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 7, 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 involucrase más en la implementación de un sistema de control inteligente y robusto.(OpenWebinars 2017)

2 Marco Teorico

2.1 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 suscribers que manejan el flujo de datos, multiplexición de la información, creación y destrucción de nodos, nodos en multiprocesamiento, parámetros de servidor y testeo en simuladores para un mejor desempeño en pruebas físicas.

2.2 Launch File para múltiples ejecuciones

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 inciar 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.(ClearpathRobotics 2015)

2.3 Entorno 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

3 Odometria:

En robots diferenciales, es común usar la odometría de un robot para llevar a cabo el control posicional de un robot. Para el caso de una simulación de Gazebo, 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:

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

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

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

Esta 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 información de la geomtria del robot

3.1 Conveción atan2 de orientación

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

4 Control Posicional:

El algoritmo requiere de 2 argumentos:

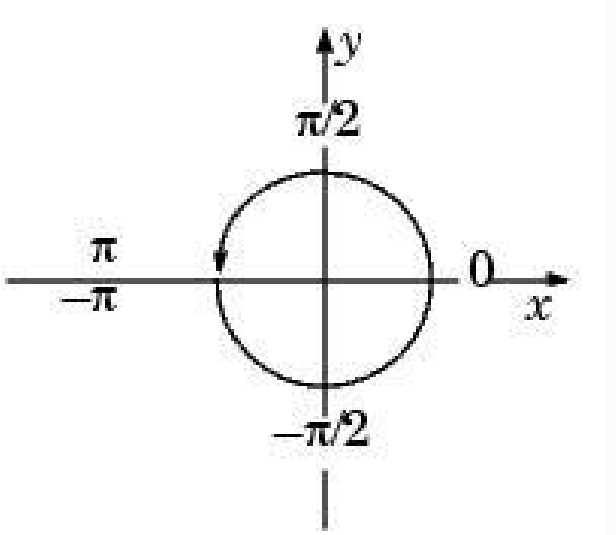

Figure 1: Conveción atan2

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:

5 Sistema de Navegación

Este sistema permite llevar al robot a una posición y orientación especifica. La posicion 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 Teorico).

6 Explicación del Funcionamiento:

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

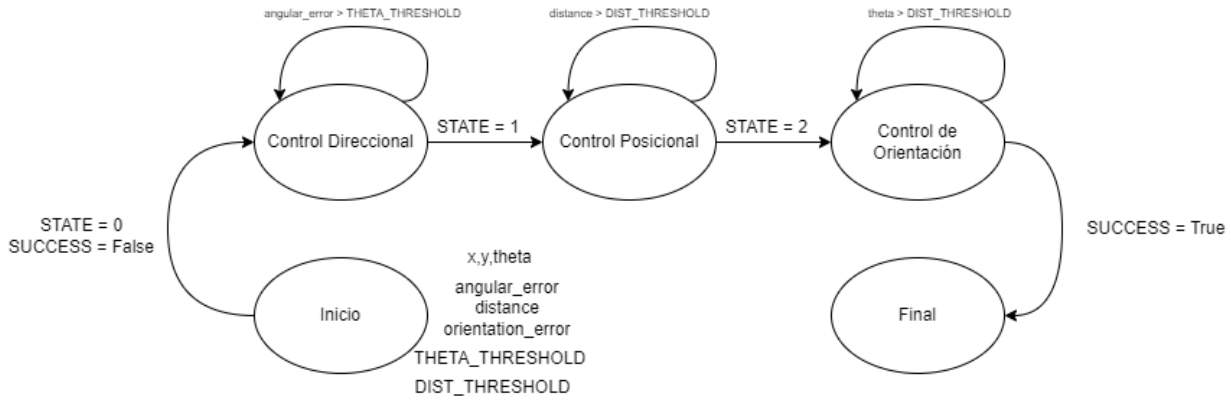

Figure 2: Máquina de estados Nodo de Navegación **??**

.

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 de tiene 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 esta dado por la ecuación (3).

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 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 euclidiana 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 euclidiana 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.

6.1 Implementación en ROS:

Este sistema consta de los siguientes componetes de ROS:

6.1.1 Nodo de Odometria:

Este nodo se encarga de subscribirse a los tópicos de velocidad angular de las llantas del robot ($/\omega r$ y $/\omega l$), 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 conveción 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:

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

6.1.2 Nodo de Navegación

Para el differencial 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 monitreo 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 considera una coordenada en (0,0).

