Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Implementación de robótica inteligente (Gpo 501)

Reporte: Reto semanal 5

Alumno
José Diego Tomé Guardado A01733345
Pamela Hernández Montero A01736368
Victor Manuel Vázquez Morales A01736352
Fernando Estrada Silva A01736094

Fecha de entrega
Jueves 23 de Mayo de 2024

RESUMEN
En este trabajo se implementó una capa de seguimiento de líneas utilizando ROS y procesamiento de imágenes con OpenCV, destinada a asistir en la etapa de decisión de un robot autónomo Puzzlebot. Para ello, se emplea una tarjeta embebida Jetson Nano de NVIDIA, que ejecuta Ubuntu. Este sistema integrado permite al Puzzlebot detectar y seguir líneas en su entorno, optimizando su navegación mediante la captura y procesamiento eficiente de imágenes, y proporcionando una base robusta para la toma de decisiones autónomas en tiempo real.

OBJETIVOS
El objetivo principal es implementar un sistema de seguimiento de líneas para un robot autónomo Puzzlebot utilizando ROS y procesamiento de imágenes con OpenCV en una tarjeta Jetson Nano NVIDIA ejecutando Ubuntu, optimizando la capacidad del robot para detectar y seguir líneas en su entorno y ajustar su velocidad en consecuencia. Dentro de los objetivos particulares se encuentran:

1. **Diseño de Pista:** Crear un diseño de pista adecuado que permita la evaluación efectiva del sistema de seguimiento de líneas. La pista debe incluir curvas para probar la robustez y adaptabilidad del sistema.
2. **Aplicación de la Detección de Imagen en ROS usando OpenCV:** Investigar y seleccionar metodologías efectivas para la detección de seguidores de línea, ajustándose para su integración en un sistema de procesamiento en tiempo real utilizando OpenCV. Esta integración debe optimizarse para funcionar dentro del entorno de ROS, permitiendo un procesamiento eficiente y preciso.
3. **Detección de Imagen:** Implementar y ajustar los algoritmos de detección de imagen necesarios para identificar las líneas en diversas condiciones de iluminación. Estos algoritmos deben estar optimizados para el hardware de la plataforma Jetson Nano, garantizando un rendimiento robusto y confiable.
4. **Etapa de Decisión del Robot:** Desarrollar una etapa de decisión que regule la velocidad angular y lineal del robot, permitiendo que el Puzzlebot se ajuste de manera precisa a la pista detectada. Esta etapa debe asegurar una navegación eficiente y precisa, adaptándose dinámicamente a las condiciones cambiantes del entorno.
5. **Pruebas y Ajuste:** Realizar pruebas exhaustivas del sistema en la pista diseñada, ajustando los algoritmos de detección y las estrategias de decisión según sea necesario. Estas pruebas deben identificar y resolver cualquier problema de rendimiento, garantizando que el sistema funcione de manera óptima en diversas condiciones de operación.

INTRODUCCIÓN

La visión por computadora es uno de los desafíos más significativos en el campo de la robótica, especialmente en el desarrollo de vehículos autónomos. La capacidad de un automóvil para interpretar su entorno a través de imágenes y tomar decisiones en tiempo real es esencial para su funcionamiento seguro y eficiente. Sin embargo, la implementación de

sistemas de visión por computadora en estos vehículos requiere la consideración de múltiples factores, como la iluminación, el ruido de fondo, la redimensión de imágenes y la similitud entre objetos.

El desarrollo de sistemas de visión por computadora ha sido impulsado por las necesidades de la industria en cuanto a la automatización de procesos tradicionalmente realizados por humanos, como la detección, reconocimiento y clasificación de objetos. Estos sistemas realizan estas tareas de manera automática, rápida y controlada. La visión artificial (1), una rama de la inteligencia artificial, comprende un conjunto de teorías, técnicas y métodos que permiten simular el proceso de visión biológica humana. La capacidad de extraer y analizar automáticamente la información de las imágenes obtenidas permite la creación de algoritmos y aplicaciones que interpretan el significado de una imagen.

En este trabajo, exploramos una pequeña parte del procesamiento de imágenes implementado en robots, enfocándonos en la resolución de la implementación de un seguidor de línea. Antes de abordar este problema específico, es crucial comprender que el procesamiento de imágenes es mucho más complejo de lo que podría parecer. Mientras que los humanos identificamos imágenes de forma intuitiva, hacer que una computadora procese e interprete la cantidad de información contenida en una imagen en un contexto real es una tarea altamente compleja.

El procesamiento de imágenes digitales (2) involucra varias etapas, desde la adquisición de la imagen hasta su procesamiento y análisis. Este proceso incluye la captura de imágenes, la mejora de su calidad mediante técnicas como la eliminación de ruido y el aumento del contraste, y la segmentación de la imagen en partes relevantes para su análisis posterior. Cada una de estas etapas es crucial para garantizar que la información extraída de la imagen sea precisa y útil para la aplicación deseada.

