Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Puebla

TE3003B.501

Integración de robótica y sistemas inteligentes

Reto Semanal 2 | Manchester Robotics

Frida Lizett Zavala Pérez A01275226

Diego Garcia Rueda A01735217

Alejandro Armenta Arellano A01734879

19 de Abril del 2024

Índice

Resumen

El proyecto se enfocó en la simulación del robot diferencial *PuzzleBot* utilizando *ROS* y RViz. El objetivo principal fue visualizar la simulación del robot ensamblado y su trayectoria utilizando datos de odometría. Se emplearon archivos *URDF* para la visualización del robot ensamblado con todas sus piezas en *Rviz,* se calculó la posición del robot a partir de sus velocidades angulares y lineales, empleando también un controlador para el movimiento.

Objetivos

El objetivo del proyecto es desarrollar la simulación del movimiento cinemático y nodos de localización y control para el robot Puzzlebot, haciendo uso de la navegación por cálculo muerto o navegación deducida. En la primera parte, se construirá un simulador basado en el modelo cinemático de un robot no holonómico para el Puzzlebot. En la segunda parte, se desarrollará un nodo de localización mediante el cálculo de posición, cuyos resultados se visualizarán en Rviz con un modelo tridimensional del robot. Finalmente, en la tercera parte, se creará un nodo de control para estabilizar la posición del Puzzlebot, para concluir en una comprensión práctica de los conceptos de cinemática, localización y control en un entorno de simulación.

Introducción

Para este reto se emplearon conceptos nuevos, que al integrarse permiten generar la solución para el reto propuesto, el primero de ellos fué los mensajes tipo *PoseStamped* de *ROS* el cual es un tipo de mensaje para representar la posición y orientación de un objeto en 3D. Este mensaje se usa en robótica para transmitir la ubicación asociada a un tiempo específico de

objetos dentro de un marco de referencia conocido, en este caso sería el robot *Puzzlebot*. La estructura que maneja este tipo de mensajes es la siguiente:

- Header: El encabezado que proporciona metadatos tales como el marco de referencia, el tiempo en el que se tomó la medición, y la secuencia.
- Pose: Describe la posición y orientación del objeto en el espacio tridimensional.
    - Posición: Representa la ubicación 3D del objeto, usualmente especificada como coordenadas (x, y, z) en metros.
    - Orientación: Representa la orientación del objeto en el espacio, especificada como un cuaternión (x, y, z, w) o como ángulos de Euler (roll, pitch, yaw).

Por otro lado también es importante comprender la cinemática de un robot diferencial, la cual es el estudio del movimiento de un robot que cuenta con dos ruedas con movimientos separados, es decir que las ruedas pueden tener diferentes velocidades cada una. Un modelo cinemático como el que se puede emplear para la resolución del reto incluye la configuración del robot, que es la posición y orientación del robot en un sistema de coordenadas cartesianas y un ángulo de orientación. Del mismo modo la velocidad lineal a la que están girando las ruedas es un factor importante a considerar, así como la relación entre dichas velocidades de las ruedas y el movimiento del robot, siendo expresado esto mediante ecuaciones que relacionan éstas velocidades, con la velocidad lineal y angular del robot. Otro punto indispensable es la trayectoria del robot, la cual puede ser calculada mediante la integración de las velocidades del robot con respecto del tiempo. Igualmente se deben tomar en cuenta las perturbaciones cinemáticas que se puedan presentar en el espacio.

Ligado a lo anterior se usó la odometría, la cual jugó un papel indispensable en la resolución del reto. La odometría en la robótica estima la posición y orientación relativas de un robot mediante la información obtenida de su movimiento, en este caso la información se obtiene a

partir de las ruedas del robot. Se basa en la medición de la velocidad y la distancia recorrida a lo largo del tiempo, con esa información y los modelos cinemáticos se realizan las estimaciones.

Retomando lo anterior, se puede comenzar a implementar el *Dead Reckoning*, también conocido como "n*avegación por cálculo muerto*" o "n*avegación deducida*", éste es un método utilizado en la navegación y la robótica para estimar la posición y la orientación de un objeto en movimiento basándose únicamente en sus movimientos previos y en los datos de su velocidad y dirección actuales. Para este reto se considera la información de las ruedas, tomando la velocidad y dirección del robot, así como su posición y orientación iniciales e ir calculando la posición y orientación actuales conforme se va moviendo. Una desventaja de este método es que es sensible a errores acumulativos, por lo que usualmente este método es acompañado de otros métodos de localización, tales como el *GPS*, o el *SLAM*.

En este reto se hizo uso de un archivo *URDF (Unified Robot Description Format)*, este formato es un estándar de descripción de robots empleado en ROS. Se usa en formato XML y proporciona una representación estructurada y detallada de la geometría, cinemática, y otros detalles importantes de un robot, este facilita su simulación, control y visualización en entornos como RViz y Gazebo.

