Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Implementación de robótica inteligente (Gpo 501)

Reporte: Reto semanal 4 Manchester Robotics

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

Fecha de entrega
Jueves 2 de Abril de 2024

RESUMEN

Se desarrolló un código en ROS para el robot Puzzlebot, permitiendo su manejo en diversas trayectorias y ajustando la velocidad según el color del semáforo. Para ello se emplearon diferentes términos como la odometría para conocer la posición del robot en todo momento, la navegación punto a punto con cálculo en tiempo real del error de posición, junto con un controlador en lazo cerrado P para minimizar dicho error y lograr que el robot alcance la posición deseada. Además se demostró el uso de sistemas de visión artificial en la robótica móvil para detectar formas y colores a través de una cámara integrada.

OBJETIVOS

El objetivo principal es desarrollar un sistema de procesamiento de imágenes en ROS usando OpenCV de Python para ajustar la velocidad del Puzzlebot en función de los colores del semáforo durante su trayectoria. Dentro de los objetivos secundarios se encuentran los siguientes:

1. Implementar un algoritmo que permita conocer las coordenadas de la posición del robot en todo momento mediante odometría.
2. Introducir un sistema de navegación punto a punto que permita al robot moverse hacia una posición deseada y calcular el error de posición en tiempo real.
3. Diseñar un controlador en lazo cerrado P para minimizar el error de posición durante la navegación.
4. Garantizar que el robot alcance la posición deseada de manera eficiente.
5. Demostrar el comportamiento de sistemas de visión artificial en la robótica móvil utilizando la cámara para la detección de formas y colores.

INTRODUCCIÓN

Los robots móviles están diseñados para moverse en entornos dinámicos y complejos, interactuando en diferentes escenarios y adaptándose en tiempo real. El estudio respecto a la navegación de estos robots va enfocado a la autonomía donde el principal objetivo es permitirles moverse de manera eficiente y segura hacia sus destinos, tomando decisiones autónomas basadas en la información que reciben de su entorno. Es por ello que para comprender cómo estos robots navegan se requiere de diferentes conocimientos, entre ellos el control.

Para un robot móvil, el control de lazo cerrado[1] es fundamental. Este tipo de control le permite al robot ajustar continuamente su actuación en respuesta a las señales de retroalimentación que recibe del entorno. Al mantener un control cercano sobre sus acciones, el robot puede corregir errores y seguir una trayectoria predeterminada con mayor precisión.

En este contexto, la implementación de un controlador proporcional (P) es particularmente relevante. Este tipo de controlador ajusta la salida del sistema en función del error proporcional entre la referencia y la salida actual, brindando una respuesta rápida y estable ante las perturbaciones del entorno. Esto ayuda al robot a mantenerse en curso hacia su destino deseado con la mínima desviación posible.

Sin embargo, a pesar de sus ventajas, la implementación de controladores P puede enfrentar problemas derivados del cálculo del error. Si el error no se calcula correctamente o si hay retrasos en la retroalimentación del sistema, el robot puede experimentar oscilaciones indeseadas o dificultades para alcanzar su objetivo con precisión. Por lo tanto, es crucial diseñar y ajustar cuidadosamente los parámetros del controlador P para garantizar un rendimiento óptimo del robot móvil en su navegación autónoma.

Retomando lo anterior junto con la figura 1, en este trabajo la salida es la velocidad del robot móvil, mientras que el control se determina con los errores de posición y orientación, los cuales son parte de su odometría. Es decir, el sistema de control utiliza la información de la odometría para calcular el error entre la posición y orientación deseada del robot y su posición y orientación actual. Este error se utiliza entonces para ajustar la velocidad del robot y corregir su trayectoria en tiempo real.