En la Figura 1 se muestran las etapas necesarias para realizar el procesamiento de imágenes. Primero se requiere de la etapa de adquisición de imágenes, donde un sensor produce imágenes que deben ser digitalizadas. Por ejemplo, utilizan la luz para la fotografía; rayos X para la radiografía, ultrasonido para la ecografía, etc. La siguiente etapa es el preprocesamiento, cuya finalidad es detectar y eliminar las fallas que puedan existir en la imagen para mejorarla. En este caso generalmente se mejora el contraste, se elimina el ruido, y por último se restaura la imagen. Posteriormente la siguiente etapa de segmentación, separa la imagen en partes necesarias de procesamiento que identifican la zona de interés para procesar; algunas de las técnicas que se ocupan en esta etapa son orientadas al píxel, los bordes, y las regiones. Sin embargo, las técnicas no son excluyentes sino que se combinan de acuerdo del tipo de aplicación.

Por otro lado, la siguiente etapa es la descripción o extracción de características, consiste en extraer características con alguna información cuantitativa de interés o que sean fundamentales para diferenciar una clase de objetos de otra. Para posteriormente terminar en la etapa de reconocimiento la cual asigna una etiqueta a un objeto basándose en la

información proporcionada por sus descriptores. La interpretación implica asignar significado a un conjunto de objetos conocidos, para finalmente llegar a la Base de Conocimiento, que almacena el dominio del problema para guiar la operación de cada módulo de procesamiento y controlar la interacción entre dichos módulos.

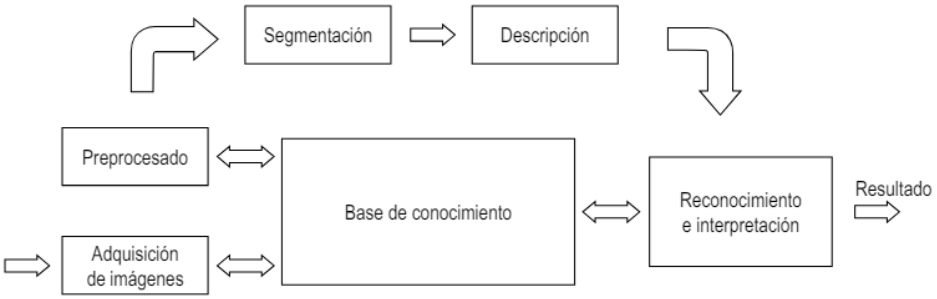

Figura 1. Etapas del procesamiento digital de imágenes.

Los sistemas de visión por computadora pueden clasificarse en tres niveles (3) según su complejidad e implementación: procesos de bajo nivel, que incluyen la captura y preprocesamiento de imágenes; procesos de nivel medio, que abarcan la segmentación y descripción de objetos; y procesos de nivel superior, que se centran en el reconocimiento e interpretación de conjuntos de objetos. Cada uno de estos niveles desempeña un papel fundamental en el desarrollo de aplicaciones de visión por computadora, como los vehículos autónomos.

El procesamiento de imágenes en sistemas embebidos (4) presenta retos únicos debido a las limitaciones de hardware y la necesidad de eficiencia en tiempo real. Las tarjetas embebidas, como las utilizadas en sistemas autónomos, deben realizar tareas complejas de procesamiento de imágenes con recursos limitados, lo que requiere una optimización cuidadosa de los algoritmos y la gestión de recursos.

Para el procesamiento de imágenes en una tarjeta embebida es necesario considerar varios aspectos críticos: La primera etapa, la adquisición de imágenes, utiliza sensores como cámaras CMOS o CCD que convierten la luz en señales eléctricas digitalizadas. Es fundamental seleccionar sensores de alta calidad y optimizar los controladores para garantizar una captura rápida y eficiente.

La interconexión entre la tarjeta NVIDIA Jetson y la cámara es crucial, con interfaces CSI o USB que facilitan la conexión. Para el mejor rendimiento, se recomiendan cámaras compatibles con CSI debido a su mayor ancho de banda y menor latencia. El preprocesamiento de imágenes, que incluye eliminación de ruido y ajuste de contraste, debe ser eficiente y no consumir muchos recursos. La segmentación, descripción, y extracción de características son etapas críticas que deben manejar variaciones ambientales y proporcionar datos precisos y rápidos. En este caso, la implementación utiliza la tarjeta NVIDIA y el sistema operativo Ubuntu, aprovechando bibliotecas optimizadas como OpenCV y

TensorFlow Lite, para desarrollar aplicaciones avanzadas de visión por computadora en tiempo real.

Para poder diseñar un sistema de procesamiento de imágenes es necesario conocer qué valores son los relevantes y cómo poder abstraernos con diferentes técnicas de procesamiento e identificación. Estas técnicas incluyen:

Detección de contornos y formas.

La detección de contornos y formas (5) es un proceso esencial en la visión por computadora que permite identificar los límites de los objetos en una imagen tal y como se observa en la figura 2, los contornos son curvas que unen todos los puntos continuos a lo largo de un límite con el mismo valor de intensidad. La detección de contornos ayuda a simplificar la representación de una imagen y facilita el análisis y la interpretación de las formas dentro de la imagen. Para implementar la detección de contornos y formas en OpenCV, se utilizan las siguientes funciones:

- **cv2.imread:** Para cargar la imagen.
- **cv2.GaussianBlur:** Para aplicar un filtro gaussiano y reducir el ruido.
- **cv2.Canny:** Para detectar bordes en la imagen.
- **cv2.findContours:** Para encontrar los contornos en la imagen de bordes.
- **cv2.drawContours:** Para dibujar los contornos encontrados en la imagen original.

