Reto semanal 4. Manchester Robotics

Implementación de Robótica Inteligente

Integrantes:

A017314050 | Eleazar Olivas Gaspar

A01735696 | Azul Nahomi Machorro Arreola

A01732584 | Angel Estrada Centeno

A01735692 | Arick Morelos del Campo

Profesores:

Rigoberto Cerino Jimenez

Alfredo Garcia Suarez

Juan Manuel Ahuactzin Larios

Cesar Torres Huitzil

Abril 2024

Resumen:

En este reto, nos sumergimos aún más en la interacción con el Puzzlebot, explorando en detalle sus capacidades de localización y movimiento en tiempo real. En lugar de simplemente enviar comandos de movimiento al robot, nos centraremos en comprender su posición actual y cómo podemos guiarlo hacia una posición deseada con la mayor precisión posible. Para lograr esto, comenzaremos asignándole al Puzzlebot la tarea de determinar su posición actual y calcular el error de posición en comparación con una posición objetivo definida por el usuario. Este proceso implica el uso de conceptos de localización, como la odometría, para obtener una comprensión precisa de la posición y el movimiento del robot en su entorno.

Una vez que hemos establecido la posición actual y la posición deseada, pasamos a diseñar un controlador proporcional (P) que nos permita minimizar el error de posición y dirigir al Puzzlebot hacia su objetivo. Este controlador ajustará continuamente la velocidad y la dirección del robot en función de la magnitud y la dirección del error de posición, utilizando ROS como plataforma de control para enviar los comandos necesarios al Puzzlebot.

Además de la implementación del controlador proporcional, este desafío integrará conceptos previamente aprendidos en el manejo de robots móviles, incluida la navegación punto a punto y el control en lazo cerrado. Estos conceptos serán fundamentales para el diseño y la implementación efectiva del controlador, asegurando que el Puzzlebot pueda moverse de manera precisa y eficiente hacia su objetivo.

Por último, exploramos el uso de sistemas de visión artificial como parte de este desafío. Nos centramos específicamente en la detección de formas y colores utilizando una cámara integrada en el Puzzlebot. Esta capacidad adicional permitirá al robot percibir y responder a su entorno de manera más sofisticada, abriendo nuevas oportunidades para la interacción y la aplicación en una variedad de escenarios.

Objetivos:

El principal objetivo es mejorar las habilidades del diseño e implementación de controladores para robots móviles, específicamente utilizando un controlador. También se busca aplicar y reforzar los conceptos previamente aprendidos en localización a través de la odometría y navegación punto a punto, utilizando estos conocimientos para calcular la posición actual del robot y guiarlo hacia una posición deseada de manera precisa.

Otro objetivo es introducirnos en el uso de sistemas de visión artificial, específicamente para la detección de formas y colores utilizando una cámara integrada en el Puzzlebot. Como último objetivo tenemos que al implementar un controlador proporcional y aplicar conceptos de control en lazo cerrado, el objetivo es mejorar la capacidad del Puzzlebot para moverse con mayor precisión y eficiencia hacia su objetivo, minimizando el error de posición en el proceso.

Introducción:
Para abordar la solución de este reto tenemos que tener en cuenta algunos conceptos teóricos importantes que explican el funcionamiento, a continuación los conceptos a conocer:

- Control en lazo cerrado para un robot móvil:

El control en lazo cerrado implica la medición continua de la posición y la orientación del robot móvil. Esto se logra mediante una variedad de sensores, que pueden incluir encoders en las ruedas para medir el desplazamiento lineal, giroscopios para detectar cambios en la orientación, acelerómetros para medir la aceleración, y cámaras u otros sensores de visión para la detección del entorno. Una vez que se obtienen estas mediciones, se comparan con la trayectoria deseada o la posición objetivo. Esta trayectoria puede ser predefinida o calculada en tiempo real, dependiendo de los requisitos del sistema. La diferencia entre la posición deseada y la posición real se conoce como error, y es la base para tomar decisiones de control.

