hace un recorte de la imagen para obtener una región de interés consistente de la parte inferior de la imagen. Esto se hace con el sistema de indexación de numpy:

```
def sliceImage(self,img):  
    return img[int(self.imgHeight*0.60):,:]
```

Nótese que un arreglo de numpy es indexado de la siguiente manera:

```
numpyArray [beginHeight:endHeight,beginWidth:endWidth]
```

Donde el índice cero está en la esquina superior izquierda de la imagen. En este caso, se extraen todos los pixeles del ancho de la imagen y solo los pixeles del 60% en adelante para el alto de la imagen. El resultado es el siguiente:



Figure 14: Región Interés

En comparación de lo que se ve con la imagen completa la cámara del puzzle-bot.

¿Cómo aplicar un correcto preprocessamiento de tu imagen para identificar las líneas del carril?

A partir de un cierta región de interés se realiza el preprocessamiento de la imagen para limitar que el robot vea otras cosas y centrarse solamente en un pedazo de la imagen, así como también al momento de realizar operaciones matriciales esto sea mas fácil y rápido para optimizar el mayor número de recursos, en este caso el tiempo. Primeramente se toma en cuenta resaltar los bordes de las líneas negras, donde se aplica algunos filtros y técnicas que permitan aún más resaltar las líneas de interés como lo es el Gaussian Blur o métodos como la dilatación y erosión, así como el uso de diferentes kernels que ayuden a mejorar la región pertinente.

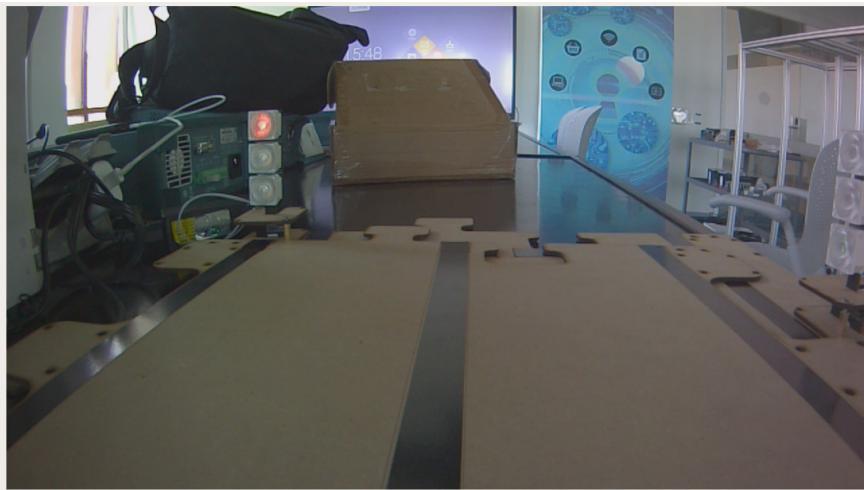


Figure 15: Región Interés

10.2.3 Suma Vertical:

En vista de que todos los valores de la imagen están en el rango $[0 : 255]$ se puede sintetizar la información de la región de interés sumando a través de las columnas (se tiene que pensar a la imagen como una matriz para asimilar este concepto). Esto entrega un vector cuyo tamaño iguala el ancho de la imagen y donde cada índice equivale a una posición o un pixel.

```
def sumVertically(self,img):  
    sum = img.sum(axis=0)  
    sum = np.float32(sum)  
    return sum
```

10.2.4 Obtención del Gradiente:

En el paso anterior se obtuvo un vector de posiciones y la cuantificación del color de la imagen por posición. Ahora, nótese que la pista es de colores claros (cercaños a 255) y la línea es de colores oscuros (cercaños a 0). Para detectar las líneas, queremos detectar cambios drásticos en el color y esto se hace calculando el gradiente del arreglo. Es decir, que tanto cambian los valores de color de todas las columnas de una posición a otra. Esto se puede realizar con funciones de numpy.

```
gradient = np.gradient(vertSum)
```

10.2.5 Separación de señales y Filtrado:

Véase el primer gradiente en la figura de abajo:

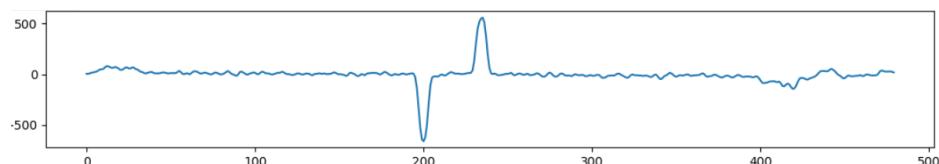


Figure 16: Primer Gradiante

Nótese que sobresalen dos picos: uno positivo y uno negativo. Esto se debe a que para los bordes de la línea izquierda el cambio es de valores cercanos a 255 a 0 (gran gradiente negativo). En cambio, en los bordes de la derecha el cambio es de 0 a 255 (gran gradiente positivo). Nos interesa separar los gradientes positivos de los negativos. Esto se hace creando copias del arreglo y filtrando porciones no deseadas de la señal por medio de thresholds.

```
def splitEdges(self,gradient,maxScaleFactor = 0.2, ...  
    minScaleFactor = 0.2):  
    left_gradient = gradient.copy()  
    right_gradient = gradient.copy()  
  
    left_gradient[left_gradient > LEFT_THRESHOLD] = 0  
    right_gradient[right_gradient < RIGHT_THRESHOLD] = 0  
  
    return left_gradient, right_gradient
```

10.2.6 Obtención del Segundo Gradiante:

Nuevamente, es necesario calcular el segundo gradiante para poder eliminar más ruido, el procedimiento es el mismo, pero se efectua sobre el gradiante separado en señales correspondientes a los contornos izquierdos y derechos:

```
secondLeftGradient = np.gradient(left)
secondRightGradient = np.gradient(right)
```

10.2.7 Segundo Filtrado:

Una vez obtenido el segundo gradiente, se vuelve a filtrar la señal para eliminar el ruido. Esta vez, se hace en función de un porcentaje del valor máximo.

```
def filterWithThreshold(self,gradient,scaleThreshold = 0.4):
    min = np.min(gradient) * scaleThreshold
    max = np.max(gradient) * scaleThreshold
    positive = gradient.copy()
    negative = gradient.copy()
    positive[positive < max] = 0
    negative[negative > min] = 0

    return positive + negative

...
secondLeftGradient =
    self.filterWithThreshold(secondLeftGradient)
secondRightGradient =
    self.filterWithThreshold(secondRightGradient)
```

10.2.8 Multiplicación de Gradiantes:

Las señales finales serán hechas a partir de la multiplicación de los primeros gradientes por los segundos gradientes. Estas señales serán las que serán utilizadas para extraer los contornos de la línea central. En nuestra implementación en particular, aun en esta etapa se tiene un poco de ruido. Es por esto por lo que se aplica un último filtro.

```
def filterNegative(self,gradient):
    gradient[gradient < 0 ] = 0
```

```

    return gradient

    ...

leftMul = left * secondLeftGradient
rightMul = right * secondRightGradient

leftPositive = self.filterNegative(leftMul)
rightPositive = self.filterNegative(rightMul)

```

10.2.9 Étapa Final: Corrimiento de Bits.