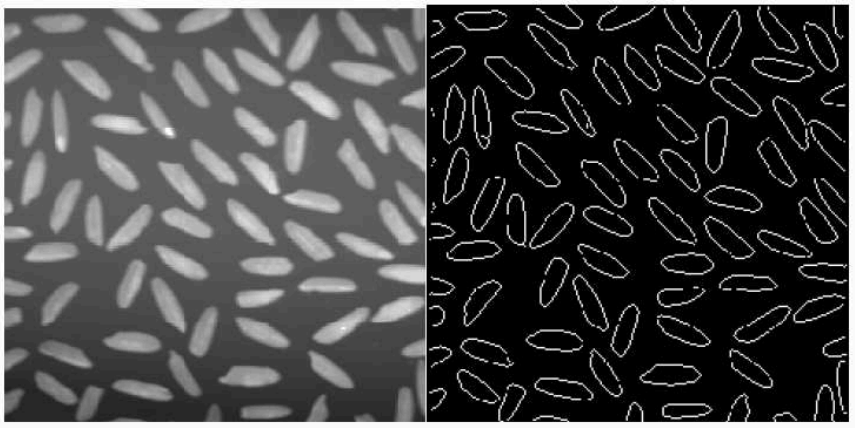

Figura 2. Ejemplo de detección de contornos.

Detección de líneas y bordes.

La detección de líneas y bordes (6) es fundamental para identificar las estructuras lineales y los límites de los objetos en una imagen. Como se observa en la figura 3, los bordes representan transiciones bruscas en la intensidad de la imagen y se detectan utilizando operadores de derivadas como Sobel, Prewitt o Canny. La detección de líneas a menudo se implementa utilizando la Transformada de Hough, que encuentra líneas en un espacio de

parámetros transformado. Para la detección de líneas y bordes en OpenCV, se utilizan las siguientes funciones:

- **cv2.imread:** Para cargar la imagen.
- **cv2.cvtColor:** Para convertir la imagen a escala de grises.
- **cv2.GaussianBlur:** Para aplicar un filtro gaussiano y reducir el ruido.
- **cv2.Canny**: Para detectar bordes en la imagen.
- **cv2.HoughLinesP:** Para detectar líneas en la imagen de bordes utilizando la Transformada de Hough.
- **cv2.line**: Para dibujar las líneas detectadas en la imagen original.

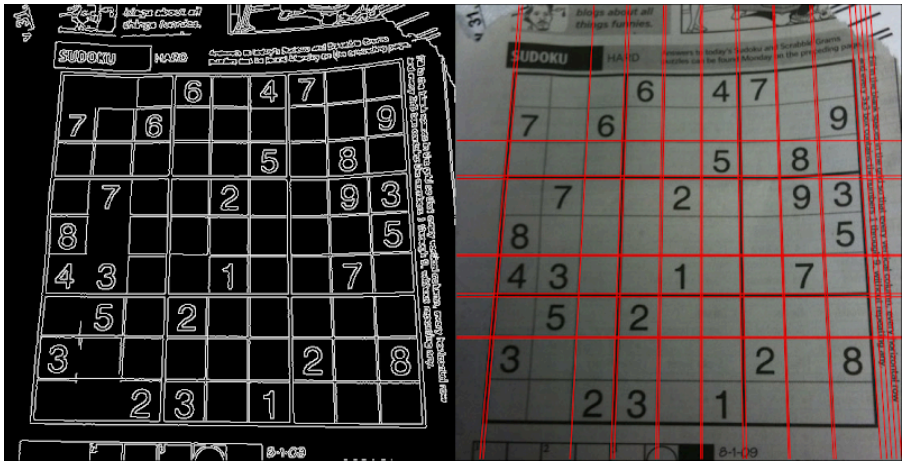

Figura 3. Ejemplo de detección de líneas usando la transformada de Hough.

Con estos procesamientos es posible crear algoritmos para el seguimiento de líneas y la planificación de trayectorias. Estos algoritmos permiten que un sistema autónomo, como un robot o un vehículo, siga un camino predefinido o ajuste su ruta en tiempo real según las condiciones del entorno. Por ejemplo, el seguimiento de líneas se puede implementar utilizando la detección de bordes para identificar el contorno de una carretera, y la planificación de trayectorias puede utilizar algoritmos de optimización para calcular la ruta más eficiente y segura. Estos algoritmos no solo mejoran la precisión del sistema, sino que también permiten la operación en entornos dinámicos y cambiantes.

