Instituto Tecnológico de Estudios Superiores de Monterrey

Campus Puebla

Fundamentación de Robótica (Gpo 101)

Reto semanal 3

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

Fecha de entrega
Domingo 28 de Abril de 2024

RESUMEN

En este proyecto, se creó y aplicó un código en ROS con el objetivo de ingresar una posición deseada y calcular el error de posición en tiempo real. Además, se diseñó un controlador P para minimizar este error y lograr que el robot alcance la posición deseada con la mayor precisión posible representando la trayectoria de 4 figuras, un triángulo, un cuadrado, un pentágono y un hexágono. Para lograrlo, se integraron conceptos de odometría, control en lazo cerrado y navegación punto a punto para un robot diferencial.

OBJETIVOS

El objetivo general es diseñar un controlador P para minimizar dicho error calculado, garantizando que el robot alcance la posición deseada con alta precisión. Sin embargo también existen otros objetivos particulares descritos a continuación:

1. Implementar un código en ROS para ingresar una posición deseada al sistema.
2. Desarrollar algoritmos para calcular el error de posición en tiempo real mediante la integración de términos de odometría.
3. Diseñar e implementar un controlador P para minimizar el error de posición.
4. Integrar el control en lazo cerrado al sistema para garantizar una respuesta precisa del robot ante las variaciones en la posición deseada.
5. Realizar pruebas con las 4 trayectorias de las figuras (triángulo, cuadrado, pentágono y hexágono).
6. Optimizar el rendimiento del sistema y su eficiencia en términos de velocidad de respuesta y precisión en la posición alcanzada por el robot.

INTRODUCCIÓN

Incrementar el grado de interacción y la implementación de funciones disponibles en ROS permite poder realizar movimientos más precisos y robustos en robots. El uso de un controlador P ayuda a mejorar la capacidad del robot para seguir una trayectoria deseada y corregir errores de posición en tiempo real. Es por eso que es importante comprender que un control en lazo cerrado para un robot móvil, a diferencia de un lazo abierto, proporciona retroalimentación continua del estado del sistema, permitiendo ajustes dinámicos que mejoran la precisión y la estabilidad del movimiento. En este contexto, este trabajo se centra en el desarrollo e implementación de un sistema en ROS que integra conceptos de odometría, control en lazo cerrado y navegación punto a punto para lograr un control preciso y eficiente de un robot diferencial.

Un sistema de control en lazo cerrado es aquel que utiliza una medida de la salida real para compararla con una señal deseada, lo cual se conoce como señal de retroalimentación [1]. En este tipo de sistemas, la salida tiene un efecto directo sobre la acción de control. La realimentación de la señal controlada se compara con la entrada de referencia, y se envía una señal actuante proporcional a la diferencia entre ambas, con el fin de disminuir el error y

corregir la salida, esta realimentación negativa ayuda a reducir el error del sistema. En resumen, el lazo cerrado implica el uso de realimentación para ajustar la salida hacia un valor deseado. La Figura 1 ilustra la relación entrada-salida de un sistema de control en lazo cerrado [2].

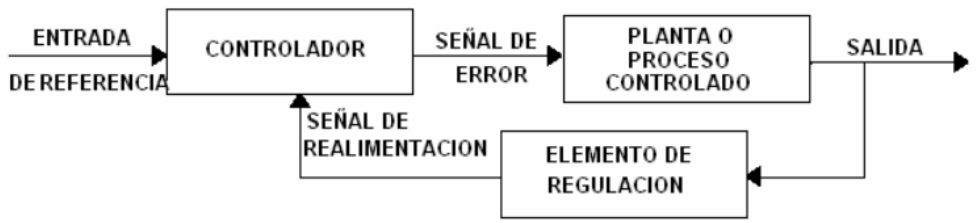

Figura 1. Sistema de Control de Lazo cerrado.

En un robot móvil implica un sistema que continuamente recibe información sobre la posición actual del robot mediante sensores, la compara con una posición deseada y ajusta su movimiento en tiempo real en función de esta diferencia. Esta retroalimentación constante permite generar una señal de control adecuada, que se aplica a los actuadores del robot para corregir su trayectoria y acercarlo a la posición deseada. Este proceso se repite de forma iterativa, garantizando un control preciso y robusto del movimiento del robot, lo que resulta en una mayor estabilidad y precisión en su desplazamiento hacia el objetivo.