Con el error calculado, se genera una señal de control adecuada para corregir la desviación del robot móvil de su trayectoria deseada. Esta señal de control se envía a los actuadores del robot, que pueden ser motores en las ruedas, servomotores para dirigir, o cualquier otro dispositivo que permita controlar el movimiento del robot. La retroalimentación continua permite al sistema realizar ajustes en tiempo real a medida que el robot se mueve. Esto significa que cualquier desviación de la trayectoria deseada se corrige de inmediato, lo que resulta en un movimiento preciso y robusto del robot.

Para implementar este proceso, se utilizan diversos algoritmos de control, como el control proporcional-integral-derivativo (PID), controladores de retroalimentación lineal cuadrática (LQR), controladores de trayectoria, entre otros. Estos algoritmos determinan cómo se calcula la señal de control en función del error y otros parámetros del sistema.El control en lazo cerrado también puede adaptarse a condiciones cambiantes del entorno, como superficies resbaladizas, obstáculos inesperados o cambios en la carga del robot. Los algoritmos de control pueden ajustar dinámicamente los parámetros para mantener el rendimiento deseado incluso en situaciones imprevistas.

- Cálculo del error:

Error de posición: se calcula como la diferencia vectorial entre la posición deseada y la posición actual del robot.

$$\mathbf{e}_{pos} = \mathbf{p}_{deseado} - \mathbf{p}_{real}$$

Fig. 1. Fórmula vectorial del error

Error de orientación: El error de orientación puede ser calculado como la diferencia entre la orientación deseada y la orientación actual. A menudo, este cálculo involucra operaciones con ángulos y puede requerir el uso de funciones trigonométricas.

$$\mathbf{e}_{ori} = \text{atan2}(\sin(\theta_{deseado} - \theta_{real}), \cos(\theta_{deseado} - \theta_{real}))$$

Fig. 2. Fórmula angular del error

Los errores calculados pueden estar sujetos a ruido y errores inherentes a los sensores utilizados. La filtración de la señal y las técnicas de suavizado pueden ser necesarias para obtener estimaciones más precisas del estado actual del robot. El cálculo del error debe realizarse a una frecuencia suficiente para permitir una respuesta en tiempo real a las dinámicas del entorno y del propio robot. Esto es crítico para mantener un comportamiento estable y eficiente. Calcular el error en el control de un robot móvil es un paso crítico que influye directamente en la efectividad del sistema de control. Este cálculo no sólo facilita la corrección y la adaptación en la trayectoria del robot sino que también ayuda a optimizar el desempeño general del sistema a través de una retroalimentación precisa y continuamente ajustada.

- Robustez en el controlador:

La robustez en los controladores de robots móviles es crucial para mantener un rendimiento estable y predecible en condiciones variables. Esto implica la capacidad de manejar perturbaciones externas y adaptarse a variaciones en el entorno y el sistema. Los controladores robustos se diseñan con márgenes de estabilidad, técnicas adaptativas y predictivas, y estrategias de control específicas. La evaluación de la robustez implica pruebas físicas y análisis de sensibilidad. En la práctica, los robots móviles robustos pueden adaptarse a superficies irregulares y condiciones climáticas cambiantes. El futuro de la robustez incluye la integración de aprendizaje automático e inteligencia artificial para mejorar aún más la capacidad de adaptación y respuesta del controlador.

Los controladores adaptativos ajustan sus parámetros en tiempo real basándose en los cambios observados en el comportamiento del sistema. Los controladores predictivos, por su parte, utilizan modelos para anticipar futuras condiciones del sistema y ajustar las acciones de control de manera proactiva. Las pruebas de robustez pueden involucrar simulaciones que modelan variaciones en las condiciones del entorno y el sistema. Pruebas físicas en entornos controlados o naturalmente desafiantes también son cruciales para validar la robustez del controlador. Un robot móvil puede encontrarse con variaciones en la tracción debido a diferentes tipos de superficies. Un controlador robusto puede ajustar la velocidad y la fuerza de los motores para mantener el control y la estabilidad.