Por otro lado, la robustez de un sistema de detección de imágenes se puede mejorar controlando la iluminación, agregando modelos de clasificación avanzados y procesando la mínima información necesaria de la imagen para ahorrar recursos. Controlar la iluminación puede incluir técnicas como la calibración de sensores y el uso de filtros para compensar las variaciones de luz. Los modelos de clasificación, como las redes neuronales profundas, pueden diferenciar entre diversos tipos de objetos con alta precisión. Además, optimizar el procesamiento de imágenes mediante la reducción de datos innecesarios y la implementación de algoritmos eficientes permite ahorrar recursos computacionales y mejorar el rendimiento general del sistema embebido.

SOLUCIÓN DEL PROBLEMA

Diseño de Pista

Antes de adentrarnos en la parte de codificación, es crucial explicar cómo se diseñó la pista, ya que esta juega un papel fundamental en la evaluación del sistema de seguimiento de líneas. Diseñamos la pista para que se asemejara lo más posible a la pista final, optando por utilizar madera MDF y cinta kinesiológica. La elección de estos materiales se debe a sus propiedades no reflectivas, lo que minimiza las interferencias en el procesamiento del seguidor de línea.

Para el diseño de la pista, utilizamos SolidWorks, creando una estructura acorde a las tres líneas de la pista, incluyendo los espaciados entre las líneas y los colores adecuados. Se diseñaron dos piezas principales: una de línea recta y otra de curva. Ambas piezas están equipadas con uniones tipo rompecabezas para facilitar su ensamble, permitiendo flexibilidad en la configuración de diferentes formas de pista.

La Figura 4 muestra ambas piezas diseñadas en SolidWorks, mientras que la Figura 5 representa el ensamblaje de una pista de carreras utilizando estas piezas, demostrando cómo pueden combinarse para formar diversas configuraciones de pista.

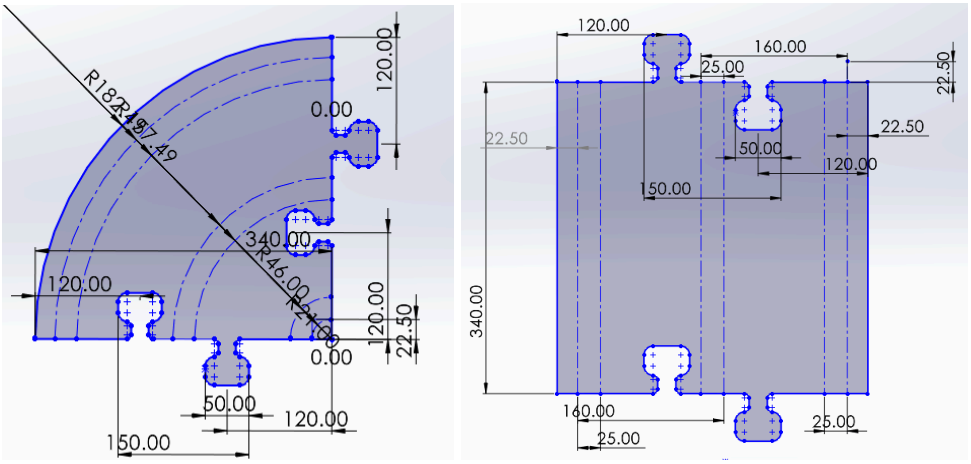

Figura 4. Diseño de piezas (recto y curvo).

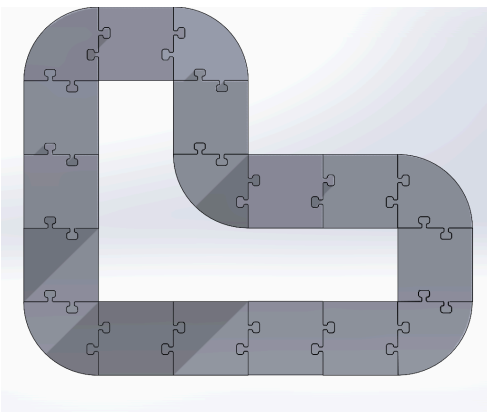

Figura 5. Ensamblaje de pista.

Comunicación ssh

A diferencia de la forma en que se ha estado trabajando hasta este momento con ros2, para este entregable se ha modificado la ejecución y transmisión de información hacia la jetson nano de nuestro Puzzlebot. Como se pudo apreciar, en la entrega anterior se presentaron diversos problemas que pusieron en juego el desempeño del algoritmo para procesar imágenes, siendo un proceso lento, transmisión de video con muy poca frecuencia y por ende, muchos errores de procesamiento. Por estas razones, nos dimos cuenta de que el problema era la forma en que el nodo de procesamiento de imágenes estaba arrancando, pues el trabajo pesado no estaba directamente realizado con la jetson, sino por la laptop del usuario y posteriormente siendo transmitido por ssh. Entonces el propósito de este entregable fue lanzar un nuevo nodo que corriera directamente sobre la jetson.

Se utilizará la conexión ssh de una forma distinta a la convencional, de forma en que pueda accederse al directorio de archivos de la jetson de manera remota. Para acceder a ella, dentro de la carpeta de archivos en Ubuntu se accederá a una conexión de servidor, donde se seleccionará un protocolo de transmisión de archivos a través de una hotspot (ssh). Para ello es necesario ya estar previamente conectado a la hotspot del Puzzlebot, como se ha realizado de costumbre. Seleccionando este protocolo, seguido del nombre de la red y su dirección IP, tendremos acceso a un entorno virtual de la jetson nano.