Un controlador PID (Proporcional, Integral, Derivativo) es un dispositivo utilizado en sistemas de control para ajustar y mantener una variable controlada (como la posición, la velocidad o la temperatura) lo más cercana posible a un valor deseado o set-point [3]. La acción proporcional, representada por la letra "P" en PID, ajusta la salida del controlador en proporción al error actual, es decir, la diferencia entre la variable controlada y el set-point. Esta ganancia proporcional determina cuánto se debe ajustar la salida del controlador en relación con el error presente.

La acción integral, representada por la letra "I" en PID, tiene en cuenta la acumulación de errores a lo largo del tiempo y ajusta la salida del controlador para eliminar el error de manera gradual. Esto ayuda a corregir desviaciones persistentes entre la variable controlada y el set-point, mejorando la precisión del control. Sin embargo, un ajuste excesivo de la acción integral puede causar oscilaciones e inestabilidad en el sistema.

Por último, la acción derivativa, representada por la letra "D" en PID, tiene en cuenta la tasa de cambio del error y ajusta la salida del controlador para prevenir cambios bruscos en la variable controlada. La acción derivativa contribuye a la estabilidad del sistema al anticipar y contrarrestar los cambios rápidos en la variable controlada.

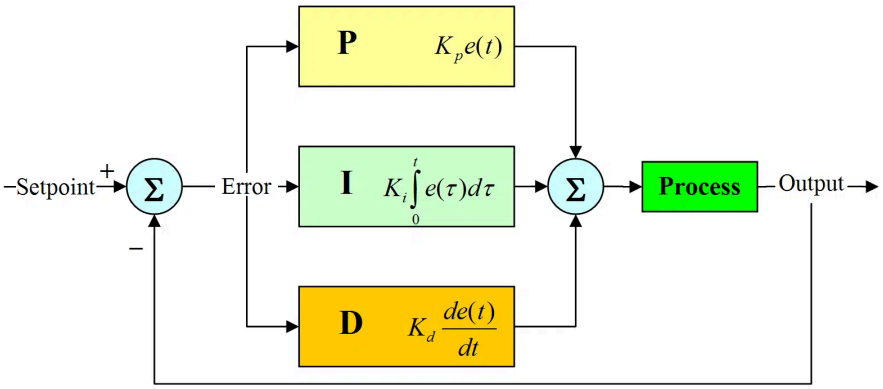

Figura 2. Sistema de Control de Lazo cerrado. con PID

Para implementar un controlador P en un robot diferencial, es importante determinar qué variable se desea controlar en el robot diferencial. Esto podría ser la velocidad lineal, la velocidad angular, la posición en un plano XY, o incluso una combinación de estas variables, dependiendo de la tarea específica que se quiera realizar, en este caso se utilizaron ambas velocidades para ser controladas con la finalidad de que el robot tenga mayor autonomía.

En un controlador proporcional (P), el error se refiere a la discrepancia entre el valor deseado o set-point y el valor real de la variable controlada que está siendo monitoreada por el sistema. Este error es la base sobre la cual el controlador P ópera para ajustar la salida y corregir la desviación entre la variable controlada y el setpoint, es por ello que en este caso tenemos dos errores a corregir. Matemáticamente, el error en un controlador P se calcula como la diferencia entre el set-point (SP) y el valor actual de la variable controlada (PV), y se expresa como:

$$\text{Error} = \text{SP} - \text{PV} \qquad [1]$$

Cuanto mayor sea el error, mayor será el ajuste realizado por el controlador. La ganancia proporcional (Kp) en el controlador P determina la relación entre el error y la señal de control generada [4]. Un valor más alto de Kp amplificar la respuesta del controlador a pequeñas desviaciones, lo que puede acelerar la respuesta del sistema pero también aumentar el riesgo de oscilaciones o inestabilidad si se ajusta incorrectamente.