En esta última etapa, se determina en qué posición están los contornos de la línea. Para esto, se llevan a cabo los siguientes pasos para cada una de las señales (contornos izquierdos y contornos derechos):

- Se crean una copia de la señal
- Sobre la copia, se hace un corrimiento a la izquierda de los valores de la señal.
- Se comparan la copia con el original bajo la condición de que el arreglo con el corrimiento sea mayor al arreglo original. En otras palabras, se crea una máscara booleana.
- Las posiciones de los contornos consisten en los índices donde se almacenen valores booleanos positivos.

```

def shiftCompare(self,gradient):
    shifted = np.roll(gradient.copy(),1) # left shify
    compare = shifted > gradient
    return compare

    ...

# pixelShift:
leftCompare = self.shiftCompare(leftPositive)

```

```

rightCompare = self.shiftCompare(rightPositive)

# right and left slices positions
leftEdges = np.where(leftCompare)
rightEdges = np.where(rightCompare)

```

1.-¿Cómo aplicar diferentes operador matriciales como el gradiente para detectar los bordes del carril y clasificarlos en izquierdos o derechos?

Los operadores matriciales nos sirvieron para ubicar en donde se estaban detectando el mayor número de pixeles negros en relación a la escala de 0 a 255 donde había mejor región de estos, de tal forma que se grafican para ver en donde sube más el comportamiento esto dando en sí una región de punto medio del cual se va a guiar nuestro puzzlebot.

2.-Selección de los diferentes thresholds para filtrado de bordes no deseados durante la etapa de visión La selección de los THRESHOLD se hizo a partir del análisis de las gráficas y con conocimiento empírico para ver en que intervalos se tiene dichos picos, es decir hasta donde llegan los máximos puntos para después ver que ruido se elimina.

```

LEFT_THRESHOLD = -150 if minVal < -150 else -50
RIGHT_THRESHOLD = 150 if maxVal > 150 else 50

```

Al final se generá los histogramas en base al primer gradiente, segundo gradiente y la multiplicación del gradiente para ver la posición en la que se encuentra detectando el line detection, de tal forma ayudar a notar el comportamiento y ver los lados izquierdo y derecho.

10.3 Sistema del Seguimiento de líneas del Carril:

En este sistema se utiliza la detección de carril con visión computacional, y después se implementa un controlador clásico de tipo PD para realizar el seguimiento y navegar sobre la mayor parte de la pista.

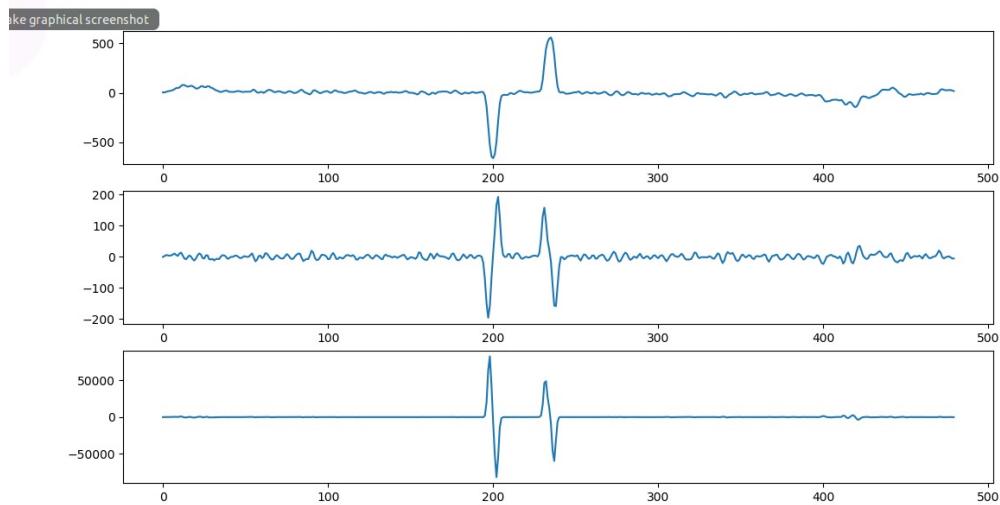


Figure 17: Gradientes

10.3.1 Arquitectura de ROS Propuesta:

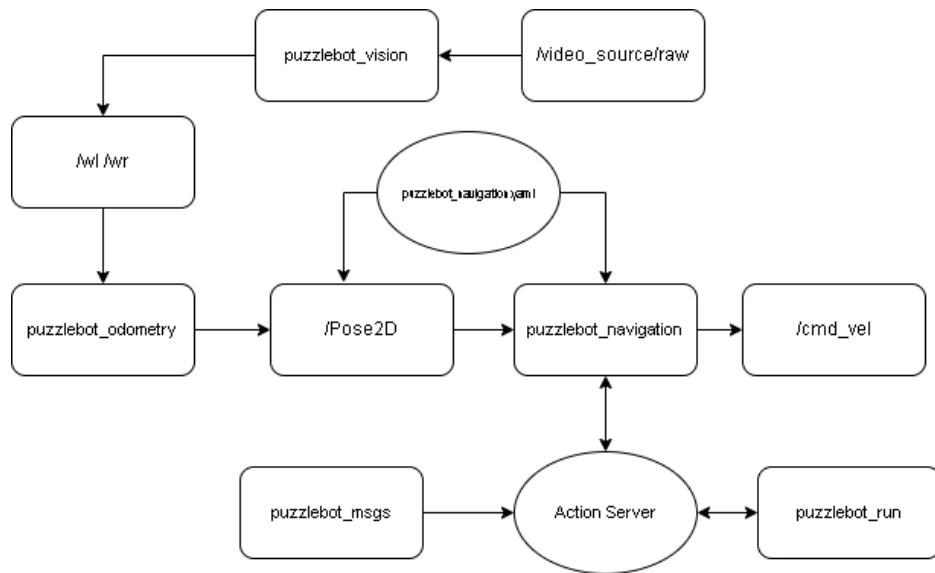


Figure 18: Seguimiento de línea

Para el módulo de detección y seguimiento de línea se va a utilizar primero una parte en el módulo de visión para la detección de dicha línea con el cálculo de gradientes, y después se va a utilizar el valor de los gradientes para relacionar al robot con su centramiento en la línea y poder interpretar los cambios en esos gradientes como curvas o despositionamientos del puzzlebot.

10.3.2 Sistema de control propuesto:

Durante la fase de diseño del módulo de seguidor de línea se concretó que solamente se utilizaría un control PD en vez de un control clásico PID, esto debido a que se requiere disminuir el error al mismo tiempo que se mantiene estable la velocidad del sistema, además por las características del sistema en específico, el error no se va sumando o acumulando de manera que sea indispensable la acción de control integral.

Una consideración importante para este sistema de navegación es que se define una velocidad lineal en el eje x constante y solamente se va a cambiar la velocidad angular en el eje z para controlar el posicionamiento central del puzzlebot y su trayecto en curvas.

```
TIME_THRESHOLD = 4.5
COUNTER_THRESHOLD = 4
self.counter = 4
self.counterNAN = 0
...
linealVel = 0.1
kp = 0.00075
kd = 0.000075
...
controlAngularSpeed = kp * angularErrorAbs +
(kd * (angularErrorAbs - self.pastAngularErrorAbs)) / angularTempDiff
```

10.3.3 Descripción de pruebas experimentales de funcionamiento:

Para las pruebas experimentales del funcionamiento del módulo de seguimiento de línea se dividió en dos etapas:

1. Trayectorias rectas: Se utilizaron varias secciones de pista recta para comprobar el funcionamiento del controlador y de que se mantuviera durante toda la trayectoria en la línea central sin tener sobreimpulsos al corregir el error angular o no hacer la corrección a una velocidad adecuada.