Este tipo de archivos se encuentra compuesto de lo siguiente:

- Robot: Como nodo raíz del archivo URDF, el robot define el nombre del robot y encapsula todos los elementos que componen su estructura.
- Enlaces: Representan las partes físicas del robot, como el cuerpo, los brazos o las ruedas. Cada enlace puede incluir propiedades visuales, inerciales y de colisión.

- Articulaciones: Definen las relaciones cinemáticas entre los enlaces y permiten el movimiento relativo entre ellos. Existen varios tipos de articulaciones, como revolutas (rotacionales), prismáticas (traslacionales) y fijas.
- Orígenes: Especifican la posición y orientación relativa de un enlace o una articulación con respecto a su marco de referencia padre.
- Geometría: Define la forma visual de un enlace mediante primitivas geométricas simples (como cubos, esferas o cilindros) o mediante mallas 3D más complejas.
- Inercias: Describen la inercia de un enlace, lo que es crucial para simular el comportamiento dinámico del robot.
- Límites: Permiten especificar límites en los rangos de movimiento, velocidades y esfuerzos máximos permitidos para las articulaciones.
- Gestión de enlaces y articulaciones: Los enlaces pueden conectarse entre sí mediante articulaciones, lo que define la estructura cinemática del robot. La jerarquía de enlaces y articulaciones determina la configuración cinemática del robot y su capacidad de movimiento.

Solución del problema

Para el desarrollo de este reto se decidió implementar la estructura planteada por *Manchester Robotics* (imagen 1). En esta arquitectura planteada se buscaba agregar a los conocimientos obtenidos con el socio formador el año pasado.

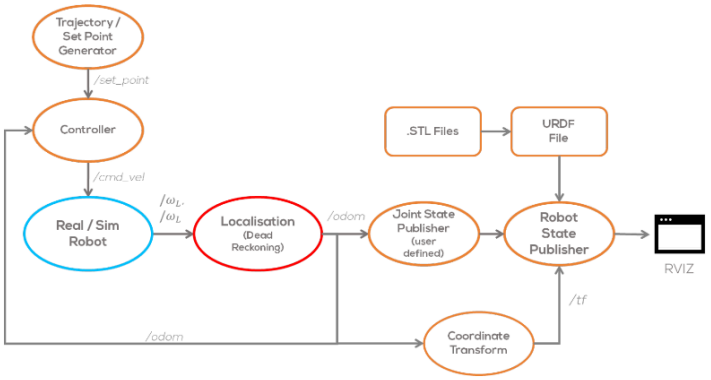

Imagen 1. Arquitectura proporcionada como solución al reto.

SetPoint:

La arquitectura consiste en obtener una ruta mediante el nodo SetPoint, este nodo únicamente consiste en que a partir de un mensaje personalizado de ROS, se mandaba una lista de las coordenadas x,y de los puntos de trayectoria deseados, las coordenadas son mandadas al nodo *Controller* mediante el mensaje personalizado SetPoint. En el desarrollo de este nodo se tuvieron complicaciones, al usar un mensaje personalizado se generaron problemas al momento de la elaboración de los documentos *CmakeList* y *package.xml* debido a esto y a la gran cantidad de errores que no permitían el uso del archivo *SetPoint.msg* ni su inclusión en los archivos previamente mencionados es que se optó por dejar este nodo de lado, modificando el nodo *Controller* que en lugar de recibir el mensaje *SetPoint* las listas de coordenadas se vuelven atributos del nodo *Controller*. A pesar de este inconveniente, se puede observar el nodo realizado original que inicialmente publica el mensaje mediante un publicador, el código se encuentra en el apéndice **Setpoint**. Este código manda en el mensaje la trayectoria para hacer un cuadrado.

Controller:

En el nodo *Controller* es que a partir de las coordenadas se obtienen valores de distancia y orientación lo que permitirá el cálculo de errores, se realizan estimaciones del error lineal y angular a partir de la posición actual del robot respecto al punto deseado (elemento de la lista de coordenadas mandadas por el nodo *SetPoint*). A partir de los errores es que se desarrolla un controlador PID, las constantes utilizadas para este controlador se volvieron a usar los valores utilizadas anteriormente con el socio formador.

En la función *controller* del nodo (apéndice **Controller**) es que se evalúa el error lineal y angular del robot a partir de la posición actual y la posición deseada, se genera un error acumulado de cada tipo de error, se generan los componentes del controlador utilizando los valores de las constantes (son un set de 3 constantes por cada error) creando así una acción proporcional, integral y derivativa, se juntan las 3 creando asi un controlador PID lineal y uno angular.