Para un robot diferencial con un controlador proporcional (P), la robustez puede ser fortalecida desde varias perspectivas. En primer lugar, la selección adecuada de la ganancia proporcional (Kp) es crucial. Una ganancia proporcional demasiado alta puede hacer que el sistema sea sensible a pequeñas perturbaciones, lo que puede llevar a oscilaciones no deseadas o incluso a la inestabilidad del sistema. Por otro lado, una ganancia proporcional demasiado baja puede resultar en una respuesta lenta del sistema, lo que puede afectar la capacidad del robot para seguir trayectorias rápidamente cambiadas o mantenerse en una posición precisa en entornos dinámicos. Por lo tanto, encontrar el equilibrio adecuado entre la sensibilidad del controlador y la estabilidad del sistema es fundamental para mejorar la robustez. [5]

SOLUCIÓN DEL PROBLEMA

Nodo Odometry

Al igual que en las actividades anteriores, para realizar el control de la trayectoria y la posición del robot debemos conocer en todo momento la posición de nuestro robot. Es por ello, que para esta actividad, resulta necesario emplear nuevamente el nodo de odometría que hemos creado anteriormente. Brevemente, recordemos que las ecuaciones de odometría provienen del modelo cinemático del robot (*Figura 3),* en el cual se pueden observar ciertos elementos de importancia que servirán para calcular la posición en todo momento del robot:

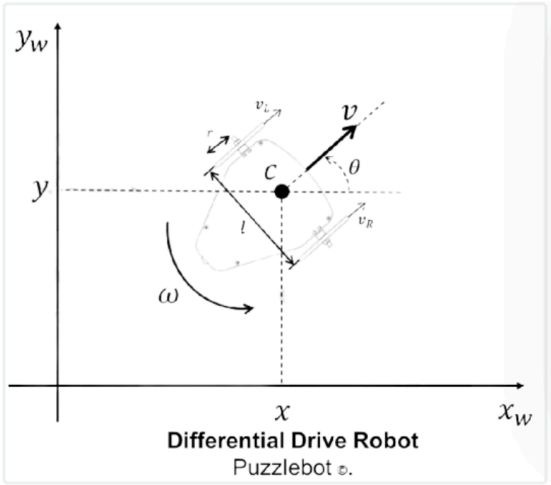

Figura 3. Diagrama del modelo cinemático del Puzzlebot

Por medio de un análisis, es posible construir ciertas fórmulas que podremos plasmar dentro de nuestro código para realizar la implementación de la odometría:

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

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

Ahora que tenemos claro lo que debemos implementar en código, procederemos a realizar la programación en ROS2. Comenzaremos por crear el paquete correspondiente en el que estaremos trabajando:

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

Ahora que tenemos claro lo que debemos implementar en código, procederemos a realizar la programación en ROS2. Comenzaremos por crear el paquete correspondiente en el que estaremos trabajando:

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

A continuación explicamos el funcionamiento del nodo:

1. Importamos las librerías correspondientes para el manejo de nuestro 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, lo que resulta fundamental para realizar el cálculo de la posición:

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

5. Declaramos otras variables que nos servirán para ir realizando los cálculos respectivos de la 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. Si volvemos atrás en la explicación del código de este nodo podremos notar que al suscribirnos a las velocidades asociamos cada suscripción a distintas funciones o callbacks (*read_left* y *read_right*). Dentro de estas funciones, lo único que realizamos es realizar una copia de la información recibida a las variables "locales" de nuestro nodo:

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

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

7. Procedemos con la explicación de nuestra función *position_callback*. Dentro de esta función, comenzamos calculando la velocidad lineal en x y y, así como el cálculo de la velocidad angular:

```
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. Ahora bien, tal y cómo mencionamos anteriormente, para obtener la posición simplemente debemos realizar la integral de la velocidad lineal y angular. Para lograr esto, debemos ir sumando a nuestras variables de posición la velocidad multiplicada el periodo de muestreo ($d = vt$):

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

9. Una vez que hemos calculado las coordenadas de nuestro robot, procederemos a copiar dichos datos a nuestra variable tipo Pose2D para finalmente publicarla a través del topico */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, lo que debemos realizar es crear 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

Tendremos nuestro siguiente nodo que tendrá la función de poder calcular el error de la posición obtenido a partir de una trayectoria determinada. Este nodo, de igual forma, nos permitirá seleccionar la figura que queremos trazar mediante arreglos previamente definidos. Este nodo, estaŕa diseñado de tal forma que nos entregué el error existente entre un punto deseado o setpoint y la posición del robot, entregándonos tanto la distancia entre punto a punto como la diferencia angular entre los mismos.

Para dar inicio, dentro de la carpeta de src del paquete *closed_loop*, se ha incluido un nuevo nodo para representar este error. Sin embargo deben tomarse ciertas consideraciones:

1. Dada la robustez del algoritmo, se han implementado varios nodos dentro el mismo paquete para realizar suscripciones y publicaciones simultáneamente.
2. Para poder compilar varios nodos, es necesario modificar nuestro archivo setup.py, el cual se encuentra dentro de la misma carpeta de *src*.

Dentro de estas modificaciones se incluye indicar el nuevo nodo que se crea y el paquete al que pertenece, almacenado en una nueva etiqueta:

```
        entry_points={

        'console_scripts': [
        'control = closed_loop.control:main',
        'odometry = closed_loop.odometry:main',
        'error = closed_loop.error:main'
        ],
        },
```

Para poder generar el trazado de las figuras y tener una visión más clara de cómo poder implementarlo, partimos del requerimiento de que las figuras debían de estar dentro de un círculo que tuviera 1 metro de diámetro. Por eso decidimos utilizar del programa de Geogebra en el que, partiendo de este círculo unitario, marcamos cada una de las coordenadas partiendo siempre del origen (0,0) ya que de esta manera sería más sencillo identificar el inicio de cada una de las trayectorias.

Como observamos de la figura 4 a la figura 7 podemos ver qué cada una de las figuras fue correctamente dibujada dentro del círculo para que de esta manera tuviéramos las coordenadas para cada uno de los puntos. En el código los declaramos como una lista de listas donde definimos las coordenadas de x,y para cada punto de la figura.

```
self.triangle = [[0.7,0.0], [0.35,0.85], [0.0,0.0]]
```

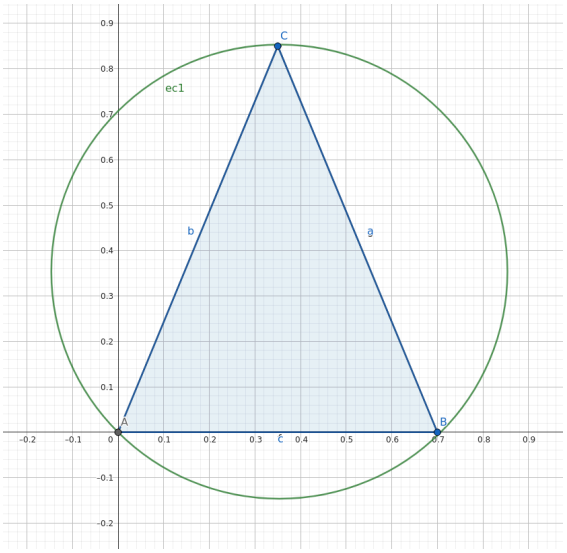

Figura 4. Trazado de las coordenadas del triángulo

```
self.square = [[0.7,0.0], [0.7,0.7], [0.0,0.7],[0.0,0.0]]
```

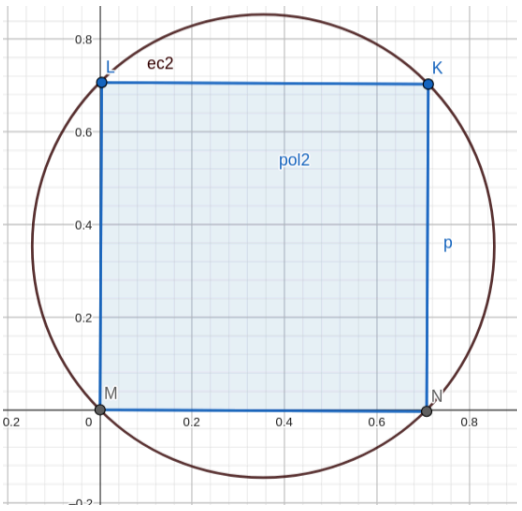

Figura 5. Trazado de las coordenadas del cuadrado

