Instituto Tecnológico y de Estudios Superiores de Monterrey
Campus Puebla

Implementación de Robótica Inteligente TE3002B GRUPO(502)

Alumnos

Diego García Rueda
A01735217

Jhonatan Yael Martínez Vargas
A01734193

Jonathan Josafat Vázquez Suárez
A01734225

Profesores

Dr. Alfredo García Suárez

Dr. Rigoberto Cerino Jiménez

Dr. Juan Manuel Ahuactzin Larios

Dr. Cesar Torres Rios

25 abril 2023, Puebla, México.

Resumen

En este reporte se mostrarán los resultados del equipo para este reto semanal, se muestran las mejoras aplicadas al resultado del reto anterior siendo la principal la implementación de un sistema de control de lazo cerrado. Se detalla la implementación de un segundo nodo llamado “SetPoint” y el proceso de obtención del sistema de control.
De la misma manera que en el reto anterior, los nodos generados de ROS se conectarán al Puzzlebot mediante una comunicación SSH. Este reporte tiene como objetivo mostrar las mejoras respecto al entregable anterior principalmente en la implementación del sistema de control en el recorrido de una trayectoria ingresada por el usuario.

Introducción

En aplicaciones como la robótica móvil, el diseño e implementación de un buen sistema de control es algo esencial ya que permite a los robots moverse de manera segura y eficiente en entornos dinámicos y desconocidos, por tal motivo y como parte del segundo mini challenge propuesto por Manchester Robotics, en esta ocasión se tiene que implementar un controlador PID dentro de un nodo de ROS para permitir que el puzzlebot, pueda seguir una trayectoria en lazo cerrado.

La implementación de un controlador PID en un nodo de ROS permite ajustar las velocidades de los motores del robot en tiempo real, lo que permite al robot adaptarse y seguir una trayectoria de manera más precisa y estable, por lo que el uso de un controlador PID también permite al robot compensar las perturbaciones externas que pueden afectar su movimiento, lo que lo hace ideal para aplicaciones en entornos dinámicos y cambiantes.

Por lo que este mini reto consta de dos partes, la primera implica el diseño del nodo ***"Set_Point"***, que es responsable de solicitar al usuario que introduzca los puntos que conforman la trayectoria que el robot deberá seguir. Los cuales posteriormente serán transformados en instrucciones que permitan al robot seguir la ruta deseada.

La segunda parte del reto implica la creación del nodo ***"Controller"***, donde se implementa el controlador PID. Este nodo recibe los *"setpoints"* y es responsable de ajustar la velocidad de los motores del puzzlebot para que éste se oriente y recorra la distancia indicada.

Objetivos

Desarrollar capacidades en el manejo de ROS y gazebo implementando a estos en el control del Puzzlebot, mejorando así el resultado del reto anterior agregando un sistema de control de lazo cerrado.
Desarrollar un nodo que mediante un sistema de control de lazo cerrado, un controlador PID, sea capaz de seguir una ruta especificada por el usuario, el Puzzlebot debe de ser capaz de modificar su trayectoria corrigiendo su error en tiempo real, logrando llegar asi a la posición deseada.

Solución del problema

Para la solución de este reto se decidió utilizar los paquetes proporcionados por la organización socio formadora, estos paquetes son de ayuda ya que contienen todos los recursos necesarios para la simulación del Puzzlebot en *gazebo* . El equipo se encargo de desarrollar 2 nodos siguiendo las indicaciones del socio formador, siguiendo como base la siguiente estructura:

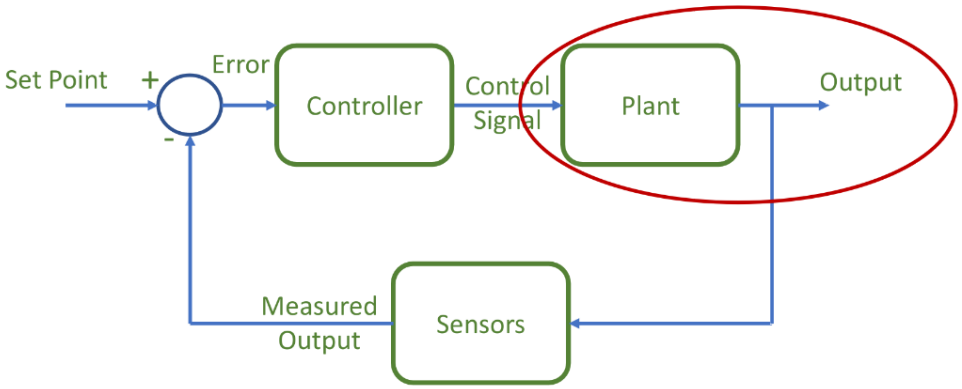

Figura 1. Estructura utilizada para el reto.