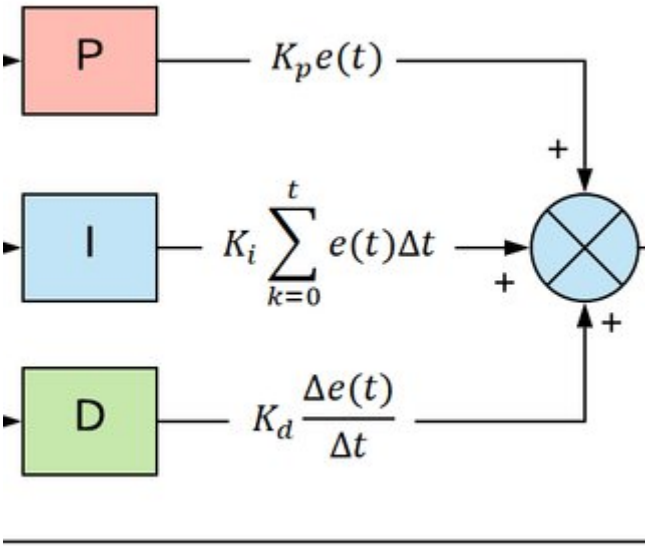

Fig. 3. variables del controlador

- Procesamiento de imágenes en una tarjeta embebida:

El procesamiento de imágenes en una tarjeta embebida implica realizar operaciones de análisis y manipulación de imágenes utilizando hardware integrado en la propia tarjeta, como una GPU o un procesador de señal digital (DSP). Estos componentes proporcionan la capacidad de realizar cálculos complejos de manera eficiente y en tiempo real, lo que es crucial para aplicaciones que requieren un alto rendimiento computacional, como la visión por computadora y el reconocimiento de patrones.

Con este hardware integrado, se pueden llevar a cabo una variedad de operaciones de procesamiento de imágenes, como filtrado y mejora de imágenes, detección de objetos, reconocimiento facial y seguimiento de objetos. Estas tareas pueden realizarse sin necesidad de una conexión externa a una computadora, lo que aumenta la portabilidad y la autonomía de los dispositivos embebidos.

La capacidad de procesamiento de imágenes en la tarjeta embebida permite lograr tiempos de respuesta rápidos y reducir la latencia asociada con el envío de datos a través de una conexión externa. Esto es especialmente importante para aplicaciones en tiempo real, como la robótica y la vigilancia, donde la velocidad y la capacidad de respuesta son críticas.

- Interconexión entre Jetson y la camara:

Jetson es una plataforma de computación de alto rendimiento desarrollada por NVIDIA, diseñada específicamente para aplicaciones de inteligencia artificial y visión por computadora en entornos embebidos. La interconexión entre Jetson y una cámara implica establecer una

conexión física y de comunicación entre la plataforma Jetson y la cámara para permitir la captura de imágenes y el procesamiento de imágenes en tiempo real.

Para lograr esta interconexión, es común utilizar interfaces estándar como USB, Ethernet, o conexiones de cámara específicas como MIPI CSI (Interfaz de Sensor de Imagen de Píxeles Móviles). Estas conexiones permiten la transferencia rápida y eficiente de datos de imagen desde la cámara a la plataforma Jetson.

Una vez establecida la conexión, la plataforma Jetson puede utilizar su potente hardware de procesamiento, que incluye GPU y CPU de alto rendimiento, para realizar tareas de procesamiento de imágenes en tiempo real. Esto puede incluir operaciones como detección de objetos, reconocimiento facial, seguimiento de objetos y más.

La combinación de Jetson y una cámara permite el desarrollo de una amplia gama de aplicaciones de visión artificial en entornos embebidos. Por ejemplo, en el caso de robots autónomos, esta configuración permite al robot percibir su entorno y tomar decisiones en tiempo real en función de la información visual capturada. Del mismo modo, en sistemas de vigilancia inteligente, la plataforma Jetson puede procesar imágenes de la cámara para detectar actividades sospechosas o eventos de interés.

- Detección de contornos:

La detección de contornos o formas en una imagen es un proceso fundamental en el análisis de imágenes y la visión por computadora. Consiste en identificar y delinear los bordes o límites de los objetos presentes en una imagen. Esto se puede lograr mediante diversas técnicas de procesamiento de imágenes, siendo una de las más comunes la detección de bordes. Para llevar a cabo la detección de bordes, se aplican operadores de gradiente que calculan gradientes de intensidad en la imagen para identificar cambios abruptos de intensidad que sugieren la presencia de bordes.