2. Trayectorias curvas: Se empleó la sección de curvas de la pista del reto para comprobar que siguiera la trayectoria durante las tres curvas (tanto en un sentido como en contrario). En esta parte se busca controlar el tiempo de trayectoria en curva antes de volver a detectar una línea y la velocidad angular en curva.

10.3.4 Ajuste de parámetros de control:

Para poder obtener los valores del controlador (kp y kd) que realizaran el seguimiento de línea sin tener un sobreimpulso y en una velocidad estable se hizo varias iteraciones sobre las secciones rectas de la pista.

Una noción a considerar es que el controlador angular está limitado a un rango de funcionamiento de -0.1 a 0.1, por lo que los valores de salida deben de estar en ese rango a una escala menor (5 a 7 decimales). Bajo la última consideración mencionada se inicio con valores diezmilésimos para ambas variables de control, más específicamente 0.0001 y se fueron ajustando sus valores conforme a las pruebas en pista.

Otro parámetro a controlar fue el de

COUNTER_THRESHOLD

este parámetro sirve para que el robot se estabilice antes de empezar a corregir el error de su posicionamiento con respecto a la línea central durante las secciones curvas. Debido a que durante la curva se pierde la detección de línea se tiene que reestablecer el control asegurandose de que no se esté detectando un valor atípico en vez de la línea.

10.3.5 Diagrama de bloques del sistema de control:

La referencia $r(t)$ es el valor esperado (0) para el sistema de control. El error $e(t)$ es la diferencia que hay entre la referencia y el valor sensado y procesado $h(t)$, este error puede ser un valor entre -256 y 256 ó NaN en caso de que ya

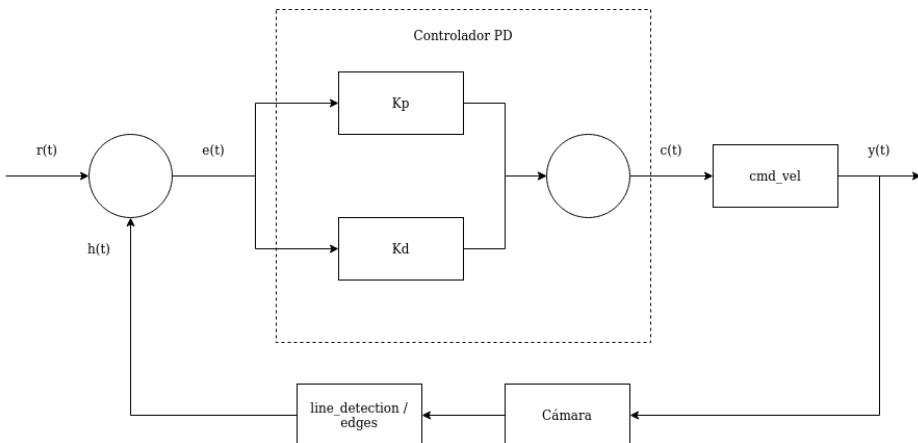


Figure 19: Diagrama de bloques sistema de control para seguidor de línea

no se detecte la línea. La salida del controlador para la velocidad angular en z $c(t)$ es el cálculo del controlador PD y este valor se va a utilizar para enviar como velocidad a los motores. Por último $y(t)$ es simplemente la salida final del sistema que realmente es interpretada como la nueva posición del puzzlebot en la pista.

11 Descripción del Sistema de Intersecciones:

Otro componente de visión computacional necesario para el proyecto es un sistema capaz de detectar las intersecciones de la calle. Esto será útil para saber en donde se tiene que frenar el robot así como para la alternancia entre los sistemas se seguidor de carriles y de navegación punto a punto.

Este sistema se encarga de usar técnicas de visión para detectar los contornos característicos de las intersecciones y publicar una bandera booleana cuando el robot esté en proximidad de la intersección,

11.1 Obtención de la imagen:

La imagen se tiene que obtener mediante un subscriptor al tópico de la cámara. En la sección: "Notas generales sobre la Obtención de Imágenes en Sistemas de Visión Computacional" se explica a detalle la metodología de obtención y publicación de las imágenes utilizadas en todos los nodos de visión.

11.2 Parámetros útiles:

Además del objeto CvBridge para procesar mensajes de imágenes de ROS, se definieron los siguientes parámetros:

```
# resize  
self.scale = IMG_SCALE_FACTOR  
  
# filtering parameters  
self.threshold = THRESHOLD  
  
self.minY = MIN_Y  
self.maxY = MAX_Y  
  
self.minArea = MIN_AREA  
self.maxArea = MAX_AREA
```

- scale: Se usa para escalar las imágenes de ROS y reducir el número de operaciones matriciales. El valor por defecto es reducir las imágenes al 50%
- threshold: Se usa para determinar si se publica la bandera con base en el número de contornos detectados.
- minY/maxY: Se usan para determinar si los centros de los contornos detectados están alineados (característica de la intersección).
- minArea/maxArea: Se usa para filtrar por área los contornos.

11.3 Preprocesamiento:

El preprocesamiento es muy similar al tipo de preprocesamiento de otros nodos de visión:

1. Se escala la imagen para reducir el número de operaciones.
2. Se cambia el canal de color a escala de grises.

3. Se ecualiza la imagen
4. Se aplica un difuminado gaussiano para aumentar los ruidos de alta frecuencia.

```
def imagePreprocessing(self,img):
    scale = 50
    width = int(img.shape[0]*self.scale/100)
    height = int(img.shape[1]*self.scale/100)
    img = cv2.resize(img,(height,width))
    rot = cv2.rotate(img,cv2.ROTATE_180)

    gray = cv2.cvtColor(rot, cv2.COLOR_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    gray =cv2.GaussianBlur(gray,(11,11),0)
    return gray,rot
```

11.4 Obtención del Área de Interés:

Si se tomase toda la imagen sería mucho más difícil filtrar los contornos pertinentes puesto que aparecerán muchos más que puede que cumplan con los requerimientos de filtrado. Es por esto por lo que se recorta la imagen de modo que solo se encuentren los contornos en el área de interés correspondiente a la carretera.

```
def sliceImage(self,img):
    h = img.shape[0]
    return img[int(h*0.4):,:]
```

11.5 Binarización de la Imagen:

El siguiente paso es binarizar la imagen para obtener solamente las figuras negras (las cuales la mayoría pertenecen a la carretera). Los parámetros fueron elegidos de acuerdo a como se adaptaban a las diferentes condiciones de luz.



Figure 20: Area de Interes

```
def thresholdImg(self,img):  
    retval, binary = cv2.threshold(img, 50, 255, cv2.THRESH_BINARY)  
    return binary
```



Figure 21: Area de Interes Binarizada

11.6 Identificación de Contornos:

El siguiente paso es identificar los contornos de la intersección a partir de la imagen binarizada y filtrarlos de modo que solo queden los contornos asociados a la intersección.

Los criterios para filtrar los contornos son los siguientes:

1. Que en area de los contornos se encuentre en el umbral delimitado por los parametros del constructor.

- Que el centro de los contornos este alineado verticalmente (para esto se usan los parámetros Y máxima y Y mínima del constructor).