Otra función importante del nodo es que es un nodo Publicador, el objetivo de este nodo de acuerdo a la arquitectura planteada por *manchester Robotics* es que el nodo publique un mensaje de tipo *Twist llamada* cmd_vel, de este mensaje los únicos parámetros necesarios son la velocidad lineal en el eje x y la velocidad angular en el eje z (debido a que el *Puzzlebot* es un robot no holónomo estos son los únicos grados de libertad que tiene), estos 2 parámetros son asignados al mensaje *Twist* y enviados mediante el publicador del nodo al siguiente nodo.

Real / Sim Robot:

Este nodo es una simulación del sistema dinámico del robot (imagen 2a), este nodo es el encargado de recibir la velocidad lineal y angular del nodo *Controller*, calcular las velocidades angulares de cada llanta del robot, a partir de la velocidad de cada llanta obtener velocidad linear y angular (v, w) (ecuaciones 4 y 5), a partir de estas velocidades es que se pueden desarrollar **Suposiciones de Markov** (imagen 3)**.** Estas suposiciones indican que la pose actual del robot depende únicamente de la pose anterior y de las velocidades de entrada, estos valores de la pose se modifican en la función *update_Pose*.

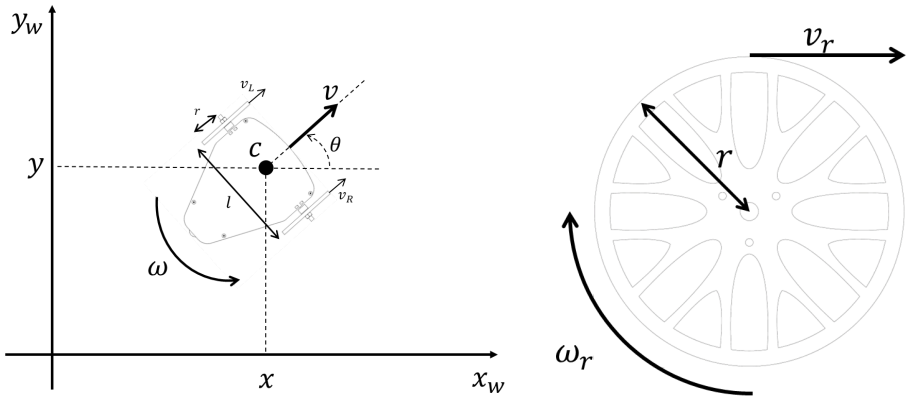

Imagen 2a y 2b. Modelo dinámico del Puzzlebot.

El nodo es un publicador y suscriptor, mediante la función *cmd_vel callback* es que se extraen los datos del nodo *Controller* siendo las 2 velocidades, el nodo es el encargado asimismo de publicar las velocidades angulares de las 2 ruedas, para esto se utilizan mensajes *Float32* y un mensaje de tipo *PoseStamped* que contiene la pose del robot (posiciones y orientaciones), para culmplir con las especificaciones del socio formador (imagen 1) solo se utilizaron los mensajes de las velocidades angulares que serían utilizadas en el nodo *Localisation.*

$$\begin{bmatrix} s_{x,k} \\ s_{y,k} \\ s_{\theta,k} \end{bmatrix} = \begin{bmatrix} s_{x,k-1} \\ s_{y,k-1} \\ s_{\theta,k-1} \end{bmatrix} + \begin{bmatrix} \Delta d \cos\left(s_{\theta,k-1}\right) \\ \Delta d \sin\left(s_{\theta,k-1}\right) \\ \Delta\theta \end{bmatrix}$$

Imagen 3. Matrices de Posición - Suposiciones de Markov

Localisation:

Este nodo se encarga de recibir del nodo *Real / Sim Robot* las velocidades angulares, su objetivo es el publicar la odometría del robot mediante un mensaje de tipo *Odometry.*

Su primera función es el cálculo de las velocidades lineal y angular (v,w) a partir de las velocidades angulares de las llantas, para esto se utilizan las ecuaciones (1) y (2).

$$v = r * \frac{wl + wr}{2} \quad (1)$$

$$w = r * \frac{wl + wr}{l} \quad (2)$$

A partir de las velocidades es que se vuelven a calcular las posiciones para odometría utilizando las suposiciones de Markov (imagen 3) obteniendo nuevamente las posiciones de x, y, theta.

En este nodo se elaborar un mensaje de tipo *Odometry,* este es el tipo de mensaje que se publicará. Este mensaje contiene en su interior mensajes de tipo *PoseStamped* y *Twist,* siendo asi que este mensaje contiene las configuraciones de velocidad y posición. Al contener todos estos parámetros en la declaración de atributos en la función *calculate_pose* (apéndice **Localisation**) es que para los parámetros de posición ya estan declarados los valores de x,y mediante las suposiciones de Markov, para las orientaciones es que utilizamos la posición en therta (orientación) en el que se realizan transformaciones a partir de los angulos de Euler (*yaw, pitch, roll*) a orientaciones, para esto se utiliza la función *quaternion_from_euler*. Esta función genera un cuaternión (vector de 4 espacios) que almacenará las rotaciones del sistema, como la entrada de esta función son ángulos de Euler, la posición en theta sirve como el angulo *yaw*. Debido a que es un robot no holónomo es que únicamente tiene una orientación es por esto que en este sistema no existen los otros ángulos de Euler.