```
self.pentagon =[[0.63,0.0],[0.82,0.6],[0.31,0.95],[-0.2,0.6],[0.0,0.0]]
```

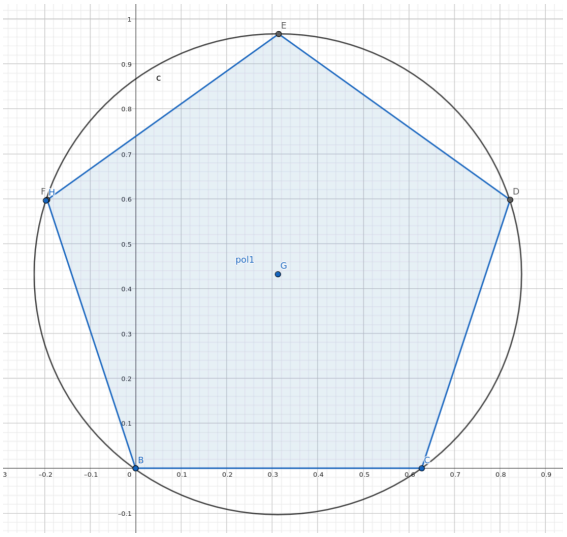

Figura 6. Trazado de las coordenadas del pentágono

```
self.hexagon=[[0.5,0.0],[0.75,0.43],[0.5,0.85],[0.0,0.85],[-0.25,0.43],
[0.0,0.0]]
```

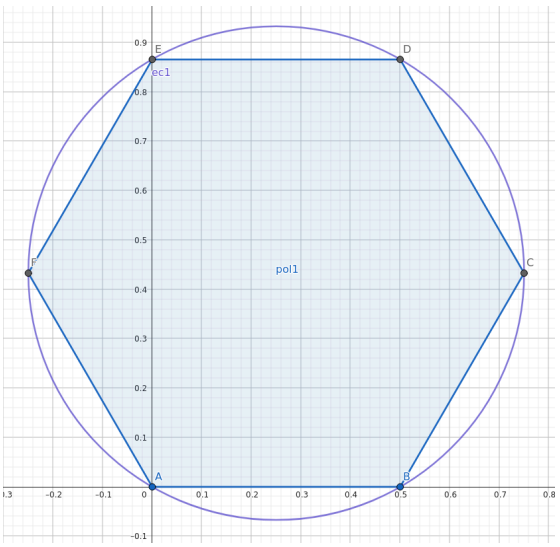

Figura 7. Trazado de las coordenadas del hexágono

Antes de continuar con la explicación del nodo, resulta importante mencionar que para este nodo y para el del controlador decidimos implementar un mensaje personalizado llamado Error2D, que incluye tanto el error de distancia como de ángulo. Dicho esto, procedemos a explicar el funcionamiento del nodo de error:

1. Al igual que en el nodo previamente descrito, importamos las librerías correspondientes:

```
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. Inicializamos el nodo bajo el nombre ‘error_node’ y creamos una suscripción al tópico de /Position, pues es el que nos servirá para calcular el error:

```
class My_publisher(Node):
   def __init__(self):
       super().__init__('error_node')
       self.position_subscriber = self.create_subscription(Pose2D, '/Position',
       self.calculate_error, rclpy.qos.qos_profile_sensor_data)
```

3. Procedemos a crear un publisher destinado a mandar el error al nodo del controlador. Este publisher publicará a la misma frecuencia que el tópico de /Position, ya que publicara cada que realice este cálculo:

```
       self.error_publisher = self.create_publisher(Error2D, 'error', 10)
```