Dicho filtrado de contornos de lleva a cabo en la siguiente función:

```
def edgeDetection(self,img):  
  
    filterC = list()  
    minY = self.minY  
    maxY = self.maxY  
    minArea = self.minArea  
    maxArea = self.maxArea  
  
    contours,hierarchy = cv2.findContours(img,cv2.RETR_TREE  
                                         ,cv2.CHAIN_APPROX_SIMPLE)  
  
    for contour in contours:  
        try:  
            M = cv2.moments(contour)  
            y = int(M["m01"]/M["m00"])  
        except:  
            continue  
  
        app = False  
        app2 = False  
  
        area = cv2.contourArea(contour)  
        if area < maxArea and area > minArea:  
            app = True  
  
        if y > minY and y < maxY:  
            app2 = True  
  
        if app and app2:  
            filterC.append(contour)  
  
    return filterC
```

En la función anterior, se encuentran los contornos y se guardan en una lista. Luego, se itera por la lista y se computan tanto el área como la coordenada del centro en el eje vertical (usando el método de momentos de OpenCv). Con estos datos, se determina lógicamente qué contornos conservar y qué contornos quitar.

A continuación, se muestra el resultado:

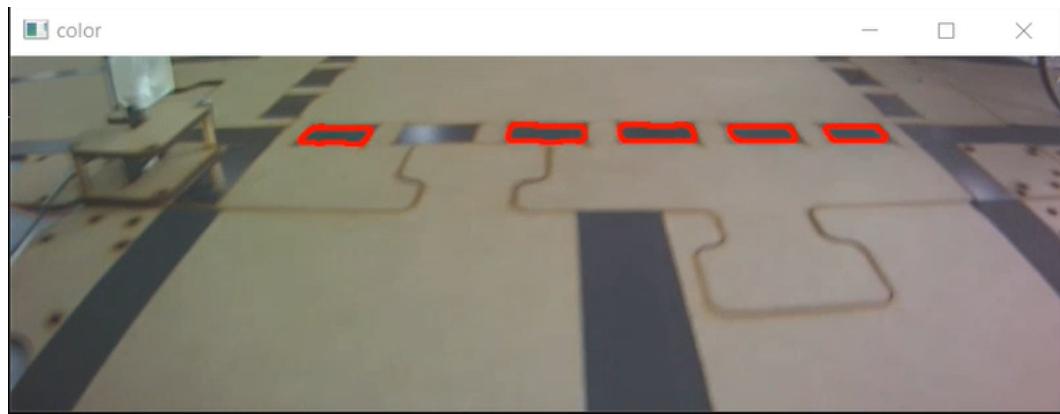


Figure 22: Contornos de la Intersección

12 Diagrama de la arquitectura de ROS

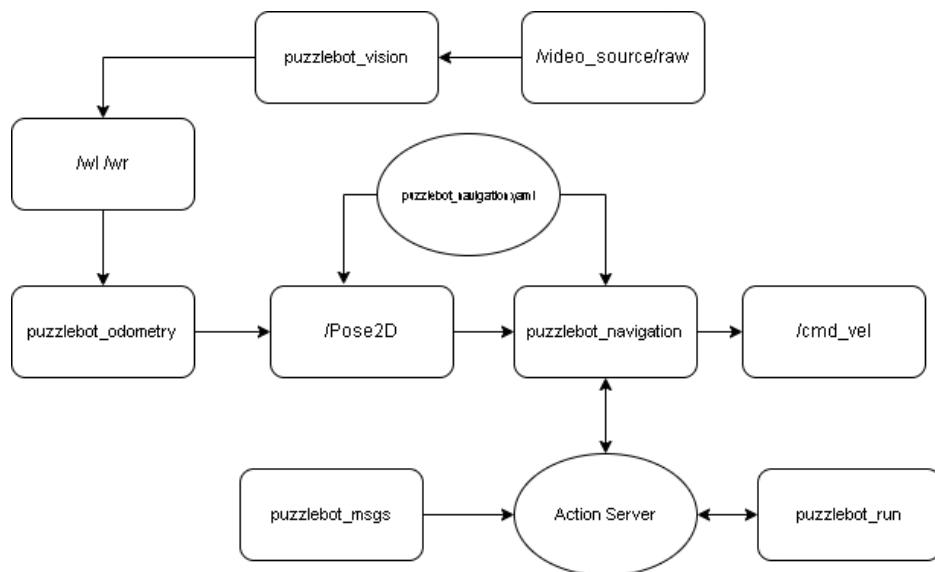


Figure 23: Primer Gradiante

Finalmente, si el número de contornos identificados después del filtrado es mayor al umbral aceptado, se publica una bandera booleana que indica que la intersección ha sido hallada:

```
flag = True if len(edges) > self.threshold else False  
self.detectedFlagPub.publish(flag)
```

13 Implementación del sistema de clasificación de señales de tráfico y toma de decisión de manejo:

Para la detección de señales de tráfico se implementaron 2 nodos de ROS:

- Segmentación de imágenes
- Predicción de señales de tráfico

13.1 Sistema de segmentación de señales de tráfico:

Este nodo se encarga de obtener una imagen de la cámara, procesarla, obtener regiones de interés que corresponden a potenciales candidatos a señales de tráfico, y publicar dichas imágenes.

13.1.1 Extracción de imagen:

Este método se hace de la misma manera que con el resto de los nodos de visión. Referise a la sección correspondiente.

En este caso se tienen un subscriptor para obtener la imagen del tópico de la cámara y dos publishers, el primero entrega la imagen segmentada al nodo de predicciones y el segundo una imagen con los candidatos resaltados para debugear:

```
self.cameraSub = rospy.Subscriber(camera_topic, Image,  
        self.image_callback)  
self.outputImagePub = rospy.Publisher(ROS_IMAGE_OUTPUT_TOPIC,  
        Image,queue_size=10)  
self.boxesImagePub = rospy.Publisher(ROS_IMAGE_SQUARE_TOPIC,  
        Image,queue_size=10)
```

13.1.2 Preprocesamiento:

Una vez que un frame es obtenido mediante un subscriber al nodo de la cámara, el único preprocesamiento que se realiza es rescalar la imagen, rotar la imagen 180 grados, y rescalarla con un factor del 50%. Dicho procedimiento se muestra a continuación:

```
width = int(img.shape[0]*self.img_scale_factor/100)
height = int(img.shape[1]*self.img_scale_factor/100)
img = cv2.resize(img,(height,width))
img = cv2.rotate(img,cv2.ROTATE_180)
```

13.1.3 Obtención de Círculos de Hough:

El siguiente paso consiste en obtener los círculos de Hough del frame en cuestión. A grandes rasgos, este algoritmo se vale de la Transformada de Hough y de algún método de detección de bordes como Canny para encontrar círculos asociados a los contornos de la imagen usando la siguiente ecuación:

$$r^2 = (x - a)^2 + (y - b)^2 \quad (4)$$

Se tiene una variable llamada acumulador y lo que se hace es computar círculos en cada pixel de los contornos y el radio del círculo corresponde al valor máximo del acumulador.

Por simplicidad, se uso la implementación de OpenCv que regresa una lista de listas de los siguientes parámetros para cada uno de los círculos:

- x: coordenada del centro
- y: coordenada del centro
- rad: radio del círculo.

Algunas consideraciones importantes respecto a esta función son las siguientes:

- Se requiere convertir la imagen a escala de grises para que el método funcione.
- Se requiere especificar un método, y un acumulador. Se usaron los valores por defecto recomendados en la documentación.
- Para la distancia mínima se usaron 100 pixeles considerando que las señales estarían separadas por lo menos por la carretera.
- Con base en las dimensiones de la imagen preprocesada, el intervalo entre el radio mínimo y máximo es apenas lo suficientemente grande para captar las señales y prescindir del mayor ruido posible.