6.1.3 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:

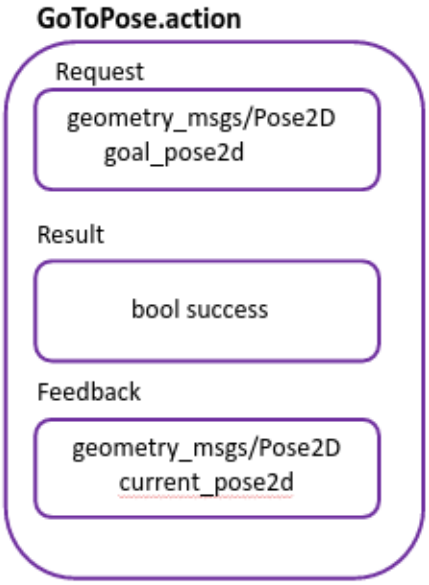

Figure 3: Acción GoToPose

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.

6.1.4 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 a corregir en 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, utiliazando la función lambda para después ponerlos en orden de la función, quitando la información inncesesaria 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 correción 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

6.1.5 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 encuentré 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 correción de la velocidad angular y proponer una velocidad lineal en donde se permita hacer la correción del trayecto.

2.-Ajuste del sistema de control de velocidad lineal y angular, elección de constantes de control (Kp, Kd) Las constantes se hizo en base a la explicación anterior, donde a través de un suscriber eecrito 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 correción del valor del Error abosoluto 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 resulto para mejorar el proceso de navegación.

6.1.6 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 parametros pertinentes y definiendolos 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
```

7 Sistema 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 linea central de la carretera.

7.1 Obtención y publicación de imagenes:

En estos algoritmos es común tener que tranformar mensajes de imagen de ROS a formatos que pueden ser procesados por OpenCV.De la misma mánera fue necesario publicar imagenes con propositos de Debuggeo.

Nótese que todos los nodos que conllevan procesamiento de imágenes, de-

tección 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, el proceso se obtención de y publicación de imagenes únicamente se explica en esta sección.

7.1.1 Obtención de Imagenes:

El proceso de obtención de imágenes se realiza mediente 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 pre-procesar), 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 llamara 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.

7.2 Publicación de Imagenes:

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

- 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)
...
# Covertir OpenCv -> ROS
output = self.bridge.cv2_to_imgmsg(binarized)
#Publicar Tópico
self.edgesImagePub.publish(output)
```

7.2.1 Sobre la Publicación de Imagenes:

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 usabala 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, altenramos entre publicar las imagenes con formato o sin formato segun los canales de color:

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

7.3 Detección de semaforos:

Este nodo se encarga de la detección de lineas de trafico rojas o verdes captando las imagenes de la cámara.

```
self.redLightFlag = rospy.Subscriber(ROS_RED_LIGHT_DETECT_TOPIC,...
Bool, self.redLightFlag_callback)
self.greenLightFlag = rospy.Subscriber(ROS_GREEN_LIGHT_DETECT_TOPIC,...
Bool, self.greenLightFlag_callback)
```

Tomando en cuenta la creación de dos suscribers mandados en el script de traffic lights.

7.4 Detección de Lineas:

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 centran alrededor del pixel 100 ($error = 0$) mediante la corrección de la trayectoria del robot.

7.4.1 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 lineas. Buscamos cambios en los valores de la imagen y esto se aprecia mejor en un unico 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. $0 \rightarrow 255$). 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)
```

7.4.2 Obtención de una Región de Interes:

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 4: Región Interés

.

En comparación de lo que se ve con la imagen completa la cámara del puzzlebot.

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

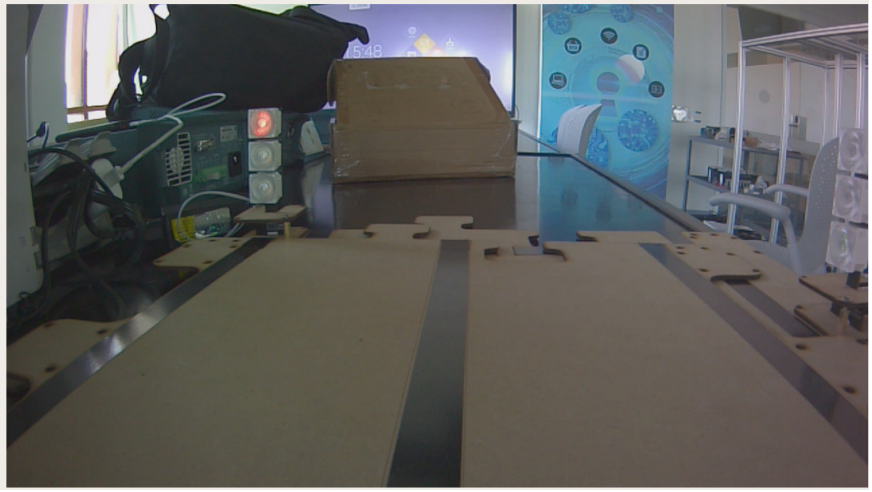

Figure 5: Región Interés

.

A partir de un cierta región de interés se realiza el preprocesamiento 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.

7.4.3 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
```

7.4.4 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 (cercanos a 255) y la línea es de colores oscuros (cercanos 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 posicón a otra. Esto se puede realizar con funciones de numpy.