Los nodos desarrollados son “SetPoint” y “Controller”. Estos se abordarán a continuación:

- **Set_Point**

Este nodo se encarga de publicar en un tópico llamado *set_point*, de la misma manera, se encuentra suscrito al tópico *status* que será el que definirá la acción que tendrá que realizar este nodo.
Este nodo es el encargado de definir cuál será la trayectoria que tiene que seguir el Puzzlebot solicitando al usuario las coordenadas globales que forman la trayectoria a seguir. Se utilizan 2 operaciones para el cálculo de la distancia que tiene que recorrer el Puzzlebot y el ángulo que tiene que girar para llegar a este punto. Las operaciones son las siguientes:

$$distance = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2} \quad (1)$$

$$angle = atan2((y_2 - y_1, (x_2 - x_1) \quad (2)$$

Los valores de ‘x’ y ‘y’ son ingresados por el usuario, se inicializan los primeros valores en 0 para poder así definir la rotación y traslación necesaria para poder llegar al primer punto. Este proceso se repite para cada uno de los puntos de las trayectorias. El ángulo pasa por una función llamada *angle_wrap*, esta función se encarga de verificar que el ángulo de giro se encuentre dentro de un rango de -π a π radianes, en caso de que no sea así esta función se encarga de ajustar el ángulo al umbral permitido.

Estos datos de rotación y de traslación se almacenan en un vector que se publicará al tópico *set_point*, tras haber definido las acciones de un punto, las nuevas coordenadas del robot pasan a ser las del punto calculado, estas coordenadas tienen que actualizarse para así poder realizar los cálculos de rotación y traslación de manera adecuada para el siguiente punto.

```
for i in range(n_points):

    target_x = float(input(f"Ingresa la coordenda x del punto {i}: "))
```

```
        target_y = float(input(f"Ingresa la coordenda y del punto {i}: "))
        print("")

            target_distance = math.sqrt(((target_x - current_x) ** 2) + ((target_y - current_y) ** 2)) +
prev_distance
        target_theta = math.atan2(target_y - current_y, target_x - current_x)
        target_theta = angle_wrap(target_theta)

        target_list.append((target_distance, target_theta))

        current_x = target_x
        current_y = target_y
        prev_distance = target_distance

    return target_list
```

Figura 2. Código de cálculo de rotación y traslación a los puntos.

El nodo se encarga de publicar los movimientos necesarios de cada punto en caso de recibir un mensaje “new_target” del status, en este caso se enviará la distancia a recorrer y el ángulo de giro, se hará un incremento al contador que define en qué punto de la trayectoria se encuentra el Puzzlebtot, en caso de que este iterador se encuentre en la última posición se enviaran los movimientos del último punto y finalizará el nodo.
En caso de que el mensaje del status sea “continue” no se realizará nada hasta que el mensaje recibido cambie.

```
def new_target_callback(msg):

    # Publica el siguiente punto en la lista si hay más elementos y el mensaje no está vacío
    if msg.data == "new_target":

        print("Nueva solicitud recibida")

        if publisher.index < len(publisher.target_list):

            target = SetPoint()

            target.id = publisher.index
            target.limit = len(publisher.target_list)
            target.distance, target.orientation = publisher.target_list[publisher.index]

            publisher.pub.publish(target)
            rospy.loginfo(f'Nuevo Objetivo Enviado: ({target.distance}, {target.orientation})')
            publisher.index += 1

        else:

            target = SetPoint()

            target.id = publisher.index
            target.limit = len(publisher.target_list)
            target.distance, target.orientation = publisher.target_list[publisher.index - 1]

            publisher.pub.publish(target)

            rospy.signal_shutdown('Trayectoria completada')

    elif msg.data == "continue":

        pass
```

Figura 3. Código para la publicación de movimientos de Rotación y Traslación

- **Controller**

Este nodo es el encargado de controlar el movimiento del robot, para esto utiliza un *Twist()*, este define las velocidades lineales y angulares del robot, el movimiento de este robot se verá afectado por un sistema de control PID tanto para el error de distancia como para el error del ángulo que exista entre la posición del robot y el objetivo, es decir, el punto de la trayectoria a la que el robot busque llegar. En este nodo se buscó tener un acercamiento mediante Programación Orientada a Objetos para una mejor organización y desarrollo del nodo.
Se inicializar atributos utilizando datos ingresados en el archivo **params.yaml** estos datos fueron ingresados mediante el comando *rospy.get_param()*, los atributos inicializados de esta manera fueron los siguientes:

- ➔ Radio de las ruedas.
- ➔ Distancia entre las ruedas.
- ➔ Velocidad angular de cada rueda.
- ➔ Tiempo de muestreo.
- ➔ Distancia recorrida por el Puzzlebot
- ➔ Velocidad linear y angular del Puzzlebot en los 3 ejes.
- ➔ Velocidad máxima linear y angular.
- ➔ Posición en el eje X y Y además de la orientación del Puzzlebot.
- ➔ Los valores de las constantes de los controladores PID para el error lineal y angular.