La implementación se muestra a continuación:

```
def getHoughCircles(self,img):
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)

    circles_img = cv2.HoughCircles(gray,cv2.HOUGH_GRADIENT,1,100,
                                   param1=50,param2=30,minRadius=15,maxRadius=30)
    return circles_img
```

13.1.4 Segmentación y Publicación:

Una vez que los potenciales candidatos han sido filtrados, la lista pasa a otra función encargada iterar por la lista de candidatos y para cada uno:

- Extraer las coordenadas del centro y del radio.
- Dibujar en la imagen original un cuadrado alrededor de la región segmentada (con propósitos de debuggeo).
- Extraer la región segmentada, guardar esta imagen en una nueva variable, y publicarla a un tópico:

Al final de la iteración, de igual manera, se publica una imagen con todas las segmentaciones encerradas en cuadrados. El propósito de este tópico es



Figure 24: Segmentación

principalmente facilitar el debugeo de este nodo o de su interacción con otros nodos:

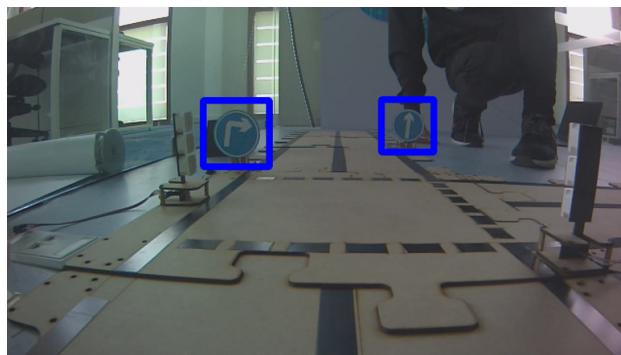


Figure 25: Imagen de Debugeo

La implementación se muestra a continuación:

```
def pubSegementedImages(self,circles,img):
    images = list()
    rectangles = list()
    outputImg = img.copy()
    if circles is not None:
        circles = np.round(circles[0, :]).astype("int")
        for circle in circles:
            x = circle[0]
            y = circle[1]
            rad = circle[2] + 10
            topLeftCorner = (x-(rad),y-(rad))
            bottomRightCorner = (x+(rad),y+(rad))
            outputImg = cv2.rectangle(outputImg,topLeftCorner,
                                      bottomRightCorner,(255,0,0),5)
            segmented = img.copy()[y-rad:y+rad,x-rad:x+rad,:]
```

```
try:  
    segmentedOutput =  
        self.bridge.cv2_to_imgmsg(segemented,encoding="bgr8")  
    self.outputImagePub.publish(segmentedOutput)  
except:  
    continue  
  
output = self.bridge.cv2_to_imgmsg(outputImg,encoding="bgr8")  
self.boxesImagePub.publish(output)
```

13.2 Sistema de clasificación de señales de tráfico:

Una vez que se tiene una segmentación de las señales, es necesario que el robot obtenga información de ellas y reaccione conforme a lo que representa cada señal. Para poder identificar las señales se hizo uso de un algoritmo de deep learning, Mismo que se describe en esta sección.

13.2.1 Descripción de la arquitectura:

Al inicio del proyecto, se había optado por una arquitectura de red neuronal de 3 canales usando los primeros 2 bloques convolucionales del modelo pre entrenado VGG16. Sin embargo, al momento de correr esta arquitectura en la jetson, esta no pudo correr el modelo. Es por esto por lo que se optó por correr una arquitectura de un solo canal de color. Misma que se muestra a continuación:

Model: "sequential"		
Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 150, 150, 32)	320
conv2d_1 (Conv2D)	(None, 73, 73, 32)	25632
max_pooling2d (MaxPooling2D)	(None, 36, 36, 32)	0
conv2d_2 (Conv2D)	(None, 34, 34, 32)	9248
conv2d_3 (Conv2D)	(None, 32, 32, 32)	9248
max_pooling2d_1 (MaxPooling 2D)	(None, 16, 16, 32)	0
flatten (Flatten)	(None, 8192)	0
dense (Dense)	(None, 32)	262176
dropout (Dropout)	(None, 32)	0
dense_1 (Dense)	(None, 32)	1056
dense_2 (Dense)	(None, 6)	198
<hr/>		
Total params: 307,878		
Trainable params: 307,878		
Non-trainable params: 0		

Figure 26: Arquitectura IA

La red neuronal tiene un tamaño de entrada de 150×150 pixeles. Esto se debe a que en los bloques convolucionales la imagen pierde dimensiones (información) debido a las convoluciones y a las capas de pooling. Entonces, se necesita evitar que los filtros de extracción de características sean demasiado pequeños. Se tienen 2 bloques de capas convolucionales con diferente número de filtros (se presentan en orden ascendente y en potencias de 2) seguidas de capas de max pooling.

Posteriormente se aplana el input en un vector que alimenta capas densas interconectadas con funciones de activación relu. Se incorporan capas de dropout para propósitos de regularización. La capa final tiene 6 neuronas de salida y una función de activación softmax. Esto se debe a que se requiere que la última capa sea una distribución de probabilidad de donde se seleccione la categoría con la probabilidad más alta.

13.2.2 Descripción del Dataset de Entrenamiento:

El dataset de entrenamiento original tenía 42 clases que corresponden a todas las clases de señales de tráfico en Reino Unido. Sin embargo, para fines de este proyecto solo se usó un subconjunto de 6 clases pertinentes. Dichas clases se despliegan en un directorio con la siguiente estructura:

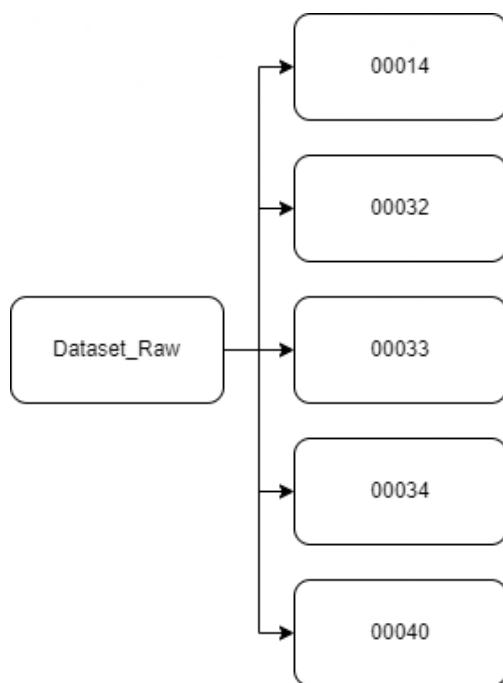


Figure 27: Estructura Dataset

En cada uno de estos folders, se tiene un conjunto de imágenes asociados a un tipo de señal de tráfico en particular. El desglose se muestra a continuación:

Folder	Categoría
00014	stop signal
00032	aplastame
00033	right signal
00034	left signal
00035	up signal
00040	around signal

13.2.3 Preprocesamiento

Mediante código se importaron las imágenes a un dataframe de pandas. Lo primero que se hizo fue aplicar a todas las imágenes la siguiente función de preprocesamiento:

```
IMG_TUPPLE_SHAPE = (150,150)
...
def imageProcessing(image):
    image = image.astype(np.uint8)
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    gray = cv2.resize(gray,IMG_TUPPLE_SHAPE)
    gray = gray.reshape(dims,dims,1)
    return gray.astype(np.float64)/255
```