Además, se pueden utilizar filtros de derivadas segundas para resaltar los cambios de intensidad más fuertes y métodos de umbralización para identificar los píxeles que pertenecen a bordes y eliminar el ruido. Una vez identificados los píxeles de borde, se aplican algoritmos para conectar estos píxeles y formar contornos continuos alrededor de los objetos en la imagen.

- Detección de colores:

La detección de colores es un proceso clave en el análisis de imágenes que se enfoca en identificar y localizar regiones de una imagen que contienen un color específico o una gama de colores. Este proceso es fundamental en una variedad de aplicaciones, desde la robótica y la visión por computadora hasta la automatización industrial y la fotografía digital. Para llevar a cabo la detección de colores, primero se analizan los valores de los píxeles en el espacio de color de la imagen. Esto puede implicar convertir la imagen de su espacio de color

original (por ejemplo, RGB) a otro espacio de color más adecuado para la detección de colores, como el espacio de color HSV (tono, saturación, valor).

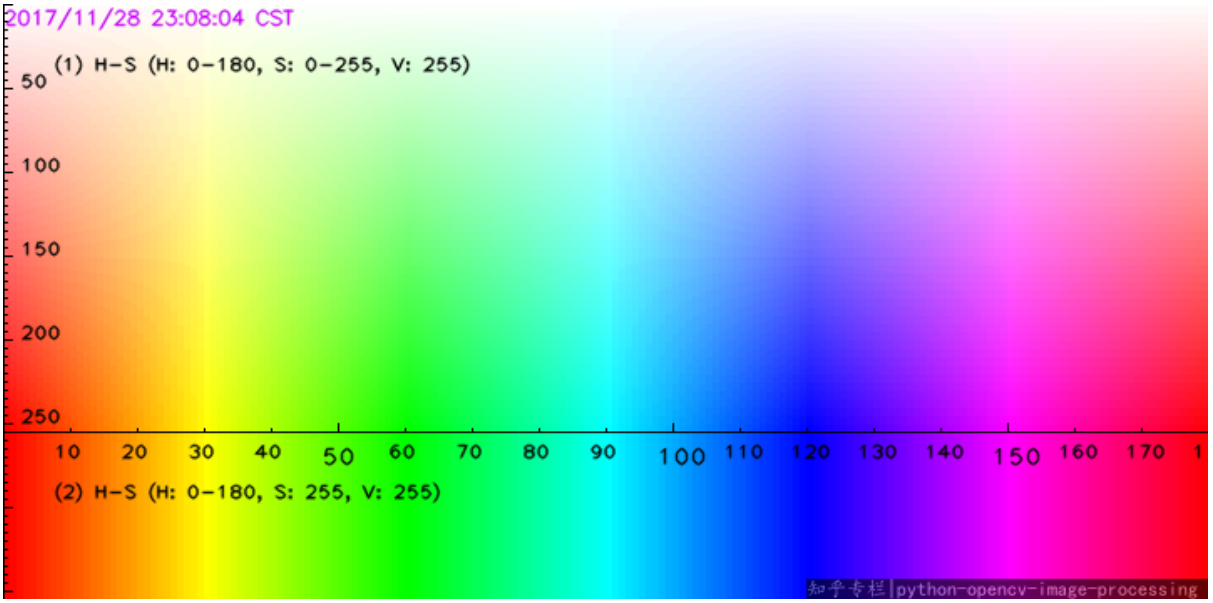

Fig. 4. Rangos de valores para HSV

Una vez que la imagen está en el espacio de color deseado, se aplican técnicas de segmentación de imágenes para separar las regiones de interés, es decir, aquellas que contienen el color específico o la gama de colores que se desea detectar. Esto puede lograrse utilizando métodos de umbralización, donde se establecen umbrales para determinar qué píxeles pertenecen a la región de interés y cuáles no.