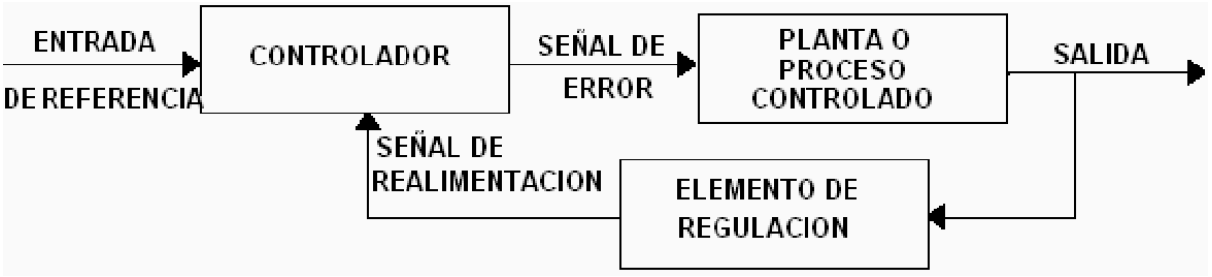

Figura 1. Sistema de control de lazo cerrado.

Al utilizar la odometría[2] como base para el cálculo de errores, se obtiene una retroalimentación precisa y confiable del estado del robot en su entorno. Esto permite que el control de la velocidad sea sensible a los cambios en la posición y orientación del robot, lo que a su vez mejora la capacidad del robot para seguir trayectorias específicas y alcanzar sus objetivos de manera eficiente.

Para que el funcionamiento del robot pueda ser más preciso ante diversos escenarios, es necesario considerar tener una mayor robustez en el controlador que permita funcionar de manera efectiva y ajustar dinámicamente su comportamiento según las condiciones del entorno. Esto implica no solo diseñar un controlador capaz de manejar errores de posición y orientación, sino también anticipar y adaptarse a posibles perturbaciones externas, como cambios inesperados en la velocidad del viento o la presencia de obstáculos móviles. En este caso, la robustez de nuestro sistema surge a partir de la experimentación de configurar la variable P y realizando diferentes pruebas en diversos escenarios.

Por otro lado, para que la navegación autónoma del robot se ajuste ante diversas situaciones reales, se consideró la implementación de un sistema embebido de procesamiento de imágenes interconectado con la Jetson y la cámara. Utilizando la librería de OpenCV, que funciona de manera eficaz, se logra detectar contornos de diversos tipos y formas, así como también reconocer colores. Esto permite que nuestro robot comprenda cómo ajustar la velocidad ante una

señal de semáforo, por ejemplo, detectando los colores específicos y actuando en consecuencia. Este enfoque de procesamiento de imágenes ofrece una herramienta valiosa para mejorar la capacidad de adaptación y respuesta del robot ante diferentes condiciones ambientales y desafíos de navegación.

En este sentido, para poder agregar robustez a nuestro programa es necesario implementar varias estrategias que fortalezcan la capacidad de nuestro sistema embebido de procesamiento de imágenes. Algunas de estas estrategias incluyen[3]:

- **Mejora en la detección de contornos y colores:** Refinar los algoritmos de detección de contornos y colores para que sean más precisos y robustos ante diversas condiciones de iluminación y fondo. Esto puede implicar ajustar los umbrales de segmentación, utilizar técnicas de filtrado de ruido y mejorar el manejo de oclusiones.

- **Validación y seguimiento de objetos:** Implementar técnicas de validación y seguimiento de objetos para verificar la consistencia y la continuidad de los contornos detectados a lo largo del tiempo. Esto ayudará a reducir los errores de detección y a mejorar la estabilidad del sistema frente a cambios abruptos en el entorno.

- **Optimización de rendimiento:** Optimizar el rendimiento del sistema para garantizar tiempos de respuesta rápidos y eficientes. Esto puede implicar el uso de técnicas de paralelización y optimización de código para aprovechar al máximo los recursos de hardware disponibles.

- **Manejo de situaciones adversas:** Implementar mecanismos de recuperación y tolerancia a fallos para manejar situaciones adversas, como la pérdida momentánea de señal de la cámara o la presencia de objetos inesperados en la escena. Esto garantizará que el sistema pueda mantener su funcionalidad incluso en condiciones imprevistas.

- **Pruebas exhaustivas:** Realizar pruebas exhaustivas en una variedad de escenarios y condiciones ambientales para validar la robustez y el rendimiento del sistema. Esto ayudará a identificar y corregir posibles problemas antes de desplegar el sistema en un entorno real.

Al implementar estas estrategias, podemos mejorar significativamente la robustez de nuestro sistema de procesamiento de imágenes, lo que a su vez mejorará la capacidad de adaptación y respuesta del robot ante diferentes condiciones ambientales y desafíos de navegación.