Finalmente para la conformación del mensaje *Odometry* se tienen que agregar las velocidades, estas son designadas con las ecuaciones (1) y (2) anteriormente descritas, tras la

conformación del paquete es que este mensaje sera publicado, como se mencionó anteriormente, al tener este mensaje valores de velocidad linear y angular, orientaciones, y posiciones es que se mandaría prácticamente el modelo dinámico del robot por completo.

Robot_State_Publisher:

Este nodo intento encapsular los nodos de la arquitectura del socio formador (imagen 1) *Joint_State_Publisher* y *Coordinate_Transform* en un solo nodo utilizando el mensaje tipo *Odomnetry* para crear transformadas necesarias para la simulación además de incorporar el modelo 3D del robot en el ambiente simulado *RVIZ.*

Este nodo se suscribe al mensaje Odometry es que podemos extraer las configuraciones de velocidad, posición y orientaciones, todos estos valores se extraen en la función *odom_callback* (apéndice **Robot State Publisher)**. Con los datos inicializados es que se manda a llamar la función *broadcast_transform*. Esta función se encarga de crear un mensaje de tipo *TransformStamped* que reune los valores de traslación y de rotación necesarios para la simulación además de establecer los frame padre e hijo necesarios para la simulación utilizando el archivo URDF definiendo el identificador de cada uno de los links utilizados para la simulación. Con este mensaje creado es que se usa la función *sendTransform()* que se encarga de enviar este mensaje funcionando asi como un publicador enviando asi traslaciones y rotaciones al simulador *RVIZ.*

Archivo URDF y simulacion RVIZ:

Para la creación del modelo 3D del robot fue necesaria la creación de un archivo URDF que permitiera unir todos los modelos *.stl* proporcionados por el socio formador (imagen 4a y 4b). Las uniones de estos modelos estan definidas en el archivo URDF.

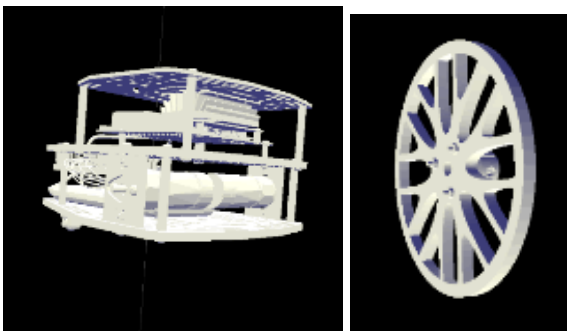

Imagen 4a y 4b. Modelos 3D proporcionados por el socio formador.

Los modelos son definidos como links mientras que las conexiones entre los *links* son conocidos como *joints*. Cada una de las uniones tiene la propiedad de enlazar 2 links declarando como padre e hijo teniendo asi un enfoque de herencia similar a programación orientada a objetos, permitiendo definir asi los cambios en rotación o traslación de los *links* heredando estos cambios y asi afectando a todos los cuerpos del sistema. Debido a que el marco de referencia de los ejes de rotación de cada *link* era diferente es necesario realizar rotaciones, las rotaciones (en angulos de Euler) se muestran en la imagen 5, es a partir de esta guia que se realizan las rotaciones para hacer coincidir los marcos de referencia de acuerdo al tipo de *joint* que tiene cada cuerpo y asi lograr que el sistema quede simulado correctamente.

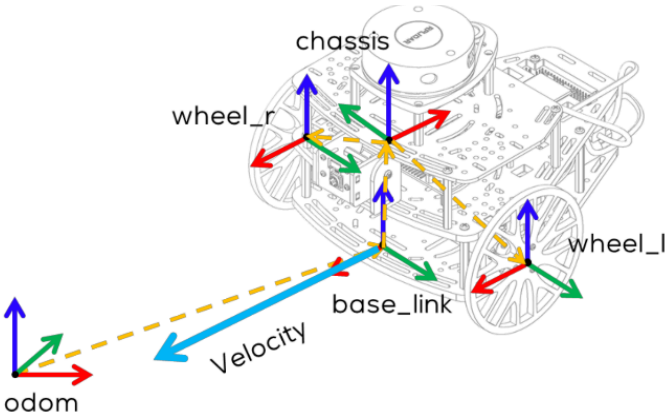

Imagen 5. Guia de orientaciones para los modelos 3D.