La razón por la cual las imágenes se guardaron en un dataframe es porque es más sencillo hacer el preprocesamiento de las categorías. Primero cada label se codificó en números:

```
# generar diccionario de codificación
class2Label = dict()
label2Class = dict()
for i in range(len(IMAGE_LABELS)):
    class2Label[IMAGE_LABELS[i]] = i
    label2Class[i] = IMAGE_LABELS[i]

# preprocesamiento:
dataset["encoding"] = dataset["label"].apply(lambda x:class2Label[x])
```

	images	label	encoding
0	[[[0.9137254901960784], [0.8392156862745098], ...	stop_signal	0
1	[[[0.996078431372549], [0.996078431372549], [0...	around_signal	5
2	[[[0.3176470588235294], [0.3176470588235294], ...	up_signal	4
3	[[[0.996078431372549], [0.996078431372549], [1...	around_signal	5
4	[[[0.050980392156862744], [0.05882352941176470...	up_signal	4

Figure 28: Categorias Preprocesadas

Posteriormente, se aplico el one hot encoding para codificar cada categoría en un vector que representa una distribución probabilística donde la probabilidad de que la imagen corresponda a la label sea igual a uno:

```
X = np.asarray(list(dataset["images"]))
Y = np.asarray(dataset["encoding"])
Y = to_categorical(Y)
```

Posteriormente, se separaron los datos en datos de entrenamiento y prueba:

```
xTrain,xTest,yTrain,yTest = train_test_split(X,Y,test_size = 0.2,
                                             random_state = 0, shuffle = True)
```

Finalmente, se aplico la siguiente data augmentation sobre los datos para poder entrenar el modelo:

```
augmentedDataGen = ImageDataGenerator(
    rotation_range = 10,
    width_shift_range = 0.2,
    height_shift_range = 0.2,
    shear_range = 0.4,
    zoom_range=0.3,
    horizontal_flip=False,
    vertical_flip = False,
```

```

validation_split=0.2,
fill_mode='nearest'

)

testDataGen = ImageDataGenerator(
)

trainGenerator = augmentedDataGen.flow(
    X,Y,BATCH_SIZE,shuffle=True,subset = "training"
)
validationGenerator = augmentedDataGen.flow(
    X,Y,BATCH_SIZE,shuffle=True,subset = "validation"
)

testGenerator = testDataGen.flow(
    X,Y,BATCH_SIZE,shuffle=True
)

```

Los parametros fueron determinados a base de prueba y error. Se descubrió que no se pueden desviar más de 10 grados y es necesario poder captar tomas parciales de las señales o tomas demasiado cercanas. No es recomendable girar las señales porque entonces se tendrían flechas apuntando hacia ambos lados tanto para izquierda como para derecha, y el modelo sería obsoleto. Asimismo, note se usan tanto datos de entrenamiento como datos de validación.

13.2.4 Resultados del entrenamiento (metricas)

Por motivos de tiempo, solamente se usaron dos métricas:

- Loss: Se puede interpretar como que tan lejos están las predicciones de ser exactas cuantificado en números de desviaciones estándar.

- Validation Accuracy: Comprende la precisión con la cual la red valida su desempeño en los datos de validación después de cada época. Se decidió ocupar esta métrica porque lo que nos interesaba era que el modelo fuera preciso en datos desconocidos, y los datos de validación equivalen a exponer la red a datos de prueba desconocidos al final de cada época.

Este es el proceso de compilación en el código:

```
model.compile(
    optimizer = "rmsprop",
    loss = "categorical_crossentropy",
    metrics = ["accuracy"]
)
```

En general, los resultados fueron bastante aceptables para todos los modelos generados, he aquí el comportamiento del último:

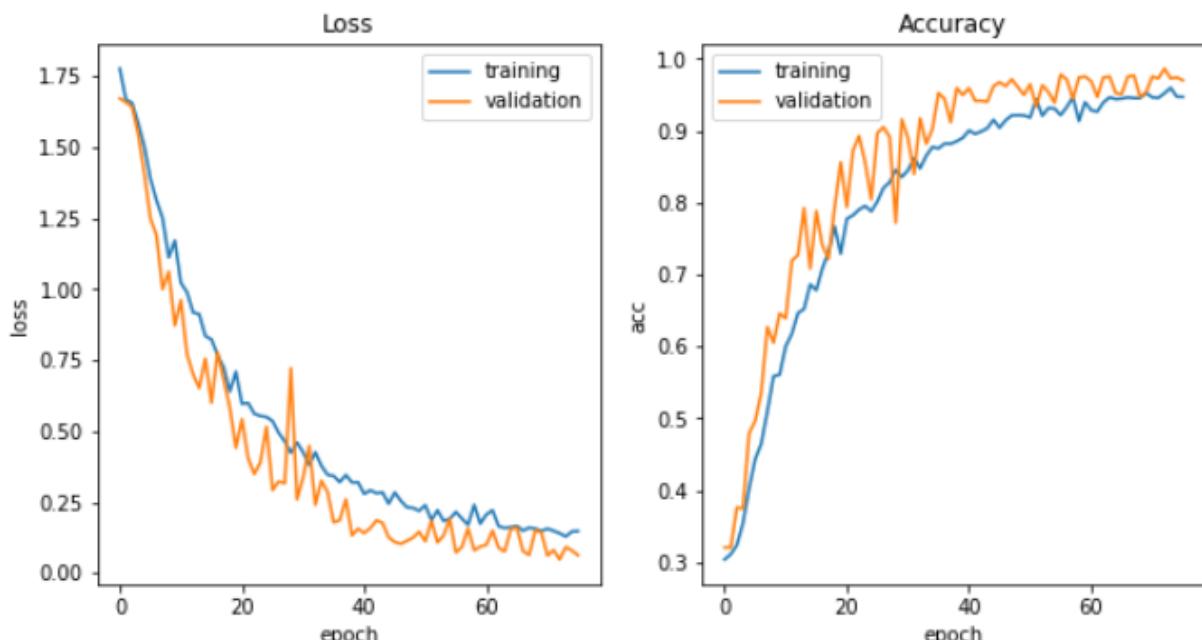


Figure 29: Desempeño del Módelo

Notese que los datos de validación siempre son un poco más precisos que los de entrenamiento; del mismo modo que la pérdida es menor para los datos de validación. Este comportamiento es benigno para los modelos de aprendizaje

profundo, y como las curvas estan separadas por menos de 10% se puede considerar que el modelo no esta cayendo en overfitting y esta generalizando bien a los datos desconocidos.

13.2.5 Validación de Red Neuronal:

Para validar la red de neuronal, se recurrio a someter la red a correr pruebas en un dataset de validación. Este dataser se crea separando los datos de entrenamiento para formar un subconjunto de datos de validación. De este modo, el modelo sería probado al final de cada epoch y la precisión en esta prueba fue la prueba para determinar qué tan bueno es el módulo.

13.3 Descripción de las pruebas experimentales de funcionamiento:

En cuanto a las pruebas de funcionamiento, lo que se hizo fueron pruebas los scripts de manchester intentando reconocer las señales de tráfico que serian utilizadas en la pista. Dicho script se corrio en una computadora remota. Adicionalmente, se intento correr de manera local en la jetson. Sin embargo, al intentar cargar el módulo, este rompia el nodo de la cámara y resultaba imposible recibir imagenes:

```
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829985.476790401]: failed to capture next frame
[Take graphical screenshot] -- failed to retrieve next image buffer
[ERROR] [1648829990.906316337]: failed to capture next frame
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829991.980535555]: failed to capture next frame
[gstreamer] gstCamera -- end of stream (EOS)
CONSUMER: ERROR OCCURRED
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829992.980840138]: failed to capture next frame
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829993.981398940]: failed to capture next frame
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829994.981762533]: failed to capture next frame
2022-04-01 17:19:55.443191: I tensorflow/core/platform/default/subprocess.cc:304] Start cannot spawn child process:
[gstreamer] gstDecoder -- failed to retrieve next image buffer
[ERROR] [1648829995.982127012]: failed to capture next frame
```

Figure 30: Error en la Jetson

Otro enfoque fue grabar un video del robot recorriendo la pista con señales y replicar el algoritmo de segmentación en una computadora remota. Entonces se procesa y se predice sobre cada frame del video como se muestra a continuación:

Figure 31: Deep Learning Experimental

13.4 Nodo de Predicciones:

Este nodo se encarga de suscribirse al tópico de imágenes de señales de transito segmentadas y hacer predicciones con base en ellas. Dichas predicciones se mandan al nodo de navegación mediante un publisher de tipo string para tomar las desiciones de manejo pertenentes con base a las señales de transito.