Figura 6. Ventana para conexión a la Jetson

Una vez conectado al servidor, se puede tener acceso remoto a todos los archivos almacenados localmente en la jetson nano. Por lo tanto, aquí es donde se creará el nuevo paquete de ros2, como ya se había trabajado anteriormente. Sin embargo, hay que cerciorarse de que se está trabajando a través de una terminal remota, para ello:

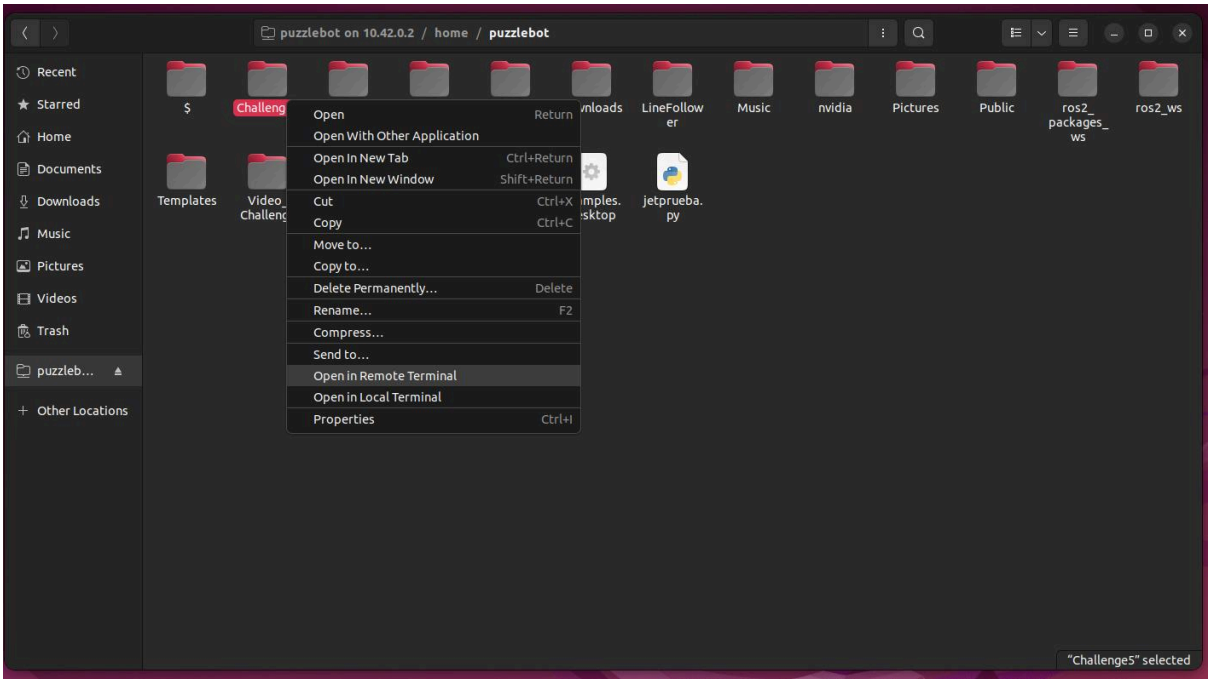

Figura 7. Ventana para abrir de manera remota la carpeta en al terminal

Una vez que se haya lanzado el nodo, será necesario ingresar a la carpeta de src hasta el código fuente donde se esté guardado el mismo. En esta ocasión, el programa se ejecuta y guarda remotamente dentro del editor de texto de Ubuntu.

Dado que el propósito es lanzar el nodo directamente en la jetson nano, tanto el *colcon build* como la inicialización del nodo deberán ser ejecutados en una terminal remota. Además, el topico de inicialización de la cámara también deberá ejecutarse de la misma manera. Debido a que el comando para desplegar la lista de tópicos puede ser fuera de la hotspot, para visualizar la imagen de la cámara en su respectivo tópico, el comando *ros2 run rqt_image_viewer rqt_image_viewer* si puede ser ejecutado de forma local.

Listado de algunos comandos necesarios para correr el programa:

- Lanzar nodos para inicializar la cámara del puzzlebot
  ros2 launch ros_deep_learning video_source.ros2.launch

- Graficar imagen procesada del puzzlebot
  ros2 run *rqt_image_viewer rqt_image_viewer*

- Habilitar el puerto serial
  ros2 run micro_ros_agent micro_ros_agente serial --dev /dev/ttyUSB0

Creación de código y funcionamiento del algoritmo

Tal y cómo se mencionó anteriormente, este código deberá ser ejecutado por la Jetson Nano debido a que de esta forma podemos lograr un mejor rendimiento en general del procesamiento de imágenes. La estructura del código es la siguiente:

1. Importamos las librerías correspondientes para realizar el procesamiento de imágenes:

```
import rclpy
from rclpy.node import Node
import numpy as np
import cv2
from cv_bridge import CvBridge
from sensor_msgs.msg import Image
from std_msgs.msg import String
from geometry_msgs.msg import Twist
```