Resultados

Tras presentar toda la arquitectura planteada es que se muestran los resultados de la simulación implementando la arquitectura anteriormente planteada. Esta se puede apreciar en la imagen 6.

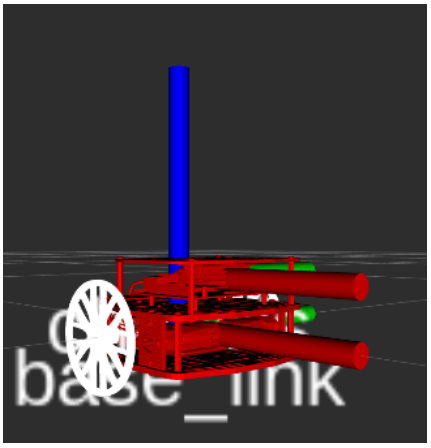

Imagen 6. Resultados de la simulación del proyecto.

Se puede observar en la simulación que la simulación funciona parcialmente, el principal problema es la falla en la conexión de las ruedas del robot con su chassis, generando tambien el problema en el que no se reconocen las transformadas para los *links* de ambas llantas. Debido a la limitación de tiempo es que ya no se pudo indagar mas a profundidad el origen de la problemática del sistema. Algunas conjeturas que se realizaron tras un analisis exhaustivo es falla enj el archivo URDF al nop asignar correctamente las posiciones de los links de las llantas. Además durante el desarrollo del proyecto surgieron diversos contratiempos que consumieron bastante tiempo en su resolución, estos vendrian sie3ndo la falla del sistema de mensajes para la creación del mensaje personalizado *SetPoint,* la falla del sistema de control del sistema optando incluso por una alternativa usando unicamente un controlador PI modificando todo el sistema de control, reestructuración de la arquitectura buscando

simplificarla, problemas con Rates de ROS con incompatibilidad de muestreo y falta de conocimientos de las funciones *Transform_Stamped* y *JointState.* Como objetivos o metas a cortos plazo después de la deadline se busca el desarrollo y finalización de este proyecto buscando obtener los resultados esperados por el socio formador haciendo un análisis exhaustivo de los fallos encontrados y proponiendo alternativas de solución que resuelvan los contratiempos encontrados.

Conclusiones

Tras la finalización del reto se lograron realizar los objetivos relacionados a la simulación en RVIZ gracias a la transformada que se realizó con los mensajes de odometría. A pesar del éxito obtenido en ese aspecto, la conexion y comunicacion entre los nodos y el URDF no se logró finalizar de la manera deseada debido a diversos factores, principalmente por la poca experiencia con la que se inició este reto en el manejo de dichos archivos y de la misma manera por la dificultad de las transformaciones, lo cual requirió más tiempo del esperado. Una posible solución para alcanzar todos los objetivos sería una mejor comunicación entre los miembros del equipo para generar una mejor distribución de las actividades, considerando priorizar las tareas que se requieren para la resolución del reto.