Eset nodo se suscribe a los tópicos: *wr, wl* y *set point* , los 2 primeros sirven para realizar todos los cálculos de la posición y la orientación del robot y el tercer tópico es del nodo que se detalló anteriormente en este reporte. Este nodo pública en los tópicos *cmd_vel* y *status*, el primer tópico es el que controla el movimiento del Puzzlebot en la simulación y de manera física, el segundo tópico se relaciona con el nodo ***Set_Point*** que definirá si se calcularán los movimientos de traslación y rotación del robot para el siguiente punto.

La función principal de este nodo es la función *Run(),* esta función para el uso del sistema de control utiliza 2 controles: *distance_error* y *orientation_error. Podemos definir a los errores con las siguientes fórmulas:*

$$distanceError = distanciaDeseada - distanciaRecorrida \quad (3)$$
$$orientationError = anguloDeseado - anguloGirado \quad (4)$$

El nodo también almacena un error de distancia y de giro previos, estos errores se utilizarán en el sistema de control. La función utiliza intervalos de tiempo llamados *dt*, los intervalos de tiempo servirán para los sistemas de control, el cálculo del desplazamiento y la orientación del Puzzlebot.

- ❖ Funcionamiento error Orientación:

Se calcula un umbral entre -0.025 y 0.025 para este error, si se encuentra dentro de este umbral se define si es que el valor del error es positivo o negativo. En caso de ser positivo significa que el error de orientación hace que el robot gire en dirección izquierda más de lo necesario. Si el error es negativo, la dirección es hacia la derecha. En este caso se manda a llamar a la función

control() especificando que es el giro al que queremos controlar, para esto se usa un ID, definiendo si el valor del error de orientación es positivo o negativo podemos conocer a qué rueda es que tenemos que aplicarle el control y utilizando el diferencial de tiempo.
Obteniendo los nuevos valores de las velocidades de cada rueda procedemos a calcular el nuevo ángulo con la siguiente fórmula:

$$\Theta_{k+1} = \Theta_k + r\frac{W_R + W_L}{2}dt \quad (5)$$

```
if self.orientation_error > 0.0:
                        print("Gira Izquierda")
                        print("")

                        self.control("wl", dt)

elif self.orientation_error < 0.0:

                        print("Gira Derecha")
                        print("")

                        self.control("wr", dt)

print(f"Nuevo wl: {self.wl}")
print(f"Nuevo wr: {self.wr}")
next_orientation = self.current.orientation - (self.r * ((self.wr - self.wl) / self.l) * dt)
```

Figura 4. Código ejemplo de la aplicación del sistema de control angular y cálculo del nuevo ángulo

La orientación actual del Puzzlebot pasa a tener el valor obtenido calculado con la fórmula anterior, finalmente se publica un status de "continue" que servirá para el nodo *Set_Point.*

- ❖ Funcionamiento error Distancia:

Se calcula un umbral de 0.025, esto indica que la distancia que el Puzzlebot ha recorrido necesita recorrer por lo menos 25 cm más para llegar a su objetivo. En esta situación se manda a llamar a la función ***control()*** especificando que es la distancia la que queremos modificar, para esto se usa un ID. Tras modificar las velocidades de ambas ruedas, se procede a calcular la siguiente posición del Puzzlebot en el eje X y Y usando las siguientes fórmulas:

$$X_{k+1} = X_k + r\frac{W_R + W_L}{2}dt\cos\Theta \quad (6)$$

$$Y_{k+1} = Y_k + r\frac{W_R + W_L}{2}dt\sin\Theta \quad (7)$$

En base a las funciones anteriores podemos calcular la nueva magnitud, que se incrementará a la distancia recorrida, de la misma manera, se modificará la posición del robot usando las fórmulas (6) y (7).

```
elif self.distance_error > 0.025:
      print(f"wl Actual: {self.wl}")
      print(f"wr Actual: {self.wr}")

               self.control("l", dt)
```

```
        print(f"Nuevo wl: {self.wl}")
        print(f"Nuevo wr: {self.wr}")

        dx = self.r * ((self.wr + self.wl) / 2.0) * dt * math.cos(self.current.orientation)
        dy = self.r * ((self.wr + self.wl) / 2.0) * dt * math.sin(self.current.orientation)
        dd = math.sqrt((dx ** 2) + (dy ** 2))

        self.current.pos_x += dx
        self.current.pos_y += dy
        self.traveled_distance += dd
```

Figura 5. Código ejemplo de la aplicación del sistema de control lineal y cálculo de la nueva posición.