Además, se pueden aplicar técnicas avanzadas de procesamiento de imágenes, como la morfología matemática, para refinar y mejorar la precisión de la detección de colores. Esto puede implicar operaciones como la erosión, la dilatación, la apertura y el cierre para eliminar el ruido y conectar regiones de interés. Las aplicaciones de la detección de colores son diversas y van desde la clasificación de objetos en entornos robóticos hasta la selección automática de elementos en aplicaciones de edición de imágenes. Por ejemplo, en robótica, la detección de colores puede utilizarse para identificar y clasificar objetos en un entorno industrial, mientras que en aplicaciones de fotografía digital, puede emplearse para seleccionar áreas específicas de una imagen para aplicar efectos o correcciones.

- Robustez de sistemas de procesamiento de imágenes:

La robustez en sistemas de procesamiento de imágenes asegura un rendimiento estable y preciso incluso en condiciones cambiantes como variaciones de iluminación o presencia de ruido. Se logra mediante técnicas de normalización de iluminación, filtrado de imágenes, transformaciones geométricas y aprendizaje automático. La realimentación continua y la validación ayudan a ajustar el sistema dinámicamente. En resumen, estas estrategias permiten

adaptarse a diversas condiciones de imagen para mantener un alto nivel de precisión en el procesamiento de imágenes.

Solución del problema:

Nodo Semáforo para la visión por computadora: Este nodo se encarga del procesamiento de imágenes adquiridas a través de la cámara del robot. Debido a la extensión del archivo sólo se mostrarán las partes más importantes del código, el código completo se podrá visualizar en la carpeta completa del paquete ros2.

Función para hacer la visión por computadora:

```python
    # Funcion callback para ejecucion de la vision por computadora
    def timer_callback(self):
        # Primero se comprueba que haya alguna imagen leida
        try:
            if self.valid_img:
                hsv = cv2.cvtColor(self.img, cv2.COLOR_BGR2HSV)

                # Limites de los colores en HSV
                lower_red1 = (0, 100, 100)
                upper_red1 = (10, 255, 255)
                lower_red2 = (170, 100, 100)
                upper_red2 = (180, 255, 255)

                lower_green = (40, 50, 50)
                upper_green = (80, 255, 255)

                lower_yellow = (20, 100, 100)
                upper_yellow = (35, 255, 255)

                # Umbrales para la deteccion de colores
                mask_red1 = cv2.inRange(hsv, lower_red1, upper_red1)
                mask_red2 = cv2.inRange(hsv, lower_red2, upper_red2)
                mask_green = cv2.inRange(hsv, lower_green, upper_green)
                mask_yellow = cv2.inRange(hsv, lower_yellow, upper_yellow)

                # Mascaras para la deteccion de colores
                mask_combined = cv2.bitwise_or(mask_red1, mask_red2)
                mask_combined = cv2.bitwise_or(mask_combined, mask_green)
                mask_combined = cv2.bitwise_or(mask_combined, mask_yellow)

                # Operacion morfologica para reducir ruido de la imagen recibida
                kernel = np.ones((5, 5), np.uint8)
                mask_combined = cv2.morphologyEx(mask_combined, cv2.MORPH_OPEN, kernel)

                # Deteccion de objetos circulares con la forma del kernel
                circle_kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (2*self.radius_cm, 2*self.radius_cm))
                mask_combined = cv2.dilate(mask_combined, circle_kernel)
```

Fig.5.

En primer lugar se declaran los rangos de valores para los colores de interés en formato HSV, que para este caso son el rojo, verde y amarillo, los colores del semáforo. Posteriormente se crean máscaras para detectar dichos colores en la imagen; también se aplica una reducción de ruido en la imagen y una forma estructurante de tipo disco, esto permite descartar cualquier objeto que no sea un círculo de la imagen y así hacer más robusta la detección únicamente de semáforos.

```
78
79              # Deteccion de bordes de los objetos en la imagen
80              contours, _ = cv2.findContours(mask_combined, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
81
82              # Deteccion del objeto mas grande en la imagen
83              max_area_red = 0
84              max_contour_red = None
85              max_area_green = 0
86              max_contour_green = None
87              max_area_yellow = 0
88              max_contour_yellow = None
89
90              for contour in contours:
91                  area = cv2.contourArea(contour)
92
93                  if area > max_area_red and self.is_color_contour(hsv, contour, lower_red1, upper_red1, lower_red2, upper_red2):
94                      max_area_red = area
95                      max_contour_red = contour
96
97                  if area > max_area_green and self.is_color_contour(hsv, contour, lower_green, upper_green):
98                      max_area_green = area
99                      max_contour_green = contour
100
101                 if area > max_area_yellow and self.is_color_contour(hsv, contour, lower_yellow, upper_yellow):
102                     max_area_yellow = area
103                     max_contour_yellow = contour
104
```

Fig.6.