Apéndice

Controller

```
#!/usr/bin/env python3

import rospy
import numpy as np
import math
from geometry_msgs.msg import Twist
from nav_msgs.msg import Odometry
```

```
#from localisation.msg import SetPoint

class ControlNode:
   def __init__(self):
       #self.flag_SetPoint = False
       #self.flag_pricipal = True

         #self.SetPoint_sub = rospy.Subscriber('/set_point', SetPoint,
self.set_point_callback)
               self.odom_sub  =  rospy.Subscriber('/odom',  Odometry,
self.odom_callback)
       self.cmd_pub = rospy.Publisher('/cmd_vel', Twist, queue_size=10)

       #Physical attributes of the Puzzlebot
       self.wheel_base = 0.19
       self.wheel_radio = 0.05

       self.kd_ang = rospy.get_param("angular_kd", 0.1)
       self.kp_ang = rospy.get_param("angular_kp", 0.3)
       self.ki_ang = rospy.get_param("angular_ki", 0.2)

       self.kd_lin = rospy.get_param("linear_kd", 0.1)
       self.kp_lin = rospy.get_param("linear_kp", 0.4)
       self.ki_lin = rospy.get_param("linear_ki", 0.2)

       self.position = 0

       self.last_med_ang = 0.0
       self.last_med_lin = 0.0

       self.error_prev_ang = 0.0
       self.error_prev_lin = 0.0

       self.error_acc_ang = 0.0
       self.error_acc_lin = 0.0

       self.error_ang = 0.0
       self.error_lin = 0.0

       self.actual_x = 0
       self.goal_x = [0, 2, 2, 0]
       self.actual_y = 0
       self.goal_y = [2, 2, 0, 0]
```

```
        #Pose attributes for Localisation
        self.pose_x = rospy.get_param("pose_x", 0.0)
        self.pose_y = rospy.get_param("pose_y", 0.0)
        self.pose_z = 0.0
        self.pose_theta = rospy.get_param("pose_theta", 0.0)

        #Time meditions
        self.dt = rospy.get_param("~dt",0.001)

        #Parameter info for Publishing
        self.msg = Twist()
        self.msg.linear.x = 0
        self.msg.linear.y = 0
        self.msg.linear.z = 0
        self.msg.angular.x = 0
        self.msg.angular.y = 0
        self.msg.angular.z = 0


    #def set_point_callback(self, msg):
        #self.goal_x = msg.x
        #self.goal_y = msg.y
        #self.position = msg.index
        #self.flag_set_point = True

    def odom_callback(self, msg):
        self.pose_x = msg.pose.pose.position.x
        self.pose_y = msg.pose.pose.position.y
        self.pose_theta = msg.pose.pose.orientation.z

    def wrap_to_pi(self, theta):
        result=np.fmod((theta+ np.pi),(2*np.pi))
        if(result<0):
            result += 2 * np.pi
        return result - np.pi

    def controller(self):
        error_ang = 0.0
        error_lin = 0.0

        self.error_prev_ang = error_ang
        self.error_prev_lin = error_lin
```

```
                                                    error_ang        =
self.wrap_to_pi((math.atan2(float(self.goal_y[self.position]),
float(self.goal_x[self.position])))  -  self.pose_theta)
       error_lin =np.sqrt(np.square(float(self.goal_x[self.position]) -
self.pose_x)     +     np.square(float(self.goal_y[self.position])     -
self.pose_y))

       self.error_acc_ang += error_ang * self.dt
       self.error_acc_lin += error_lin * self.dt

       #Components of PID controller
       accion_proporcional_ang = self.kp_ang * error_ang
       accion_proporcional_lin = self.kp_lin * error_lin

       accion_integral_ang = self.ki_ang * self.error_acc_ang
       accion_integral_lin = self.ki_lin * self.error_acc_lin

       accion_derivativa_ang = self.kd_ang * self.error_acc_ang
       accion_derivativa_lin = self.kd_lin * self.error_acc_lin

          control_ang= accion_proporcional_ang + accion_integral_ang +
accion_derivativa_ang
          control_lin= accion_proporcional_lin + accion_integral_lin +
accion_derivativa_lin

                   vel_ang   =   self.last_med_ang   +   ((control_ang   -
self.last_med_ang))
                   vel_lin   =   self.last_med_lin   +   ((control_lin   -
self.last_med_lin))

       self.last_med_ang = vel_ang
       self.last_med_lin = vel_lin

       self.msg.angular.z = vel_ang

       #Measurements for error correction
       #Below the threshold its considered error free
       if error_ang < 0.02:
           self.msg.linear.x = vel_lin

       if error_lin < 0.05 :
           self.msg.linear.x = 0.0
```

```
            self.msg.angular.z = 0.0
            self.position += 1
            self.pose_x=0.0
            self.pose_y=0.0

if __name__=='__main__':

    rospy.init_node("ControlNode")
    CN = ControlNode()

    try:
        while not rospy.is_shutdown():
            loop_rate = rospy.Rate(100)

            CN.controller()

            CN.cmd_pub.publish(CN.msg)
            loop_rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

Localisation

```
import rospy
import math
import numpy as np
from std_msgs.msg import Float32
from nav_msgs.msg import Odometry
from tf.transformations import quaternion_from_euler

class Localisation:
    def __init__(self):
        #Declaration of the delta Time
        self.dt = rospy.get_param("~dt",0.001)

        #Declaraqtion of Publishers and Subscribers of the Node
        self.wl_sub = rospy.Subscriber("/wl", Float32, self.wl_callback)
        self.wr_sub = rospy.Subscriber("/wr", Float32, self.wr_callback)
                    self.odom_pub   =   rospy.Publisher("/odom",   Odometry,
queue_size=10)
```

```
        #Init Puzzlebot wheel velocities
        self.wl = rospy.get_param("wl", 0.0)
        self.wr = rospy.get_param("wr", 0.0)

        #Wheel Attributes / Physical Robot
        self.wheel_radio = 0.05
        self.wheel_base = 0.19

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

        #Pose attributes for Localisation
        self.pose = Odometry()
        self.pose.header.stamp = rospy.Time.now()
        self.pose.header.frame_id = "odom"
        self.pose.pose.pose.position.x = rospy.get_param("pose_x", 0.0)
        self.pose.pose.pose.position.y = rospy.get_param("pose_y", 0.0)
        self.pose.pose.pose.position.z = 0.0
                                    self.pose.pose.pose.orientation.z     =
rospy.get_param("pose_theta", 0.0)

        self.pose.twist.twist.linear.x = 0.0
        self.pose.twist.twist.angular.z = 0.0

    def calculate_pose(self):
        #Calculation of the linear and angular velkocities
        v = self.wheel_radio * (self.wl + self.wr) / 2.0
        w = self.wheel_radio * (self.wr - self.wl) / self.wheel_base