SOLUCIÓN DEL PROBLEMA

Nodo Odometry

Como en actividades previas, para controlar la trayectoria y la posición del robot es crucial conocer su ubicación en todo momento. Por lo tanto, es imprescindible volver a utilizar el nodo de odometría que hemos desarrollado anteriormente. Recordemos brevemente que las ecuaciones de odometría se derivan del modelo cinemático del robot (ver Figura 2), el cual incluye ciertos elementos clave que nos permiten calcular la posición del robot en todo momento:

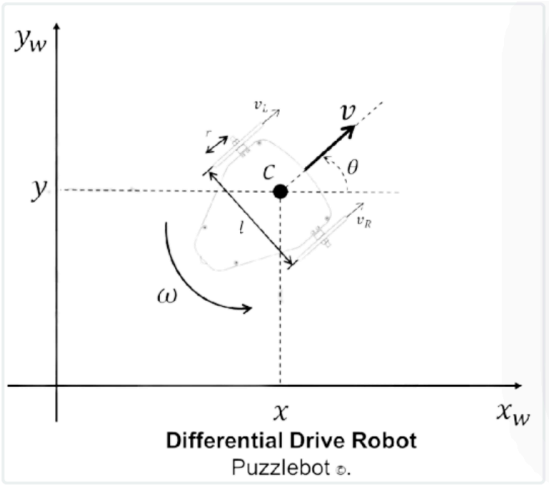

Figura 2. Diagrama del modelo cinemático del Puzzlebot

A través de un análisis detallado, podemos desarrollar ciertas fórmulas que serán la base de nuestra implementación de odometría en el código:

$$d = v \cdot dt = r\left(\frac{\omega_r + \omega_l}{2}\right) \cdot dt \quad [1]$$

$$\theta = \omega \cdot dt = r\left(\frac{\omega_r - \omega_l}{l}\right) \cdot dt \quad [2]$$

Una vez que hemos definido claramente lo que necesitamos implementar en el código, procederemos a programar en ROS2. Comenzaremos creando el paquete correspondiente en el que trabajaremos:

```
ros2 pkg create --build-type ament_python --license Apache-2.0 closed_loop -node-name odometry --dependencies std_msgs geometry_msgs
```

Para poder determinar el funcionamiento del nodo realizamos diferentes pasos los cuales son:

1. Importamos las librerías necesarias para el manejo del nodo:

```
import rclpy
import numpy as np
from rclpy.node import Node
from geometry_msgs.msg import Pose2D
import rclpy.qos
from std_msgs.msg import Float32, Bool
```

2. Inicializamos el nodo bajo el nombre ‘odometry_node’ y nos suscribimos a los tópicos ‘VelocityEncL’ and ‘VelocityEncR’ para obtener la velocidad de las llantas:

```
class My_publisher(Node):
   def __init__(self):
       super().__init__('odometry_node')
       self.left_speed_subscriber = self.create_subscription(Float32,
       'VelocityEncL', self.read_left, rclpy.qos.qos_profile_sensor_data)
       self.right_speed_subscriber = self.create_subscription(Float32,
       'VelocityEncR', self.read_right, rclpy.qos.qos_profile_sensor_data)
```

3. Procedemos a crear un publisher para enviar datos de tipo Pose2D al tópico '/Position' a una frecuencia de 100 Hz

```
       self.position_publisher = self.create_publisher(Pose2D, 'Position', 10)
       self.timer_period = 0.01
       self.timer = self.create_timer(self.timer_period,self.position_callback)
```

4. Definimos variables importantes como la distancia entre ejes y el radio de las ruedas:

```
       self.l = 0.19
       self.r = 0.05
```

5. Declaramos otras variables para realizar los cálculos de odometría:

```
      self.right = Float32()
      self.left = Float32()

      self.xp = 0.0
      self.yp = 0.0
      self.thetap = 0.0

      self.x =  0.0
      self.y = 0.0
      self.theta = 0.0

      self.position = Pose2D()

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

6. Asociamos las suscripciones de velocidad con funciones de callback para copiar la información recibida a variables locales:

```
    def read_left(self, msg):
        self.left = msg

    def read_right(self, msg):
        self.right = msg
```

7. Explicamos la función position_callback, donde calculamos las velocidades lineales y angulares:

```
def position_callback(self):
        self.xp = self.r*((self.right.data+self.left.data)/2)*np.cos(self.theta)
        self.yp = self.r*((self.right.data+self.left.data)/2)*np.sin(self.theta)
        self.thetap = self.r*(self.right.data - self.left.data)/self.l
```

8. Realizamos la integración para obtener las coordenadas de posición:

```
        self.x += self.xp * self.timer_period
        self.y += self.yp * self.timer_period
        self.theta += self.thetap * self.timer_period
```

9. Copiamos los datos a la variable tipo Pose2D y los publicamos en el tópico '/Position'::

```
         self.position._x = self.x
        self.position._y = self.y
        self.position._theta = self.theta*180/np.pi
        self.position_publisher.publish(self.position)
```

10. Finalmente, creamos el nodo dentro de nuestro main:

```
def main(args=None):
    rclpy.init(args=args)
    m_p = My_publisher()
    rclpy.spin(m_p)
    m_p.destroy_node()
    rclpy.shutdown()


if __name__ == '__main__':
    main()
```

Nodo Error

El nodo error calcula y publica errores de posición y ángulo basados en la diferencia entre la posición actual del robot y una serie de puntos de referencia predefinidos, para ello seguimos los siguientes pasos:

1. Importamos las librerías necesarias de ROS2 y los tipos de mensajes que utilizaremos.

```
import rclpy
import numpy as np
from rclpy.node import Node
from geometry_msgs.msg import Pose2D, Point
import rclpy.qos
from std_msgs.msg import Float32, Bool, Empty
from srv_int.msg import Error2D
```

2. Creamos una clase ‘My_publisher’ que hereda de Node de ROS2 y la inicializamos con el nombre 'error_node'.

```
class My_publisher(Node):
   def __init__(self):
       super().__init__('error_node') #Inicializamos el nodo
```

3. Creamos una subscripción al tópico /Position para obtener la posición actual del robot y definimos un publicador para enviar mensajes de error al tópico 'error'.

```
self.position_subscriber = self.create_subscription(Pose2D, '/Position',
self.calculate_error, rclpy.qos.qos_profile_sensor_data)
self.error_publisher = self.create_publisher(Error2D, 'error', 10)
```

4. Definimos los puntos del trazado de trayectoria donde necesitamos generar el trazado de la pista, para ello tenemos que definir una trayectoria concisa que seguirá el puzzlebot de principio a fin, con el fin de poder demostrar correctamente como funciona el procesamiento de los colores decidimos tener una trayectoria completamente lineal, al tener que dibujarlo en un plano cartesiano como se observa en la figura 3 para poder tomar la coordenada hacia el punto de la trayectoria; gracias a ello podemos construir una lista donde definimos esta coordenada (x,y) para el punto final al que debe ir el puzzlebot.

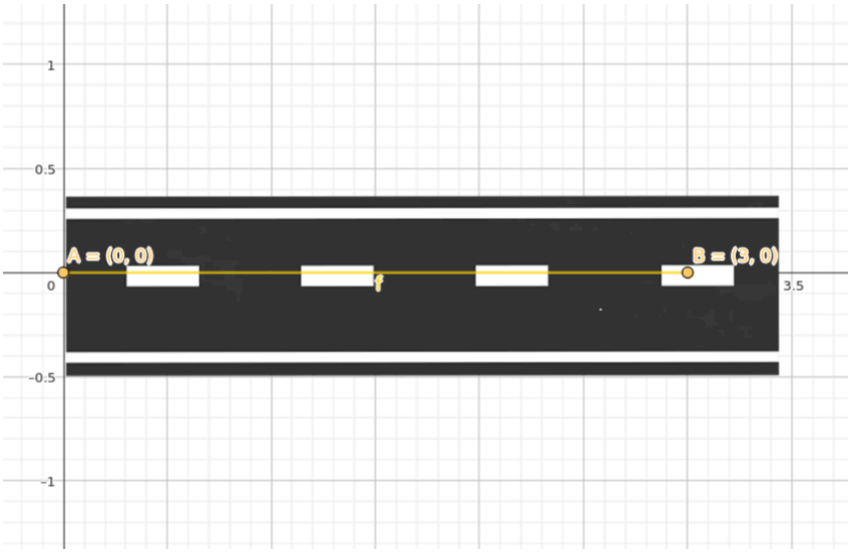

Figura 3 . Trazado de las coordenadas para la trayectoria