A cada objeto en la imagen que coincida con la forma y color establecido se le determinan los bordes y se toma en cuenta sólamente el más grande. Esto nos permite descartar en el caso de haber muchos objetos muy pequeños regados por toda la imagen que no necesariamente sean luces de semáforos, así como sólo reconocer toda una luz del semáforo y no muchas partes más pequeñas que formen parte de la misma.

```
105             # Impresion del color del objeto detectado y publicacion al topico
106             if max_area_red > max_area_green and max_area_red > max_area_yellow:
107                 self.get_logger().info("Color detected: Red")
108                 self.color = 0.0
109             elif max_area_green > max_area_red and max_area_green > max_area_yellow:
110                 self.get_logger().info("Color detected: Green")
111                 self.color = 1.0
112             elif max_area_yellow > max_area_red and max_area_yellow > max_area_green:
113                 self.get_logger().info("Color detected: Yellow")
114                 self.color = 0.5
115             else:
116                 self.get_logger().info("No color detected")
117                 # Si se detecta el color rojo mantener al robot detenido hasta hallar luz verde
118                 if self.last_color == "Red":
119                     self.color = 0.0
120
121             # Publicar la proporcion de velocidad acorde al color
122             color_msg = Float32()
123             color_msg.data = self.color
124             self.pubv.publish(color_msg)
125
126             # Actualizar ultimo color detectado
127             if max_area_red > max_area_green and max_area_red > max_area_yellow:
128                 self.last_color = "Red"
129             elif max_area_green > max_area_red and max_area_green > max_area_yellow:
130                 self.last_color = "Green"
131             elif max_area_yellow > max_area_red and max_area_yellow > max_area_green:
132                 self.last_color = "Yellow"
133
134         # Si no hay imagen mandar mensaje de error para no matar al nodo y esperar alguna imagen
135         except Exception as e:
136             self.get_logger().info(f'Failed to process image: {str(e)}')
```

Fig.7.

Por último, se detecta el área dominante en la imagen sobre todas las demás y se imprime un mensaje por la terminal con el color detectado. También se publica el valor correspondiente

al color detectado en el tópico /color, esto con la finalidad de controlar la velocidad del robot acorde al semáforo: verde mantiene su velocidad, amarillo la reduce un 50%, rojo lo detiene completamente. Esta técnica nos ha resultado sumamente satisfactoria para adecuar el comportamiento del robot acorde a su entorno, pues permite hacer avanzar al robot o detenerlo por completo sin interferir en lo absoluto con los demás nodos.

Nodo controller: Este nodo es el encargado de calcular las velocidades lineales y angulares , dependiendo las variables que recibe de los nodos odometría, point generator y camarita, este nodo recibe la posición en “x”, “y” y “theta” para después calcular el error_w y error_v con respecto al punto objetivo que recibe del nodo point_generator, una vez que contamos con el error se aplica una kp, y limitadores de velocidad para que no sobrepase los niveles que puede soportar el puzzlebot, posteriormente tomamos la variable que nos manda el tópico color (1.0 para verde, .5 para amarillo y 0.0 para rojo) y se multiplica por la velocidad resultante.

```
        #verificamos que el robot todavia no ha llegado al objetivo
        if error_w >= 0.05 or error_w <= -0.05:
            #verificamos que los valores enten dentro de -pi a pi
            if error_w >= np.pi:
                error_w -= 2*np.pi
            elif error_w <= -np.pi:
                error_w += np.pi
            #Aplicamos la KP_w
            ang_control_output = self.kp_ang * error_w
            #Aplicamos un limite minimo para que no nos de valores con los que el bot no se puede mover
            if ang_control_output < 0.1 and ang_control_output > 0.0:
                ang_control_output = 0.1
            if ang_control_output > -0.1 and ang_control_output < 0.0:
                ang_control_output = -0.1
            #le mandamos una velocidad lineal de 0.0 hasta que su error angular sea 0.0
            lin_control_output = 0.0
        #si el error_w es 0.0
        else:
            #verificamos que el robot todavia no ha llegado al objetivo lineal
            if error_v >= 0.05 or error_v <= -0.5:
                #Aplicamos la KP_v
                lin_control_output = self.kp_lin * error_v
                #le mandamos una velocidad angular de 0.0 hasta que su error linear sea 0.0
                ang_control_output = 0.0
            #si el error_v es 0.0
            else:
                #le damos una velocidad linear y angular de 0.0
                lin_control_output = 0.0
                ang_control_output = 0.0
```