13.4.1 Sobre la obtención de las imágenes:

En este caso, para correr tensorflow, es necesario tener un script corriendo en python 3. Sin embargo, las dependencias de ROS no funcionan en esta versión. Es por esto, por lo que es necesario que el callback del subscriber reciba el mensaje de la cámara de la siguiente manera:

```
def imageCallback(self,img):
    self.image =
        np.frombuffer(img.data, dtype=np.uint8).reshape(img.height,
                                                       img.width, -1)
```

13.4.2 Preprocesamiento:

Lo único que se hace después es transformar la imagen a escala de grises (el número de canales que recibe la red neuronal), se escala al tamaño de entrada del modelo, se equaliza, y se normaliza la imagen:

```
def imageProcessing(self,image):
    image = image.astype(np.uint8)
    gray = cv2.cvtColor(image,cv2.COLOR_BGR2GRAY)
    gray = cv2.equalizeHist(gray)
    gray = cv2.resize(gray,IMG_TUPPLE_SHAPE)
    gray = gray.reshape(1,dims,dims,1)
    return gray.astype(np.float64)/255
```

13.4.3 Predicciones:

Esto se hace metiendo la imagen al modelo y extrayendo la predicción de la clase más alta junto con la probabilidad de que sea correcta:

```
def getClassName(self, classNo):
    labels = {
```

```

0: 'stop_signal',
1: 'aplastame',
2: 'right_signal',
3: 'left_signal',
4: 'up_signal',
5: 'around_signal'}
return labels[classNo]

...
prediction = model.predict(img)
index = np.argmax(prediction)
label = self.getClassName(index)
proba = prediction[0][index]

```

13.4.4 Publicación de la categoría:

Posteriormente, antes de publicar se toma en consideración el número de repeticiones (las veces seguidas que una categoría se repite) y la probabilidad (que sea mayor a cierto umbral).

```

def pubLabelFlag(self,label):
    if self.lastLabel == label:
        self.labelCounter += 1
    if self.labelCounter > self.labelThreshold:
        self.labelCounter = 0
        flag = True if self.lastLabel != "not_found" else False
        self.labelPub.publish(self.lastLabel)
        self.detectedFlagPub.publish(flag)

    self.lastLabel = label

label = "not_found" if proba > 0.95 else label
self.labelPub.publish(label)
self.pubLabelFlag(label=label)

```

...

Es decir, antes de publicar una categoría, es necesario que esta se repita al menos 15 veces. Esto garantiza que no se publique un falso positivo. También, es necesario que la probabilidad de que la predicción sea correcta sea mayor a un 90%.

13.5 Toma de decisión de manejo:

13.5.1 Implementación de ROS propuesta para el control de condiciones de manejo:

La integración del módulo de deep learning (detección de señales) y el módulo de navegación (seguidor de línea y navegación punto a punto) es una de las etapas finales en la integración de todos los módulos de nuestra propuesta de solución al reto con el puzzlebot.

En esta integración el tópico de predicciones basado en deep learning va a clasificar la señal que previamente haya procesado el módulo de visión para detección de señales, y la salida del tópico será una etiqueta indicando cual es la señal encontrada. Esta etiqueta será utilizada como un 'switch' para establecer condiciones al controlador general del puzzlebot.

13.5.2 Siga:

Para la señalización de siga o ("up-signal") se establece el primer estado del control de intersecciones debido a que esta señal indica que se debe de seguir recto en esta intersección.

```
self.STATE = 0  
...  
if self.STATE == 0:  
    goal_pose.x = curr_pose.x + self.dist  
    goal_pose.y = curr_pose.y  
    goal_pose.theta = curr_pose.theta
```

13.5.3 Vuelta a la derecha:

Para la señalización de vuelta a la derecha o ("right-signal") se establece el segundo estado del control de intersecciones y se espera que además realice también el tercer estado debido a la condición de curva en dos partes en la intersección donde se desea una vuelta.

```
self.STATE = 1  
...  
elif self.STATE == 1:  
    goal_pose.x = curr_pose.x  
    goal_pose.y = curr_pose.y - (self.dist / 2.0) - 0.2  
    goal_pose.theta = curr_pose.theta - (pi / 2.0)  
else:  
    goal_pose.x = curr_pose.x - (self.dist / 2.0)  
    goal_pose.y = curr_pose.y  
    goal_pose.theta = curr_pose.theta
```

13.5.4 Stop:

Para la señalización de stop o ("stop-signal") simplemente se hace un override a las velocidades que controlan el movimiento del puzzlebot, es decir la lineal en el eje x y la angular en el eje z, y se envían a 0.

```
cmd_vel.linear.x = 0  
cmd_vel.angular.z = 0
```

13.5.5 No speed limit:

Para la señalización de no speed limit o ("aplastame") se calcula una velocidad lineal en el eje x mayor a la ya establecida en un factor de 1.5, este factor debido a que el aumento en velocidad debe de ser notorio pero no debe de comprometer la estabilidad de la navegación.

```
cmd_vel.linear.x *= 1.5
```

14 Implementación del sistema de integración final:

14.1 Descripción lógica del sistema de integración final para manejo autónomo:

La integración final de la propuesta de solución al reto trabajado con el puzzlebot se va a ocupar del control general del puzzlebot para que haya un manejo autónomo en 3 etapas.

1. Inicializar nodos de visión y deep learning: Es importante que los nodos de visión se inicialicen primero que el resto ya que de ellos dependerá muchas de las señales debido a que la cámara es el sensor principal del puzzlebot y con el que se va a realizar las demás acciones, además el nodo de deep learning puede ser inicializado después de los de visión para recibir las señales de tráfico en caso de detectarse alguna.
2. Inicializar nodos de odometría y navegación: Para la comunicación con los actuadores (en este caso motores) se deben de inicializar los nodos que van a estar a cargo de procesar ciertas señales que vienen de los nodos de visión, para poder controlar al robot sobre la pista.
3. Controlar la navegación: Por último se va a tener un controlador general el cual va a estar recibiendo la información de los nodos de visión para detección de señales, intersecciones y de semáforos, y del nodo de navegación para recibir los posibles valores de navegación. Con estos valores va a priorizar las acciones (que únicamente ocurren cuando detecta una intersección) y de lo contrario va a navegar utilizando la información del seguidor de línea, tomando siempre en cuenta las señales obtenidas con deep learning y sus propiedades.