```
self.selection = [[3.0,0.0]]
```

De la misma manera necesitamos tomar en cuenta una posición fija del semáforo durante toda la trayectoria, por lo que ajustamos el semáforo en la posición que se observa, el recorrido se hará en línea recta para probar los diferentes escenarios que puede enfrentar en respuesta al cambio del semáforo.

5. Luego iniciamos y configuramos varias variables necesarias para calcular los errores de posición y ángulo.

```
self.error = Error2D()
self.setpoint = Float32()
self.move = True

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

6. Creamos una función con el nombre ‘calculate_error’ quese llama cada vez que se recibe un mensaje de posición del robot. Calcula el error de posición y ángulo entre la posición actual del robot y el punto de referencia más cercano, y publica el error.

Para este caso, se sabe que las siguientes fórmulas sirven para calcular el error de posición y de ángulo entre la posición del robot y un setpoint definido:

$$[3] \quad e_0 = \theta_T - \theta_R = atan2(x_T, y_T) - \theta_R$$

$$[4] \quad e_d = \sqrt{(x_T - x_R)^2 + (y_T - y_R)^2}$$

```
    def calculate_error(self,msg):
        if self.move:
            #Calculamos la distancia al punto
            x_reference = self.selection[self.index_point][0]
            y_reference = self.selection[self.index_point][1]
            x_distance = x_reference- msg._x
            y_distance = y_reference- msg._y
            distance = np.sqrt(x_distance**2 + y_distance**2)

            #Calculamos el error del ángulo:
            if(self.index_point>0):
                x_reference = x_reference - self.selection[self.index_point-1][0]
                y_reference = y_reference - self.selection[self.index_point-1][1]
            agnle =np.arctan2(y_reference, x_reference)
            angle_degrees = np.degrees(agnle)
                  self.get_logger().info(f"X:  {x_reference},  Y:  {y_reference},  index:
{self.index_point}")
            angle_degrees = angle_degrees + 360 if angle_degrees < 0 else angle_degrees
            angle_error = angle_degrees - msg._theta
```

```
                #Copiamos a la variable del error
                self.error.distance_error = distance
                self.error.theta_error = angle_error

                if (distance <= 0.05 and angle_error <=1.0):
                    self.index_point +=1
                    if self.index_point >= len(self.selection):
                        self.move = False
                        self.error.distance_error = 0.0
                        self.error.theta_error = 0.0
                self.error_publisher.publish(self.error)
```

Nodo color_detector

Este nodo se encargará de realizar todo el procesamiento de imágenes para el puzzlebot. A través de un algortimo de detección colores, el programa sefa capaz de indicar si el semáforo se encuentra en estado verde, amarillo o rojo, para posteriormente piblicar un valor en su tópico e indicar la velocidad correspondiente que debe tener el robot, ya sea para avanzar hacia su camino o detenerse.

El programa inicia con la importación de la librería ya mencionada de OpenCV, la cual ya incluye todos los métodos necesarios para implementar procesamientos de imágenes sofisticados y optimizados computacionalmente. Además se incluye un nuevo tipo de mensaje que se encarga de realizar lecturas de la cámara integrada al puzzlebot.

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

Inicializando el nodo ‘color_detector’, se realiza la suscripción al tópico encargado de leer la camara ‘`/video_source/raw`’ camara. Como publicador, se crea uno para transmitir la imagen ya procesada por el robot, siendo la detección de los distintos colores.

```
class ColorId(Node):
   def __init__(self):
       super().__init__('traffic_light')
       self.bridge = CvBridge()
       dt = 0.1
                self.subscription   =   self.create_subscription(Image,   '/video_source/raw',
self.camera_callback, 10)
       self.subscription
       self.timer = self.create_timer(dt, self.timer_callback)
         self.light_detection_pub = self.create_publisher(Image, '/image_processing/color_id',
10)
```