4. Para este caso, hemos decidido declarar los puntos de las trayectorias o setpoints dentro de una lista de listas, en la que contiene las coordenadas tanto en el eje x como en el eje y. De igual forma, contamos con una variable de selección que se deberá modificar acorde a la figura que deseamos trazar:

```
        self.triangle = [[0.7,0.0], [0.35,0.85], [0.0,0.0]]
        self.square = [[0.7,0.0], [0.7,0.7], [0.0,0.7],[0.0,0.0]]
        self.pentagon = [[0.63,0.0], [0.82,0.6], [0.31,0.95],[-0.2,0.6], [0.0,0.0]]
    self.hexagon = [[0.5,0.0], [0.75,0.43], [0.5,0.85],[0.0,0.85], [-0.25,0.43], [0.0,0.0]]

        #se realiza cambio dependiendo de la figura
        self.selection = self.square
```

5. Contamos con una variable tipo index que nos servirá para ir pasando de punto a punto de las trayectorias. De igual forma, creamos nuestra variable de error y una variable auxiliar (move) que será útil para detener el robot cuando termine la trayectoria:

```
        self.index_point = 0

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

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

6. Procedemos a explicar la función destinada a calcular 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:

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

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

La primera parte de la función se dedica a implementar estas funciones en código, específicamente la fórmula dedicada a calcular la distancia entre los puntos:

```
def calculate_error(self,msg):
    if self.move:

        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)
```

6. Posterior a esto, realizamos el cálculo del error de ángulo pero considerando siempre el punto objetivo y el punto anterior, ya que considerar el error en relación al origen sería una equivocación que afectaría en el comportamiento de nuestro algoritmo:

```
try:
    x_reference = x_reference - self.selection[self.index_point-1][0]
    y_reference = y_reference - self.selection[self.index_point-1][1]

except:
    pass

try:
    angle = np.arctan2(y_reference, x_reference)
except:
    angle = 0

angle_degrees = np.degrees(angle)
```

De igual forma, se agregan unos cuantos condicionales para que el ángulo sea positivo, manteniendo asi el mismo sentido de giro siempre:

```
if angle_degrees < 0:
    angle_degrees +=360

angle_error = angle_degrees - msg._theta

self.setpoint.data = angle_degrees
```

7. Por último, copiamos el error a la variable de tipo Error2D y la publicamos:

```
self.error.distance_error = distance
self.error.theta_error = angle_error

if (distance <= 0.17 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)
```

Notemos que tenemos un pequeño condicional encargado de verificar que el error sea pequeño (dentro de un margen) y, cuando es el caso, cambia al siguiente punto o setpoint y en

caso de sobrepasar la cantidad de puntos del arreglo modifica la variable auxiliar move para ya no continuar con el cálculo de error y, además, manda errores nulos al publisher para que de esta forma el controller pueda detener el movimiento.

8. Finalmente, inicializamos el nodo:

```
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()
```

```
#se realiza cambio dependiendo de la figura
        self.selection = self.hexagon
        self.index_point = 0
```

Nodo Controller

En términos generales este nodo se encarga de leer el error obtenido a partir de los cálculos de odometría y agregarle un controlador proporcional para disminuirlo y tener un control de la velocidad hasta llegar a un punto objetivo. Con este método es posible construir una serie de trayectorias para el puzzlebot y lograr que este las recorra de una manera controlada.

A continuación se muestra el funcionamiento del nodo:

Como se ha estado trabajando anteriormente, se utilizan las librerías convencionales

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

Posteriormente, se crea la clase My_publisher y se agrega el nodo de trabajo controller node. Dentro de este, la tarea principal es manipular el error y escribir una nueva velocidad ya

controlada. Por ello, se realiza una suscripción al nodo error y una publicación de una nueva velocidad, a través del tópico ya trabajado /cmd_vel

```
self.error_subscriber=self.create_subscription(Error2D,'/error',    self.control_puzzlebot,
rclpy.qos.qos_profile_sensor_data)
self.speed_publisher = self.create_publisher(Twist, '/cmd_vel', 10)
```

Ahora se definen las variables de la clase que llevarán la lectura de la velocidad controlada, para ello, se incluyen variables para delimitar la velocidad angular y lineal máxima y mínima del robot, de esta forma se garantiza el correcto funcionamiento del mismo. También se encuentran las variables que multiplican al error obtenido, cada una para su velocidad correspondiente. Finalmente se definen variables para leer el mensaje obtenido del tópico *error*.