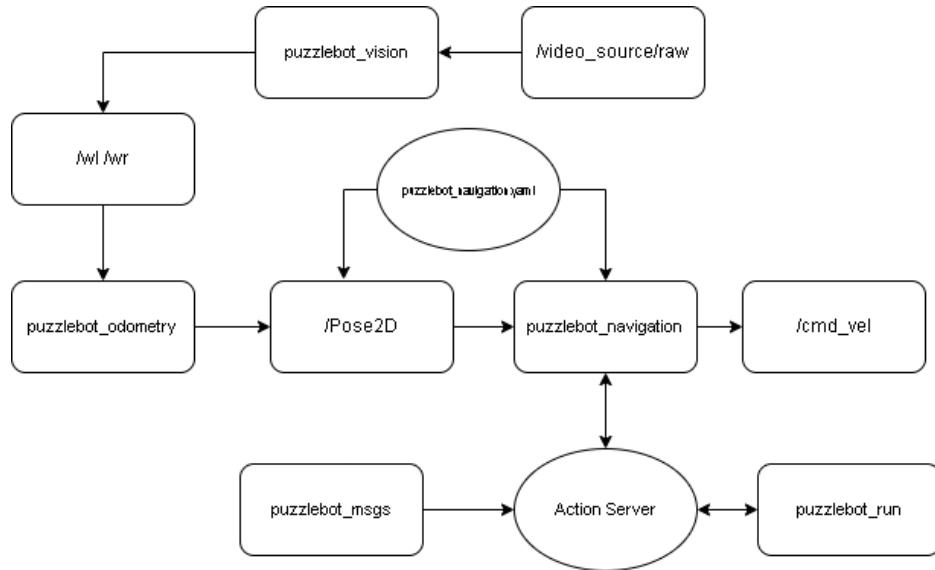


Figure 32: Arquitectura ROS

14.2 Diagrama General de la arquitectura de ROS:

15 Análisis de Resultados experimentales y propuestas de mejora:

15.1 Sistema de manejo punto a punto:

En esta sección se proponen 2 posibles mejoras:

1. Mejorar la detección de intersecciones: El mayor problema al integrar todos los módulos del puzzlebot en uno fue el de mantener las funciones de detección de intersección de manera adecuada para poder realizar la navegación de punto a punto. Debido a las condiciones cambiantes de la luz y la visualización de la cámara fue difícil integrar esta parte ya que no siempre se estaba detectando la intersección, es por esto que mejorar este sistema para distintas condiciones de visión sería un paso apropiado para resolver el problema.
2. Robustecer el sistema de odometría: Uno de los principales problemas con el sistema de manejo de punto a punto es que es muy dependiente de la odometría debido a que se debe de conocer la posición precisa del puzzlebot al momento de querer realizar una acción de manejo de punto a punto y se

debe de calcular la posición final deseada de un punto en la pista.

Es posible que en vez de que se establezcan cambios en (X , y) de forma arbitraria se pueda calcular el punto final deseado basado en el ángulo del puzzlebot al momento de querer realizar la acción, y la distancia que se espera recorrer.

15.2 Sistema de detección de luces de parada:

Sobre el sistema de detección de luces se destaca las condiciones cambiantes de luz, donde alrededor del día si tienen diferentes tonos de los colores de los semáforos lo que ocasiona que el filtrado de pixeles tanto verdes o rojos pueda cambiar, haciendo el tono más blanco para identificar, donde la solución como propuesta de mejora es la búsqueda de semáforos para después identificar que luz se encuentra encendida, ya sea por patrón de luminosidad, o simplemente en la identificación de posición de cada luz del semáforo, otro mejora puede ser el ajuste de búsqueda de THRESHOLD para cada condición de luminosidad que puede ser de gran ayuda al momento de cada cambio de clima con el pasar del tiempo.

15.3 Sistema de detección de seguimiento del carril

En general se tuvieron que hacer varias iteraciones del nodo de seguimiento de línea debido a que las condiciones cambiantes de luz para detectar la línea central, además de tener que resolver el problema de las curvas y del déficit de la cámara al tener un rango mínimo de 20cm por delante del puzzlebot. Al final este sistema fue uno de los que se completó satisfactoriamente con la propuesta de diseño y su implementación tanto en visión como en un controlador clásico de tipo PD. Para mejorar este nodo hay varias propuestas posibles, primero que nada se buscaría hacer un seguir de carril con mejores capacidades de detección al utilizar una segunda cámara orientada perpendicular al suelo como en un seguidor de carril convencional, de esta misma forma se podría utilizar sensores fotosensibles en vez de cámara pero que estén en esta misma orientación donde se reduce el déficit de visión de 20 a 2-5 cm.

15.4 Sistema de clasificación de señales de tráfico

En esta sección se proponen 3 posibles mejoras:

1. Mejorar la lógica del sistema de segmentación: El principal problema con el sistema de segmentación fue el hecho de que a veces enviaba imágenes que no contenían información pertinente para la red neuronal (la imagen no contenía señales de tráfico). Entonces, se podría mejorar dicho sistema para que se segmenta mejor. Es decir, hay que implementar más sistemas de clasificación (por área o por forma) para garantizar que solo se envíen círculos. Adicionalmente, en condiciones más ideales, se podría comprobar haciendo revisando que las coordenadas del centroide de los círculos detectados coinciden con nuestra implementación de segmentación por color (se realizó un sistema de segmentación por color que fue descartado por que las condiciones del salón no le favorecían). Estas propuestas servirán para hacer la lógica del sistema más robusta y garantizar que por iteración sólo se envíe una sola imagen.

2. Implementar una acción para la segmentación de señales de tráfico:

Como siguiente mejor proponemos implementar el nodo de segmentación de imágenes a manera de acción. La ventaja de esto es que las acciones tienen una retroalimentación. Entonces, resulta sencillo poder llamar la acción solo en el momento en el que sea requerida y solo segmentar señales de tráfico cuando se alcance una condición específica (ej. cuando se llegue a las intersecciones). Asimismo, el robot sólo ejecutaría el comando asociado a la acción correspondiente al recibir la retroalimentación de la acción.

3. Mejorar el modelo de Deep Learning:

Esto solo se haría si las otras 3 propuestas no sirven y pues consiste en seguir experimentando con la arquitectura del modelo, los hiperparámetros y hacer el data augmentation más robusto. Podría servir agregar imágenes de las señales de tráfico de la pista para igualar la distribución de datos de la red. Esto se podría hacer modificando el nodo para grabar video para que actúe con el nodo de segmentación y predicciones y guarde en un directorio las imágenes que fueron clasificadas con alguna de las categorías de señales de tráfico.

16 Conclusions

En conclusión, ROS nos permite modular de manera ordenada cada sección de robot, de tal forma que es más fácil la implementación por medio de nodos y paquetes, esto ayuda de a tener mayor claridad de lo que se está haciendo en cada una de las secciones. El puzzlebot se rige de variables que se van interconectando entre sí, formando un sistema que depende de cada paquete para su correcto funcionamiento. Podemos concluir que se requiere en gran parte de experimentación, siendo el entorno físico un gran desafío para adaptar en código el funcionamiento del puzzlebot, tomando en cuenta factores como la luz, posición de la cámara, detección de falsos, entre otras cosas que se van encontrando mientras se sacan debugs. La implementación de sistemas más completos se basan en estos pequeños sistemas, siendo un hardware limitado que puede llevar a la creación de un sistema grande y complejo.

References

- [1] IBM. *¿Qué es la visión artificial?* 2022. URL: <https://www.ibm.com/mx-es/topics/computer-vision>.
- [2] OpenWebinars. *Qué es ROS (Robot Operating System)*. 2017. URL: <https://openwebinars.net/blog/que-es-ros/>.
- [3] RedHat. *¿Qué es YAML?* 2021. URL: <https://www.redhat.com/es/topics/automation/what-is-yaml>.
- [4] SAS. *Deep Learning*. 2021. URL: https://www.sas.com/es_mx/insights/analytics/deep-learning.html.