A continuación, se muestran las variables de la clase que indican el rango de valores en formato hsv para detectar la mascara de colores (rojo, verde y amarillo). Nótese que se incluye, 2 rangos únicamente para el valor de los rojos, ya que se encuentran posicionados en ambos extremos de la escala dentro de este formato. También se agregan las dimensiones de la imagen a desplegar. Finalmente se indica un mensaje de éxito tras inicializar el nodo.

```
        #Definimos los valores para la máscara de colores
        self.lower_red = np.array([0, 100, 20], np.uint8)
        self.upper_red = np.array([5, 255, 255], np.uint8)

        self.lower_red2 =np.array([175, 100, 20], np.uint8)
        self.upper_red2 = np.array([179, 255, 255], np.uint8)

        self.lower_green = np.array([40, 100, 0], np.uint8)
        self.upper_green = np.array([80, 255, 255], np.uint8)

        self.lower_yellow = np.array([15, 100, 20], np.uint8)
        self.upper_yellow = np.array([45, 255, 255], np.uint8)

        #Dimensiones de la imágen
        self.frame_width = 320
        self.frame_height = 180

        #Variables para la imagen
        self.frame = None

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

Posteriormente se encuentra el la función draw_circles, la cual consiste básicamente en dibujar círculos alrededor de los contornos detectados sobre una máscara en la imagen. Una vez que se ha detectado el contorno, se dibuja el círculo en el centro del contorno para iluminar el color detectado.

```
def draw_circles(self, mask, color, label):
        # Encuentra los contornos en la máscara
        countours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
        for c in countours:
            area = cv2.contourArea(c)
            if area > 300:
                M = cv2.moments(c)
                if M["m00"] == 0:
                    M["m00"] = 1
                x = int(M["m10"] / M["m00"])
                y = int(M['m01'] / M['m00'])
                newCountour = cv2.convexHull(c)
                cv2.circle(self.frame, (x, y), 7, (128, 0, 255), -1)
                cv2.drawContours(self.frame, [newCountour], 0, color, 3)
                # Etiqueta de color
                  cv2.putText(self.frame, label, (x - 50, y - 50), cv2.FONT_HERSHEY_SIMPLEX, 1,
color, 2)
```

La siguiente función se encarga de realizar las lecturas de la cámara y almacenar cada fotograma en una variable. Se reajusta el tamaño de acuerdo a la resolución indicada al inicio del código.

```
  def camera_callback(self, msg):
       try:
           self.frame = self.bridge.imgmsg_to_cv2(msg, "bgr8")
           #Reajustamos el tamaño
           self.frame = cv2.resize(self.frame, (self.frame_width, self.frame_height))
           self.frame = self.adjust_gamma(0.5)
       except:
           self.get_logger().info('Failed to convert image to CV2')
```

Estas funciones ayudan a generar una imagen más clara del objetivo de la imagen. Se trata de un procesamiento adicional de video obtenido, siendo un ajuste de gamma y un proceso de erosión. Un valor bajo aplicado de gamma resultará en una imagen oscurecida, lo cual sirve para enfocarse únicamente en objetos luminosos, tal como lo será el semáforo. Por otro lado, la erosión ayudará a reducir la detección de posibles iluminaciones en el fondo de la imagen, que crean ruido y pueden alterar el resultado.

```
   def adjust_gamma(self, gamma = 1.0):
       inv_gamma = 1.0 / gamma
             table = np.array([((i / 255.0) ** inv_gamma) * 255 for i in np.arange(0,
256)]).astype("uint8")
       return cv2.LUT(self.frame.copy(), table)

   def erode(self,mask):
       kernel = np.ones((6, 6), np.uint8)
       return cv2.erode(mask, kernel, iterations=1)
```

Finalmente, la función *timer_callback* se encarga de repetir el proceso para cada fotograma dentro del video. Este proceso consiste en convertir la imagen al dicho formato HSV para posteriormente aplicar la mascara de color correspondiente. Con esta máscara se obtiene el color deseado y simultáneamente se aplica el proceso de erosión. Al color detectado, se le agrega una etiqueta la cual indica su color. El programa termina publicando cada fotograma procesado al tópico `image_processing/color_id'.`

```
   def timer_callback(self):

       if self.frame is not None:

           #Convertimos el frame a escala de grises
           gray_frame = cv2.cvtColor(self.frame, cv2.COLOR_BGR2GRAY)

           frameHSV = cv2.cvtColor(self.frame, cv2.COLOR_BGR2HSV)