2. Continuamos la declaración del nodo *'controller'*:

```
'''Realizamos la declaración del nodo'''
class Controller(Node):
   def _init_(self):
       super()._init_('controller')
```

a) Creamos una suscripción al tópico encargado de publicar la imágen capturada por la cámara, asociando dicha suscripción a la función self.camera_callback:

```
self.bridge = CvBridge()
self.subscription = self.create_subscription(Image, '/video_source/raw',
self.camera_callback, 10)
self.subscription
```

b) Creamos un publisher para publicar la imágen que hemos procesado. Esto nos servirá para depurar nuestro algoritmo de ser necesario:

```
self.new_image_processed = self.create_publisher(Image, '/image_processed',
10)
```

c) Creamos otro publisher para mandar la velocidad que nuestro robot debe ajustar para seguir la trayectoria:

```
self.speed_publisher = self.create_publisher(Twist, 'cmd_vel', 10)
```

De igual forma, creamos una variable para mandar la velocidad deseada:

```
self.speed_configuration = Twist()
```

d) Declaramos algunas variables que servirán para el funcionamiento general del algoritmo.

- Variables para el redimensionamiento de la imágen a procesar:

```
self.frame_height =  60
self.y = 120
self.x = 360
```

- Creamos variables para almacenar las matrices correspondientes a las imágenes:

```
self.frame = None
self.frame_gray = None
self.thers = None
self.blur = None
self.imagen_erosionada = None
```

- Declaramos variables “extra” que nos servirán para realizar el procesamiento de la imágen:

```
# Umbral para la binarización
self.umbral = 100

# Variables para calcular y almacenar el centroide
self.cX = 0
self.cY = 0

self.cX2 = 180
self.cY2 = 120

# Kernel
self.kernel = None

# La variable angle será útil para implementar el controlador
self.angle = None

# Constante del controlador proporcional
self.kp = 0.2

self.get_logger().info('Controller node succesfully initialized!!')
```

3. Continuamos con la declaración de la función encargada de “recibir” la imágen capturada por la cámara:

a) Copiamos la imágen a la variable de la clase *self.frame:*

```
def camera_callback(self, msg):
        try:
```

```
        self.frame = self.bridge.imgmsg_to_cv2(msg, "bgr8")
```

b) Ajustamos su tamaño de tal manera que podamos disminuir el trabajo de la Jetson haciendo más efectivo el algoritmo:

```
        self.frame = cv2.resize(self.frame, (self.x, self.y))
        self.frame = self.frame[0 : self.frame_height, :]
```

c) Procesamos la imágen:

```
            self.process_image()

    except Exception as e:
        #Mensaje de error en la recepción de la imágen
        self.get_logger().info(f'Failedto convert image to CV2:{e}')
```

4. Continuamos con el procesamiento de la imágen. Para este caso, optamos por realizar la rotación de la imágen debido a que la cámara está montada sobre el puzzlebot de manera invertida. Posterior a esto, realizamos la conversión del frame a escala de grises:

```
def process_image(self):
    self.frame = cv2.rotate(self.frame, cv2.ROTATE_180)
    self.frame_gray = cv2.cvtColor(self.frame, cv2.COLOR_RGB2GRAY)
```

5. Posteriormente a obtener la imágen en escala de grises, realizamos la binarización de la imágen usando el umbral previamente declarado.

```
    _,self.thers = cv2.threshold(self.frame_gray, self.umbral, 255, cv2.THRESH_BINARY)
    self.blur = cv2.medianBlur(self.thers, 5)
```

6. Una vez que hemos binarizado la imagen, procedemos a aplicar erosión para reducir el ruido en la imagen. Es importante mencionar que en este caso se aplica la operación de dilatación en lugar de la erosión, ya que los colores no se han invertido, lo que resulta en que la pista esté en color blanco. Para realizar este ajuste, se utiliza el siguiente código.

```
    self.kernel = np.ones((8,8), np.uint8)
    self.imagen_erosionada = cv2.dilate(self.blur, self.kernel, iterations=1)
    self.imagen_erosionada = cv2.bitwise_not(self.imagen_erosionada)
```

7. Una vez que se ha obtenido un fotograma del video ya preprocesado, se realiza un proceso adicional para determinar la velocidad que el robot debe seguir para continuar su trayectoria. Para esto, se realiza lo siguiente:

Se obtienen los contornos de la imagen obtenida por la cámara (línea de camino) para posteriormente obtener el centro de masa del objeto detectado en primer plano. Como referencia, se dibuja un punto con coordenadas fijas en el centro de la parte inferior del

fotograma. En cuanto al centroide, también se dibuja un punto y se almacenan sus coordenadas en variables.

```
        self.contours,_=cv2.findContours(self.imagen_erosionada,cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        cv2.circle(self.frame, (self.cX2, self.cY2), 2, (255, 0, 0), -1)

        for contour in self.contours:
            # Calcula el momento del contorno
            self.M = cv2.moments(contour)
            # Calcula las coordenadas del centro de masa
            if self.M["m00"] != 0:
                self.cX = int(self.M["m10"] / self.M["m00"])
                self.cY = int(self.M["m01"] / self.M["m00"])
            else:
                self.cX, self.cY = 0, 0
```