```
        #variables del controlador para velocidad lineal y angular
        self.kv = 0.4
        self.kw = 0.03
        #velocidades para limitar la velocidad del puzzlebot
        self.v = 0.0
        self.v_max = 0.4
        self.v_min = 0.05
        self.w = 0.0
        self.w_max = 0.45
        self.w_min = -0.3

        #error
        self.e_theta = 0.0
        self.e_distance = 0.0
        self.speed_bot = Twist()
        #mensaje para indicar que se ha inicializado el nodo correctamente
        self.get_logger().info('Controller node succesfully initialized !')
```

Para indicar que el robot debe detener su movimiento, se crea la siguiente función. Se encarga de indicar que la velocidad lineal y angular cuando el error leído es 0 para el ángulo y la distancia.

```
    def control_puzzlebot(self, msg):
      if msg.theta_error == 0.0 and msg.distance_error == 0.0:
           self.speed_bot.linear.x = 0.0
           self.speed_bot.angular.z = 0.0
```

En caso contrario, se ejecutan distintas funciones:

1. Obtener el error leído en el tópico al que se ha suscrito y se almacena en su respectiva variable.

```
      self.e_theta = msg.theta_error
      self.e_distance = msg.distance_error
```

2. Se multiplican los errores lineal y angular por una constante kv y kw respectivamente y se almacena en una velocidad general (que se envía a cmd_vel).

```
self.v = self.e_distance * self.kv
```

3. En caso de que la velocidad actual del robot sobrepase un valor y pueda presentar problemas para el sistema, esta se delimita y se obtiene un mejor control de la misma.

```
#se delimita la velocidad
if self.v > self.v_max:
   self.v = self.v_max
elif self.v < self.v_min:
      self.v = 0.0
      self.w = self.e_theta * self.kw
if self.w > self.w_max:
     self.w = self.w_max
elif self.w < self.w_min:
     self.w = self.w_min
```

4. Finalmente, se almacena dentro de la variable a publicar la velocidad lineal y angular obtenida y controlada en formato get_logger dentro del tópico speed_publisher.

```
 self.speed_bot.linear.x = self.v
    self.speed_bot.angular.z = self.w

self.get_logger().info(f'v: {self.v}, w: {self.w}')
self.speed_publisher.publish(self.speed_bot)
```

Al igual que el resto de los nodos, es importante inicializar el nodo del controlador:

```
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()
```

RESULTADOS

Video de funcionamiento:
https://drive.google.com/file/d/1Ay7U7glR4_E_gQBLd5Ju28F9KCEOxInX/view?usp=sharing

Para la parte de los resultados se tuvo un ajuste correcto para el controlador de lazo cerrado por lo que podemos observar en el video que el Puzzlebot correctamente está siguiendo cada una de las figuras que plantea el reto, además sabemos que como ya lo explicamos en la parte del error_node fue donde propusimos los siguientes arreglo de arreglos para cada una de las figuras:

```
self.triangle = [[0.7,0.0], [0.35,0.85], [0.0,0.0]]
self.square = [[0.7,0.0], [0.7,0.7], [0.0,0.7],[0.0,0.0]]
self.pentagon=[[0.63,0.0],[0.82,0.6],[0.31,0.95],[-0.2,0.6], [0.0,0.0]]
self.hexagon=[[0.5,0.0],[0.75,0.43],[0.5,0.85],[0.0,0.85],[-0.25,0.43],
[0.0,0.0]]
```

Donde vamos a poner las coordenadas que con ayuda del programa de Geogebra podemos ajustarlo de buena manera a un círculo con un diámetro de 1 m para que las coordenadas de la figura estuvieran dentro de este mismo círculo. También tenemos la variable de selection para poder cambiar la figura dependiendo de la que queramos trazar con el robot:

```
#se realiza cambio dependiendo de la figura
        self.selection = self.hexagon
        self.index_point = 0
```

Para comenzar en el video mostramos la figura del triángulo donde por cada punto que pase nosotros pusimos una tarjeta para poder identificar correctamente cuáles eran los puntos o vértices y hace la rotación para continuar al siguiente punto. Después continuamos con el cuadrado que también resultó con un trazado correcto debido a qué no tuvo ninguna irregularidad para poder llegar a cada punto, posteriormente para las figuras con más lados como lo fue el pentágono y el hexágono también se generaron las trayectorias de una buena manera aunque al principio si tuvimos un poco de complicaciones debido que las variables para el contolador podrían seguir trayectorias largas de puntos más separados pero con puntos más cercanos unos a otros si empezaba a comenzar a tener un desfase de las trayectorias. Con el controlador ajustado de mejor manera y con valores más calculados que fuimos probando también con los valores que arrojaban los tópicos del error y de la posición con los comandos:

ros2 topic echo /error
ros2 topic echo/Position

Pudimos ajustar las variables del controlador para cada velocidad y que no tuviera ninguna irregularidad en ningún trazado y en ningún giro cuando llegaba al punto y tenía que ir al siguiente:

```
#variables del controlador para velocidad lineal y angular
 self.kv = 0.4
 self.kw = 0.03
```

Como una posible mejora que podemos implementar podría ser el utilizar un controlador más preciso y que tuviera un mejor ajuste para los parámetros de la ganancia proporcional (Kp), además de también poder probar con el control integral debido a que este calcula la integral de la señal del error y la multiplica por la constante Ki, de esta manera la integral se ve como la acumulación del error, consigue poder reducir el error del sistema y aumenta un poco la velocidad del sistema.

CONCLUSIONES

El resultado obtenido por el controlador P en este proyecto fue exitoso, demostrando su eficacia en la minimización del error de posición y en la consecución de una alta precisión en el movimiento del robot. Mediante la implementación de un sistema en ROS que integra conceptos de odometría, control en lazo cerrado y navegación punto a punto, logramos alcanzar el objetivo general de diseñar un controlador P para garantizar que el robot alcance la posición deseada para la trayectoria de las distintas figuras.

La adecuada selección de la ganancia proporcional (Kp) fue fundamental para el éxito del controlador P. Esta ganancia determina la relación entre el error y la señal de control generada, permitiendo ajustes precisos en la respuesta del sistema ante desviaciones entre la posición deseada y la posición real del robot. Además, el control en lazo cerrado proporcionó una retroalimentación continua del estado del sistema, permitiendo ajustes dinámicos que mejoraron la precisión y la estabilidad del movimiento del robot.

Las pruebas realizadas en los diferentes escenarios confirmaron el correcto funcionamiento del sistema, validando su eficiencia en términos de velocidad de respuesta y precisión en la posición alcanzada por el robot. Una posible mejora a la metodología implementada podría ser la exploración de técnicas avanzadas de ajuste automático de parámetros, como el uso de algoritmos de sintonización automática. Estos algoritmos pueden optimizar automáticamente los parámetros del controlador P, como la ganancia proporcional (Kp), en función del

comportamiento dinámico del sistema en tiempo real. La implementación de tales técnicas podría permitir una adaptación más rápida y precisa del controlador a cambios en las condiciones del entorno o en la dinámica del robot, mejorando aún más la robustez y la eficacia del sistema de control.

REFERENCIAS

1. Nise, N. S., & Romo, J. H. (2002). *Sistemas de control para ingeniería*. Patria Cultural.
2. 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
3. Manufactura de Ingenios Tecnológicos. (18 de Agosto de 2022). ¿Qué es el control PID? Sus aplicaciones en la Industria 4.0. https://mintforpeople.com/noticias/ciberseguridad-industria-4-0/
4. Navarro, D., Gilabert, G. B., Rios, L. H., & Bueno, M. (2007). Mejoras de la localización odométrica de un robot diferencial mediante la corrección de errores sistemáticos. *Scientia et technica*, *13*(37), 37-42.
5. Vilanova, R., & Alfaro, V. M. (2011). Control PID robusto: Una visión panorámica. *Revista Iberoamericana De Automática E Informática Industrial*, *8*(3), 141–158. https://doi.org/10.1016/j.riai.2011.06.003