           # Máscaras para cada color
           maskAzul = cv2.inRange(frameHSV, self.lower_green,self.upper_green)  # Verde
```

```
                maskAmarillo = cv2.inRange(frameHSV, self.lower_yellow, self.upper_yellow)   #
Amarillo
            maskRed1 = cv2.inRange(frameHSV, self.lower_red, self.upper_red)  # Rojo (parte 1)
              maskRed2 = cv2.inRange(frameHSV, self.lower_red2, self.upper_red2)  # Rojo (parte
2)
            maskRed = cv2.add(maskRed1, maskRed2)  # Combinar máscaras de rojo

            erosiongreen = self.erode(maskAzul)
            erosionamarillod = self.erode(maskAmarillo)
            erosionred = self.erode(maskRed)
            # Llama a la función para dibujar los contornos para cada color
            self.draw_circles(erosiongreen, (0, 255, 0), 'Verde')  # Verde
            self.draw_circles(erosionamarillod, (0, 255, 255), 'Amarillo')  # Amarillo
            self.draw_circles(erosionred, (0, 0, 255), 'Rojo')  # Rojo

                        self.light_detection_pub.publish(self.bridge.cv2_to_imgmsg(self.frame,
encoding="bgr8"))
```

RESULTADOS

Video del funcionamiento:
https://drive.google.com/file/d/1ZBG-LoK1gBuQT9IPLLq3PkINYT1vCIop/view?usp=sharing
https://youtu.be/Tpn5DZGeUF4

Al principio comenzamos con probar nuestro método en nuestra cámara de nuestras laptops para la detección de los colores del semáforo y podemos decir que funcionaba de una forma correcta, con la definición de los parámetros para los rangos de cada color teniendo una gran ajuste para cada parte; debemos de indicar que para el color rojo se tienen dos declaraciones debido a que está en los dos extremos del modelo del color HSV.

```
# Definición de rangos de colores
verdeBajo = np.array([40, 100, 0], np.uint8)
verdeAlto = np.array([80, 255, 255], np.uint8)

amarilloBajo = np.array([15, 100, 20], np.uint8)
amarilloAlto = np.array([45, 255, 255], np.uint8)

redBajo1 = np.array([0, 100, 20], np.uint8)
redAlto1 = np.array([5, 255, 255], np.uint8)