        #Actualization of position and coordinates
        self.theta += self.wrap_to_pi(self.theta + w * self.dt)
        self.theta = np.arctan2(np.sin(self.theta), np.cos(self.theta))

        self.x += v * math.cos(self.theta) * self.dt
        self.y += v * math.sin(self.theta) * self.dt

        self.pose.header.stamp = rospy.Time.now()
        self.pose.header.frame_id = "odom"
        self.pose.pose.pose.position.x = self.x
        self.pose.pose.pose.position.y = self.y
        quaternion = quaternion_from_euler(0, 0, self.theta)
```

```
        self.pose.pose.pose.orientation.x = quaternion[0]
        self.pose.pose.pose.orientation.y = quaternion[1]
        self.pose.pose.pose.orientation.z = quaternion[2]
        self.pose.pose.pose.orientation.w = quaternion[3]

        self.pose.twist.twist.linear.x = v
        self.pose.twist.twist.angular.z = w

        self.odom_pub.publish(self.pose)

    def wl_callback(self, msg):
        self.wl = msg.data

    def wr_callback(self, msg):
        self.wr = msg.data

    def wrap_to_pi(self,theta):
        result = np.fmod((theta + np.pi),(2 * np.pi))
        if(result < 0):
            result += 2 * np.pi
        return result - np.pi

if __name__=='__main__':
    #Initialise and Setup node
    rospy.init_node("Localisation")
    LOC = Localisation()

    loop_rate = rospy.Rate(100)

    try:
        while not rospy.is_shutdown():
            LOC.calculate_pose()
            loop_rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

PuzzleBot Kinematic Model

```
#!/usr/bin/env python
import rospy
import math
```

```
from geometry_msgs.msg import Twist, PoseStamped
from std_msgs.msg import Float32

class Real_Sim_Robot:
   def __init__(self):
       self.dt = rospy.get_param("~dt",0.001)
       #Physical attributes of the Robot
       self.wheel_radio = 0.05
       self.wheel_base = 0.19

       #Publishers and Subscribers
                  self.cmd_sub   =   rospy.Subscriber('/cmd_vel',   Twist,
self.cmd_vel_callback)
                self.pose_pub   =   rospy.Publisher('/pose',   PoseStamped,
queue_size=10)
       self.wl_pub = rospy.Publisher('/wl', Float32, queue_size=10)
       self.wr_pub = rospy.Publisher('/wr', Float32, queue_size=10)

       # Init Robot pose attributes
       self.robot_pose = PoseStamped()
       self.robot_pose.pose.position.x = 0.0
       self.robot_pose.pose.position.y = 0.0
       self.robot_pose.pose.orientation.z = 0.0
       self.robot_pose.pose.orientation.w = 1.0

       #Init Puzzlebot wheel velocities
       self.wl = rospy.get_param("wl", 0.0)
       self.wr = rospy.get_param("wr", 0.0)

   def cmd_vel_callback(self, msg):
        # Calcular las velocidades angulares de las ruedas izquierda y
derecha
       linear_vel = msg.linear.x
       angular_vel = msg.angular.z

        self.wl = (2.0 * linear_vel - angular_vel * self.wheel_base) /
(2*self.wheel_radio)
        self.wr = (2.0 * linear_vel + angular_vel * self.wheel_base) /
(2*self.wheel_radio)

       self.update_Pose()

   def update_Pose(self):
```

```
        v = self.wheel_radio * (self.wl + self.wr) / 2.0
        w = self.wheel_radio * (self.wr - self.wl) / self.wheel_base

                            self.robot_pose.pose.position.x    +=    v    *
math.cos(self.robot_pose.pose.orientation.z) * self.dt
                            self.robot_pose.pose.position.y    +=    v    *
math.sin(self.robot_pose.pose.orientation.z) * self.dt
        self.robot_pose.pose.orientation.z += w * self.dt

    def publish_robot_state(self):
        self.pose_pub.publish(self.robot_pose)
        self.wl_pub.publish(self.wl)
        self.wr_pub.publish(self.wr)


if __name__ == '__main__':
    rospy.init_node("Real_Sim_Robot")
    RSR = Real_Sim_Robot()
    loop_rate = rospy.Rate(100)  # 10 Hz
     try:
        while not rospy.is_shutdown():
            # Publish Pose and Wheel Velocities
            RSR.publish_robot_state()
            loop_rate.sleep()

    except rospy.ROSInterruptException:
        pass
```