8. Una vez que se ha identificado el contorno del camino en el fotograma, se utiliza la función contourArea para calcular la superficie en píxeles que abarca el contorno detectado. Entonces, si el área detectada es mayor a 1800 pixeles cuadrados, significa que se trata del camino que debe seguir el robot, por lo que entonces se dibuja el centroide del objeto. Se dibujan los contornos del camino para mostrarlos como salida en la imagen.

```
if area > 1800:
                # Dibuja el centro de masa
                cv2.circle(self.frame, (self.cX, self.cY), 2, (0, 0, 0), -1)
                #se dibujan los bordes de los contornos detectados
                cv2.drawContours(self.frame, [contour], -1, (0,255,0),2)
```

9. Dentro del mismo condicional, y para determinar la velocidad necesaria, se calcula el ángulo generado por los dos puntos anteriormente comentados y se aplica un seno para obtener un intervalo de valores entre -1 a 1 para determinar si existe velocidad angular, se omite el caso en que exista una división entre 0.

```
try:
        self.angle = -1*np.degrees(np.arctan2(self.cY - self.cY2, self.cX -self.cX2))
        #Calculamos el seno ( normalizamos de -1 a 1)
        angle_radians = np.radians(self.angle)
        self.sine = 1-np.sin(angle_radians)
        print(self.sine)

except ZeroDivisionError:print('division by zero')
```

10. Se imprimen etiquetas para facilitar la visualización del puzzlebot, tal como:
    1. Etiqueta del ángulo calculado

```
        cv2.putText(self.frame, f"Theta: {self.sine:.2}", (self.cX - 50, self.cY - 10),
        cv2.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 1, cv2.LINE_AA)
```

2. Línea de indicador de curva

```
cv2.line(self.frame, (self.cX2, self.cY2), (self.cX, self.cY), (0, 255, 0), 2)
```

11. Se declaran las velocidades que tendrá el robot ante distintas circunstancias. La lineal se mantendrá siempre constante para tener mejor precisión, y la angular dependerá del valor del seno obtenido, multiplicada por un controlador *kp*.

```
self.speed_configuration.linear.x = 0.1
self.speed_configuration.angular.z = self.kp * self.sine*-1
```

Finalmente, se publican las velocidades en el tópico y la imagen original con las etiquetas.

```
self.speed_publisher.publish(self.speed_configuration)
self.new_image_processed.publish(self.bridge.cv2_to_imgmsg(self.frame, encoding="bgr8"))
```

RESULTADOS

Mostraremos a continuación nuestro video del funcionamiento donde pondremos a prueba la respuesta de el puzzlebot a la pista que diseñamos donde contamos con dos pruebas para poder observar su desempeño durante las trayectorias propuestas.

Link del video de funcionamiento: https://youtu.be/h295UVcr548

- **Prueba 1 - Cuadrado**

Al inicio nosotros decidimos optar por un diseño de pista cerrada para observar cómo actúan los giros del Puzzlebot en cada instante en que la pista tiene un giro, debido a que esta parte resultaría ser más difícil para ajustar la trayectoria. Como observamos en el video tenemos un buen desempeño de toda la trayectoria ajustándose bien en cada instante y también tomando en cuenta las curvas desde la perspectiva de la cámara, donde gracias a que los contornos de la imagen binarizada y el cálculo del área nos ayuda a poder filtrar los ruidos asegurando que tome en cuenta los contornos significativos para poder recorrer todo correctamente.

Observamos cómo se hace el rastreo de la pista con la cámara del Puzzlebot donde para cada contorno significativo tenemos el cálculo del centro de la masa, también calculamos el ángulo entre un punto fijo de la imagen y el centro de la masa, este ángulo nos ayuda para poder poder convertirlo primeramente a radianes y de esta manera calculamos el seno con una normalización de -1 a 1 y tenemos el valor del seno que se va a imprimir como se puede ver en el rastreo en cada iteración donde ajusta la velocidad angular asegurando que corrija el rumbo de manera proporcional al error detectado.

Figura 8. Rastreo de la pista en recta.

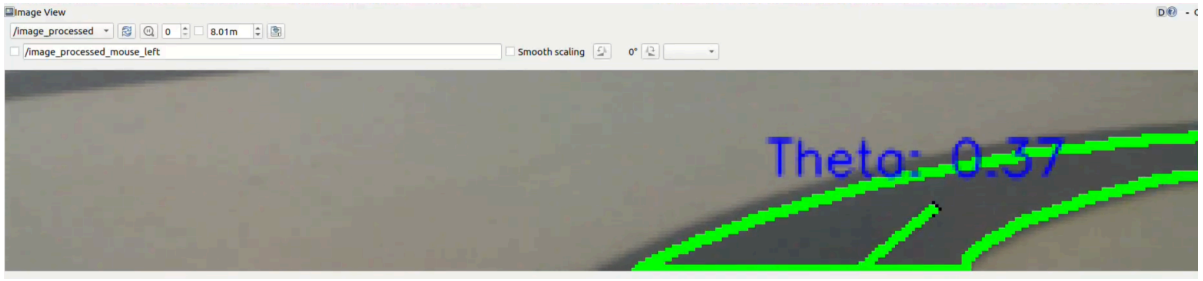

Figura 9. Rastreo de la pista en la curva.