redBajo2 = np.array([175, 100, 20], np.uint8)
redAlto2 = np.array([179, 255, 255], np.uint8)
```

A continuación como podemos observar de la figura 4 a la figura 6 vemos la detección desde nuestras cámaras de las laptop, donde se ve que se logró hacer un gran método y también con la

ayuda de la función de findContours, circle y drawContours de opencv construimos un método para que la figura también la pudiera detectar que en este caso es un círculo.

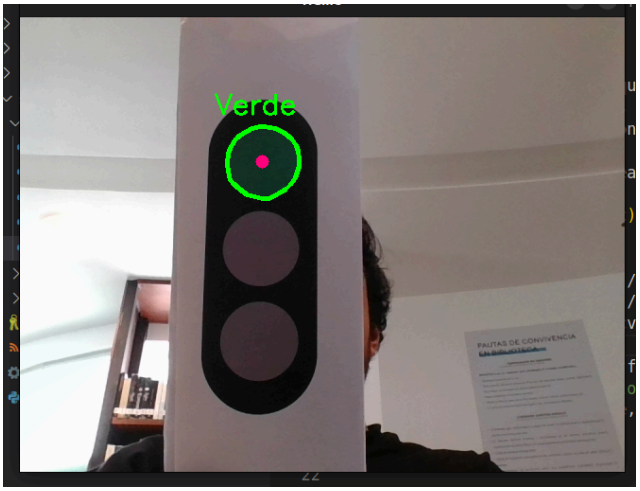

Figura 4. Procesamiento de detección de color verde

Figura 5. Procesamiento de detección de color amarillo

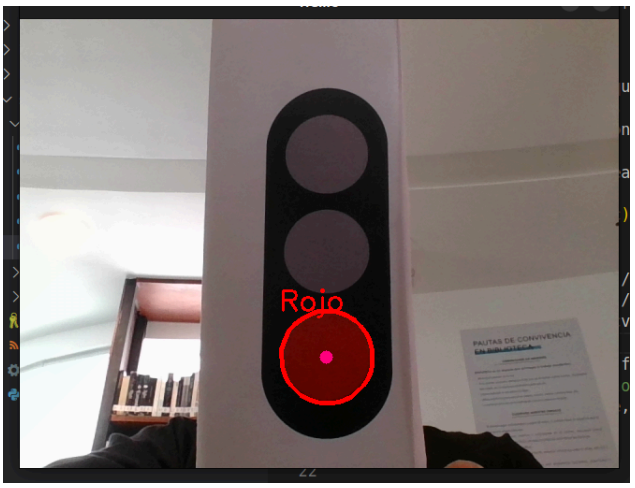

Figura 4. Procesamiento de detección de color rojo

Como vemos a lo largo del video podemos decir que el puzzlebot sigue de buena forma la trayectoria que planteamos, que si bien es una trayectoria simple, facilita la implementación del detector de semáforo. No obstante, tuvimos algunas complicaciones para poder detectar correctamente de manera rápida el color debido a tanto la calidad de la cámara del puzzlebot como el procesamiento que tiene la cámara ya que tiene un pequeño delay y no actúa de forma rápida el procesamiento de los colores para el semáforo. A pesar de tener ese inconveniente podemos ver que se hace una buena detección de los colores conforme vamos poniendo el semáforo y de esta manera logramos tener un buen procesamiento de lo que se va presentando

CONCLUSIONES

Durante el desarrollo tuvimos una buena respuesta al procesamiento de imágenes al poder tomar en cuenta un adecuado ajuste para nuestros valores de la máscara de cada color, los objetivos se concluyeron de forma exitosa al lograr generar un método adecuado con ayuda de Opencv para poder procesar cada uno de los colores del semáforo; además de tener el sistema de navegación punto a punto con la posición deseada en todo momento. La implementación para el sistema de procesamiento nos permitió poder ajustar la velocidad del robot en función a los colores del semáforo durante la trayectoria, el diseño del controlador en lazo cerrado P fue nuevamente exitoso al poder minimizar el error de posición durante la navegación.

Es importante mencionar que enfrentamos múltiples desafíos al principio debido a que no teníamos un sistema tan robusto para poder hacer la detección de los colores del semáforo, además de que la calibración de la cámara y los cambios en el brillo también dificultan el poder procesar la imagen continuamente, posteriormente pudimos tener una experimentación que nos ayudó a mejorar la detección pero seguíamos teniendo un sistema medianamente ajustado. Aunque se tuvo un nivel de robustez considerable, deberíamos mejorar el sistema para que fuera mucho más eficiente y conciso ante condiciones externas que afecten la visión de la cámara.

Como mejora podríamos implementar una evaluación más precisa de los algoritmos de detección de colores, ya que aunque tuvimos un buen ajuste podríamos mejorar los parámetros pero sobre todo componer un método más robusto que pueda tomar en cuenta la segmentación de colores y la umbralización para eliminar el ruido del fondo y ser más preciso en solo tomar en cuenta la detección del objeto deseado, en este caso los colores del semáforo. Se podrían integrar estrategias que tuvieran la implementación de sensores para una detección mejorada del semáforo e incluso un algoritmo de planificación para una experimentación más certera del procesamiento de imágenes.

Creemos que el algoritmo necesita implementar muchas mejoras y nos comprometemos a trabajar para lograr un algoritmo super eficiente que nos pueda ayudar a mejorar este mini-reto.

REFERENCIAS

1. Perez M, A. (2007). *Introducción a los sistemas de control y modelo matemático para sistemas lineales invariantes en el tiempo*. Universidad Nacional de San Juan. http://dea.unsj.edu.ar/control1/apuntes/unidad1y2.pdf
2. Toapanta, J. M., Chancusi, B. Z., Cadena, L. O., & Nieto, G. C. (2014). *Diseño y construcción de un robot móvil que permita la obtención de una nube de puntos del escaneo de habitaciones utilizando láser y webcams.* Ingenius, (11), 53-61.
3. Solano, G. (2019). *Detección de colores y Tracking en OpenCV*. https://omes-va.com/deteccion-de-colores2/#google_vignette