Robot State Publisher

```
#!/usr/bin/env python3
import rospy
import tf2_ros
import numpy as np
import math

from geometry_msgs.msg import TransformStamped, Quaternion
from tf.transformations import quaternion_from_euler
from nav_msgs.msg import Odometry
```

```
from sensor_msgs.msg import JointState

class RobotStatePublisher:
   def __init__(self):

       #Robot's Position
       self.pose_x = rospy.get_param("pose_x", 0.0)
       self.pose_y = rospy.get_param("pose_y", 0.0)
       self.pose_z = rospy.get_param("pose_z", 0.0)
       self.pose_theta = rospy.get_param("pose_theta", 0.0)

       #Roboty's Velocities
       self.v = rospy.get_param("vel_lin", 0.0)
       self.w = rospy.get_param("vel_ang", 0.0)

       self.broadcaster = tf2_ros.TransformBroadcaster()

                self.odom_sub   =   rospy.Subscriber('/odom',   Odometry,
self.odom_callback)

         self.joint_pub  =  rospy.Publisher("/joint_states",  JointState,
queue_size=10)

   def odom_callback(self, msg):
       self.pose_x = msg.pose.pose.position.x
       self.pose_y = msg.pose.pose.position.y
       self.pose_z = msg.pose.pose.position.z
       self.pose_theta = msg.pose.pose.orientation.z

       #Calculation of Linear and Angular velocities using Odomeetry
       linear_velocity_x = msg.twist.twist.linear.x
       linear_velocity_y = msg.twist.twist.linear.y
       angular_velocity = msg.twist.twist.angular.z

       # Velocities using Odometry attributes
       self.v = math.sqrt(linear_velocity_x**2 + linear_velocity_y**2)
       self.w = angular_velocity

       quaternion = [0.0, 0.0, 0.0, 0.0]
       quaternion[0] = msg.pose.pose.orientation.x
       quaternion[1] = msg.pose.pose.orientation.y
       quaternion[2] = msg.pose.pose.orientation.z
       quaternion[3] = msg.pose.pose.orientation.w
```

```
        self.broadcast_transform(self.pose_x, self.pose_y, self.pose_z,
quaternion)

    def broadcast_transform(self, x, y, z, quaternion):

        t = TransformStamped()
        t.header.stamp = rospy.Time.now()
        t.header.frame_id = "odom"
        t.child_frame_id = "chassis"

        t.transform.translation.x = x
        t.transform.translation.y = y
        t.transform.translation.z = z
        t.transform.rotation.x = quaternion[0]
        t.transform.rotation.y = quaternion[1]
        t.transform.rotation.z = quaternion[2]
        t.transform.rotation.w = quaternion[3]

        self.broadcaster.sendTransform(t)

    def wrap_to_pi(self, theta):
        result=np.fmod((theta+ np.pi),(2*np.pi))
        if(result<0):
            result += 2 * np.pi
        return result - np.pi

if __name__ =='__main__':
    #Initialise and Setup node
    rospy.init_node("RobotStatePublisher")
    RobotStatePublisher()
```

SetPoint

```
!/usr/bin/env python
import rospy
from localisation.msg import SetPoint

class SetPointNode:
    def _init_(self):
```

```
            self.cmd_pub  =  rospy.Publisher('/set_point',  SetPoint,
queue_size=10)
       self.SetPoint = SetPoint()
       self.SetPoint.list_x = [0]
       self.SetPoint.list_y = [0]
       self.SetPoint.index = len(self.SetPoint.x)

if _name_ == '_main_':

   rospy.init_node("SetPointNode")
   SPN = SetPointNode()
   SPN.SetPoint.x = [0, 2, 2, 0]
   SPN.SetPoint.y = [2, 2, 2, 0]
   SPN.SetPoint.index = len(SPN.SetPoint.x)
   SPN.cmd_pub.publish(SPN.SetPoint)

   rospy.spin()
```

Bibliografía o referencias

Linus, N., Clemens, F., Bartsch, K., & Rueckert, E. (2023). ROMR: A ROS-based open-source mobile robot. *HardwareX*, *14*, e00426. https://doi.org/10.1016/j.ohx.2023.e00426

urdf - ROS Wiki. (s. f.-b). http://wiki.ros.org/urdf

Valencia, J. A., V., O, A. M., & Rios, L. H. (2009). MODELO CINEMÁTICO DE UN ROBOT MÓVIL TIPO DIFERENCIAL y NAVEGACIÓN a PARTIR DE LA ESTIMACIÓN ODOMÉTRICA. Redalyc.org. https://www.redalyc.org/articulo.oa?id=84916680034

Edisonsasig. (2023, 14 julio). Modelo cinemático y simulación de un robot móvil diferencial. Roboticoss. https://roboticoss.com/modelo-cinematico-y-simulacion-con-python-robot-movil-diferencial/

Joseph, A. (2014). The History of Measuring Ocean Currents. En Elsevier eBooks (pp. 51-92). https://doi.org/10.1016/b978-0-12-415990-7.00002-8