- **Prueba 2 - Rectángulo**

Para la segunda prueba hicimos un recorrido recortando una parte para poder formar un rectángulo de tal manera que observamos en que las curvas están una tras de otra para los extremos del rectángulo por lo que el Puzzlebot tendrá que tomarlas de manera continua y podemos observar que se está teniendo una respuesta rápida donde hace el cambio de cada curva de la mejor manera sin inconsistencias durante la trayectoria. Con ayuda también del cálculo del seno, que recordemos refleja de manera proporcional la desviación del robot con respecto a la línea, el seno será cercano a 0 cuando se encuentre de manera recta lo que no necesita de una corrección significativa de la trayectoria.

Pero de otra manera en las curvas el valor del seno llega a ser cercano a 1 por lo que se necesita de una ajuste más fuerte para que no tenga una trayectoria incompleta de la curva, esto mejora debido a que se contempla una corrección suave para la transición de los cambios del robot y no los genere bruscamente haciendo que se tenga un movimiento más estable y un seguimiento de la línea más eficiente donde se busque una corrección.

Es importante recordar que para el ajuste de la velocidad angular para los giros tendremos en cuenta el control proporcional para ajustar el valor, donde determina cuanto debe de girar el robot, multiplicando por -1 para asegurar que gire en la dirección correcta

```
#la velocidad angular depende del valor del seno del ángulo, usando controlador kp
self.speed_configuration.angular.z = self.kp * self.sine*-1
```

CONCLUSIONES

Este trabajo permitió incrementar el grado de complejidad en el desarrollo de un robot autónomo al utilizar únicamente procesamiento de imagen en lugar de sensores seguidores de línea. A través de este enfoque, se exploraron y aplicaron técnicas avanzadas de visión por computadora, lo que enriqueció significativamente el proyecto.

Podemos determinar que se cumplió con el objetivo principal, que era implementar un sistema de seguimiento de líneas para el Puzzlebot utilizando procesamiento de imágenes con OpenCV y ROS. Además, se lograron los objetivos secundarios relacionados con el diseño de la pista, la aplicación de técnicas de detección de imagen, y la integración de estas en el entorno de ROS. Los algoritmos desarrollados permitieron al robot seguir las líneas de la pista con precisión en un entorno controlado.

Sin embargo, no se cumplieron completamente todos los objetivos, específicamente en lo relacionado con la robustez del sistema bajo diferentes condiciones de iluminación. Las pruebas se realizaron en un espacio cerrado con iluminación constante, lo que limitó la evaluación del sistema en escenarios más variados. Esto dejó varias áreas de mejora, como la adaptación a cambios de iluminación y la capacidad de manejar trayectorias más complejas. Para mejorar la metodología implementada, se podrían considerar las siguientes acciones:

- **Ajuste para trayectorias más cerradas o curvas pronunciadas:** Refinar los algoritmos de detección para que el robot pueda manejar mejor las curvas cerradas y trayectorias complejas.
- **Mejorar la binarización:** Implementar técnicas avanzadas de binarización que funcionen de manera efectiva bajo diferentes condiciones de iluminación, asegurando un desempeño consistente.
- **Uso de líneas de guía:** Incorporar líneas de guía en el diseño de la pista para mejorar la precisión del algoritmo de seguimiento de líneas.
- **Alcanzar velocidades más altas:** Optimizar el procesamiento de imágenes y la etapa de decisión para que el robot pueda operar a velocidades más altas sin perder precisión.
- **Incluir filtros para eliminar ruido de fondo:** Aplicar filtros adicionales para reducir el ruido de fondo en las imágenes, mejorando la fiabilidad del sistema de detección de líneas.

Estas mejoras no solo incrementarían la robustez y eficiencia del sistema, sino que también ampliarían su aplicabilidad en diversos entornos y condiciones, avanzando hacia un robot autónomo más versátil y capaz.

REFERENCIAS

1. Viera Maza, G. I. (2017). Procesamiento de imágenes usando OpenCV aplicado en Raspberry Pi para la clasificación del cacao.

2. Palomino, N. L. S., & Concha, U. N. R. (2009). Técnicas de segmentación en procesamiento digital de imágenes. *Revista de investigación de Sistemas e Informática*, *6*(2), 9-16.
3. Sucar, L. E., & Gómez, G. (2011). Visión computacional. *Instituto Nacional de Astrofísica, Óptica y Electrónica. México*.
4. Arreguín, C. A. R. (2010). Desarrollo e implementación de algoritmos de procesamiento de imágenes en dispositivos programables en campo FPGA.
5. Rebaza, J. V. (2007). Detección de bordes mediante el algoritmo de Canny. *Escuela Académico Profesional di Informática. Universidad Nacional de Trujillo*, *4*.
6. Tutor de programación. (2017) . Detección de líneas y círculos usando la transformada de Hough con OpenCV. https://acodigo.blogspot.com/2017/09/deteccion-de-lineas-y-circulos-usando.html