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

7.4.5 Separación de señales y Filtrado:

Véase el primer gradiente en la figura de abajo:

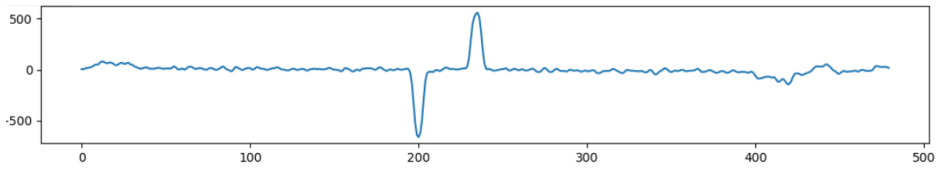

Figure 6: Primer Gradiante

.

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
```

7.4.6 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)
```

7.4.7 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)
```

7.4.8 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)
```

7.4.9 Étapa Final: Corrimiento de Bits.

En esta última etapa, se determina en qué posición están los contornos de la linea. 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 o 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 izquiero y derecho.

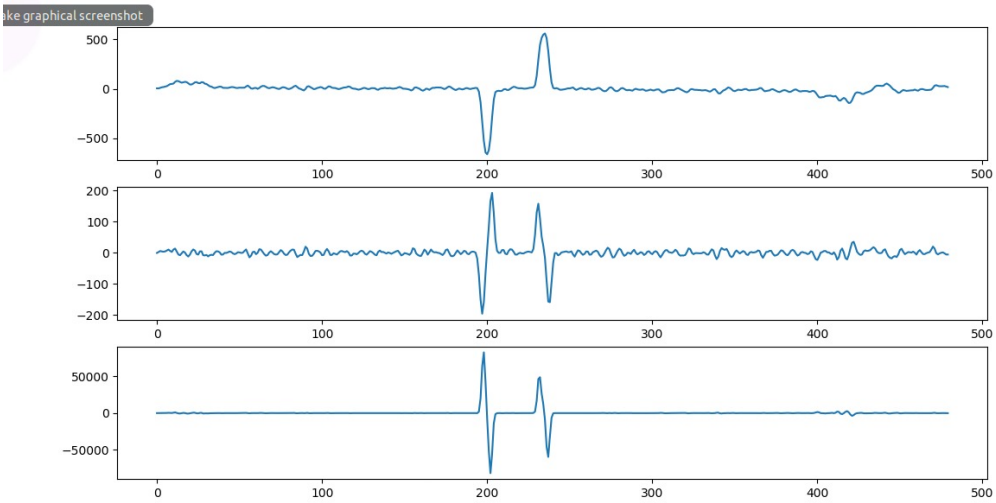

Figure 7: Gradientes

.

8 Diagrama de la arquitectura de ROS

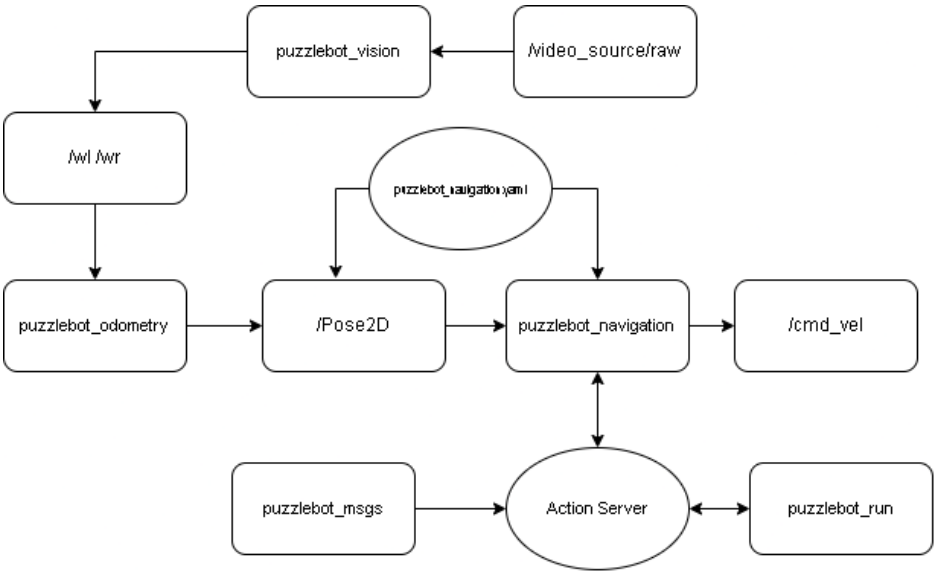

Figure 8: Primer Gradiante

.

9 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] ClearpathRobotics. *Launch Files*. 2015. URL: https://www.clearpathrobotics.com/assets/guides/melodic/ros/Launch%20Files.html.

[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.