Finalmente, si los 2 errores son 0, esto nos quiere decir que llegamos al punto de la trayectoria deseada, los valores de los errores se reinician y se cambia el status a *new target.*

```
else:
    self.distance_error = 0.0
    self.orientation_error = 0.0
    self.last_time = rospy.get_time()
    self.status_pub.publish("new_target")
```

Figura 6. Código reinicio Errores y cambio de status.

La función de control se relaciona completamente con la función de movimiento, esta función usa los ID de la función ***run()*** para definir qué tipo de control se utilizará, cada tipo utiliza diferentes valores de las constantes proporcional, integral y derivativa.

Si el control es para movimiento lineal se calcula el control Proporcional, Integral y Derivativo multiplicando el error de distancias por cada constante y el diferencial de tiempo *dt*, el nuevo valor de la velocidad es la suma de los 3 controles, también se establece que la velocidad angular en el eje Z (esta velocidad realiza la rotación del robot) sea 0, esto con motivo de que únicamente haya velocidad lineal, en caso de que esta nueva velocidad sea mayor que la velocidad máxima entonces obtendrá el valor de la velocidad máxima.

```
if id == "l":
        print("Control Lineal")
        print("")

        # Calculo Control Proporcional
        p = self.linear_kp * self.distance_error

        # Calculo control Integral
        self.integral_distance += self.distance_error * dt
        i = self.linear_ki * self.integral_distance

        # Calculo Control Derivativo
        self.derivative_distance = (self.distance_error - self.prev_distance_error) / dt
        d = self.linear_kd * self.derivative_distance

        self.vel.angular.z = 0.0

        new_vel = p + i + d

        if new_vel >= self.max_linear_vel:
```

```
            self.vel.linear.x = self.max_linear_vel

        else:

            self.vel.linear.x = new_vel
```

Figura 7. Código sistema de Control - Error de Distancia

Si el control es para rotación del robot, se calculan los 3 controles utilizando el error de orientación, el diferencial de tiempo *dt* y las constantes proporcional, integral y derivativa para el movimiento rotacional. Se establece que la velocidad lineal en el eje X sea 0 para así definir que únicamente haya velocidad angular.
La nueva velocidad será la suma de los 3 controles, este nueva velocidad se le asignará a la rueda que se haya indicado mediante el ID, en caso de que esta velocidad tenga un valor mayor a la velocidad máxima angular del Puzzlebot, el valor será el de la velocidad máxima angular.

```
else:
        print("Control Angular")
        # Calculo Control Proporcional
        p = self.angular_kp * self.orientation_error

        # Calculo control Integral
        self.integral_orientation += self.orientation_error * dt
        i = self.angular_ki * self.integral_orientation

        # Calculo Control Derivativo
        self.derivative_orientation = (
            self.orientation_error - self.prev_orientation_error) / dt
        d = self.angular_kd * self.derivative_orientation

        self.vel.linear.x = 0.0
        new_vel = p + i + d
        if id == "wl":

            if new_vel >= self.max_linear_vel:
                self.vel.angular.z = -self.max_linear_vel

            else:
                self.vel.angular.z = -new_vel

        else:

            if new_vel >= self.max_linear_vel:
                self.vel.angular.z = self.max_linear_vel

            else:
                self.vel.angular.z = new_vel
```

Figura 8. Código sistema de Control - Error de Orientación

Resultados

A continuación se presentan los resultados obtenidos, los siguientes videos muestran los resultados de la simulación en Gazebo y el funcionamiento del robot de manera física.

Simulación Gazebo:
https://youtu.be/4fwzuqbmKxE

Implementación física:
https://www.youtube.com/watch?v=j-nXb1ERA-s&ab_channel=JhonatanYaelMart%C3%ADnezVargas

Conclusión del equipo

Como equipo consideramos que este challenge nos ayudó a comprender de mejor manera cómo implementar este tipo de controladores en código, pues han sido pocas las veces que hemos tenido que diseñar e implementar un PID, lo cual si bien nos has costado trabajo, también nos ha permitido desarrollarnos y comprender un poco más acerca del funcionamiento y aplicaciones de este mismo en proyectos reales.