Fig.8. Cálculo de la velocidad angular y lineal.

```
#le damos limites a las velocidades
if ang_control_output > 0.25:
    ang_control_output = 0.25

if ang_control_output < -0.25:
    ang_control_output = -0.25

if lin_control_output > 0.25:
    lin_control_output = 0.25

if lin_control_output < -0.25:
    lin_control_output = -0.25

#multiplicamos por el valor que nos mando la camara dependiendo el color que detecta
ang_control_output = ang_control_output * self.color
lin_control_output = lin_control_output * self.color

#mandamos las velocidades a cmd_vel
cmd_vel.angular.z = ang_control_output
cmd_vel.linear.x = lin_control_output
self.publisher.publish(cmd_vel)

#mensajes de depuracion
self.get_logger().info(f'V: {lin_control_output}')
self.get_logger().info(f'W: {ang_control_output}')
self.get_logger().info(f'error_V: {error_v}')
self.get_logger().info(f'error_W: {error_w}')
```

Fig.9. Aplicamos limitadores para las velocidades y multiplicamos por el valor mandado por color.

Resultados:

Repositorio GitHub:
https://github.com/AzulMachorro/Retos_Manchester_Robotics_IRI-Week-4

Video demostrativo: https://youtu.be/PyXJFXkMHsw

Conclusiones:

En conclusión, este reporte resalta la importancia crucial de la localización precisa del robot para calcular con exactitud el error de posición respecto a un objetivo establecido. La utilización de técnicas como la odometría proporciona una comprensión detallada de la posición y el movimiento del robot en su entorno, lo cual es fundamental para su funcionamiento efectivo.

Además, se ha evidenciado la eficacia del controlador proporcional (P) para minimizar el error de posición y dirigir al Puzzlebot hacia su objetivo deseado. Este controlador ajusta de manera continua la velocidad y la dirección del robot en función del error de posición, utilizando la plataforma de control ROS para enviar los comandos necesarios.

Por último, se ha explorado el uso de sistemas de visión artificial, específicamente la detección de formas y colores mediante una cámara integrada en el Puzzlebot. Esta capacidad

adicional ha permitido al robot percibir y responder a su entorno de manera más sofisticada. La capacidad de identificar colores específicos y ajustar su comportamiento en consecuencia ha añadido una dimensión significativa a su funcionalidad. La detección de color rojo que provoca la detención del robot, el amarillo que reduce su velocidad a la mitad y el verde que le permite avanzar, demuestran la eficacia del sistema de visión artificial integrado en el robot.

Bibliografía o referencias:

- CONTROL ROBUSTO DE UN SISTEMA MECÁNICO SIMPLE MEDIANTE UNA HERRAMIENTA GRÁFICA. revistas.unal.edu.co. (s.f). https://revistas.unal.edu.co/index.php/dyna/article/view/15852/36191

- Pérez, J. (2007). Análisis de sistemas robóticos. *Revista de Ingeniería Robótica*, 7(2), 55-65.https://www.redalyc.org/pdf/784/78470208.pdf

- Ortíz del Castillo, C. (2005). Robustez en controladores retroalimentados. Repositorio institucional del Instituto Tecnológico de Celaya. http://www.iqcelaya.itc.mx/~richart/TesisDoctorado/2005%20Ort%C3%ADz%20del%20Castillo.pdf