Instituto Tecnológico y de Estudios Superiores de Monterrey Campus Puebla

TE3003B.501

Integración de robótica y sistemas inteligentes

Reto Semanal 5 | Manchester Robotics

Frida Lizett Zavala Pérez A01275226

Diego Garcia Rueda A01735217

Alejandro Armenta Arellano A01734879

3 de Mayo del 2024

Objetivos

Mediante el proceso de obtención de incertidumbres gaussianas para *Puzzlebot*, un sistema no lineal, a partir de la técnica de linealización alrededor de un punto operativo específico. Los mecanismos se utilizan para calcular la covarianza, considerando mediciones de incertidumbres gaussianas, proporcionando una herramienta esencial para comprender y predecir la variabilidad y precisión del sistema. Además, se describe la arquitectura ROS utilizada, incluyendo la implementación de una arquitectura de bucle abierto para obtener las constantes necesarias y la publicación de posiciones entre el modelo ideal y real de *Puzzlebot*. Los resultados obtenidos muestran la eficacia de la aproximación y la validez de las incertidumbres calculadas, destacando la importancia de este proyecto como base para futuros desarrollos.

Resumen

En este reto se generó una optimización en la localización del Puzzlebot, a partir de la realización de diversos experimentos, del movimiento del robot dónde se hicieron mediciones en lazo abierto con la finalidad de afinar las constantes necesarias para el modelo a partir de parametrizaciones, esto en diversos escenarios para considerar factores diversos. Igualmente se trazó una elipsoide de confidencia con el propósito de evaluar la precisión y estabilidad del robot.

Matriz de covarianza para localización

La matriz de covarianza es una herramienta estadística que permite entender cómo se relacionan diferentes variables entre sí, podría verse como una tabla que muestra las covarianzas entre las parejas posibles de variables, ayuda a entender la relación entre más de dos variables de manera eficiente.

En el contexto de localización la matriz de covarianza juega un papel importante para estimar la incertidumbre asociada a la posición y orientación del robot. Describe como la incertidumbre en una variable se relaciona con la incertidumbre de otras, por ejemplo la posición en el eje x con con la posición en el eje y o la orientación. Sí se cuenta con una matriz de covarianza calibrada correctamente se puede utilizar para calcular la propagación de la incertidumbre a través de las diferentes variables del sistema. La relación entre las variables del modelo ayuda a tener una comprensión más completa de la incertidumbre global del sistema, lo que es crucial para la toma de decisiones y la planificación de los pasos a seguir para la localización del robot consiguiendo una navegación más precisa y segura.

Mensajes tipo odometry y Pose Messaje

Estos tipos de mensaje se utilizan para representar la informaciónd de la odometria de un robot móvil, en ROS el mensaje se define como ***nav_msgs/Odometry***, y contiene información sobre la posición y orientación estimadas del robot, así como su velocidad angular y lineal y la covarianza de estas. Al representar y publicar esta información permite que los otros nodos puedan usarla para navegación y planificación de trayectorias.

El mensaje tipo Pose Message es empleado para representar la posición y orientación del robot en un espacio 3D usando un quaternión, el cual es una extensión de los números complejos y se usa para representar las rotaciones que tenga el robot. El mensaje es parte del paquete ***geometry_msgs*** y se define como ***Pose.msg.***

Elipsoide de confianza

Una elipsoide de confianza sirve para representar la incertidumbre en la posición y orientación de un robot, esta se basa en la matriz de covarianza explicada anteriormente. En la elipsoide se muestra visualmente cómo esta incertidumbre se propaga a lo largo del plano, lo que explica de cierto modo la precisión y fiabilidad de la estimación de la posición del robot. Entre mayor sea, maypr será la incertidumbre, por lo que sí está es más pequeña, indica una estimación más precisa y confiable. Sin embargo es esperado que esta elipsoide se vaya propagando a lo largo del tiempo, creciendo en mayór proporción con las constantes angulares.

Solución del problema

Obtención de incertidumbres Gaussianas:

Gracias a que la linealización nos permite simplificar un sistema no lineal alrededor de un punto de operación específico es posible aproximar el comportamiento no lineal del *puzzlebot* mediante un modelo lineal en el entorno de la posición y orientación. Gracias a la linealización y a las fuentes de error en las mediciones mejor conocidas como incertidumbres gaussianas , es posible calcular la covarianza mediante la siguiente ecuación:

$$\Sigma_K = H_K \cdot \Sigma_{K-1} \cdot H_K^T + Q_K \qquad (1)$$

$$\Sigma_K = H_K \cdot \Sigma_{K-1} \cdot H_K^T + \nabla_{wk} \cdot \Sigma_{\Delta k} \cdot \nabla_{wk}^T \qquad (2)$$

Las ecuaciones (1) y (2) representan el cálculo para obtener la covarianza en una matriz 3x3 que representa una suma de dos multiplicaciones de 3 matrices siendo la principal diferencia que en la ecuación (1) se pone Q_K como variable y en la ecuación (2) se desglosa esa variable representando que $Q_K = \nabla_{wk} \cdot \Sigma_{\Delta k} \cdot \nabla_{wk}^T$, siendo el resto de variables matrices de diferentes dimensiones generando la siguiente operación de matrices:

$$\Sigma_K = (3x3 \cdot 3x3 \cdot 3x3 + 3x2 \cdot 2x2 \cdot 2x3) \qquad (3)$$

La ecuación (3) representa las dimensiones de las matrices de cada variable cada una representando los siguientes valores:

$$H_K = \begin{bmatrix} 1 & 0 & -\Delta t * V * sin(\Theta)) \\ 0 & 1 & \Delta t * V * cos(\Theta)) \\ 0 & 0 & 1 \end{bmatrix} \tag{4}$$

$$\Sigma_{\Delta k} = \begin{bmatrix} k_r \, |\Delta s_r| & 0 \\ 0 & k_l \, | \, \Delta s_l \end{bmatrix} \tag{5}$$

$$\nabla_{wk} = \frac{1}{2} r \Delta t \begin{bmatrix} \cos(s_{\theta,k-1}) & \cos(s_{\theta,k-1}) \\ \sin(s_{\theta,k-1}) & \sin(s_{\theta,k-1}) \\ \frac{2}{l} & -\frac{2}{l} \end{bmatrix} = \begin{bmatrix} \frac{1}{2}\cos\left(\theta + \frac{\Delta\theta}{2}\right) - \frac{\Delta s}{2b}\sin\left(\theta + \frac{\Delta\theta}{2}\right) & \frac{1}{2}\cos\left(\theta + \frac{\Delta\theta}{2}\right) + \frac{\Delta s}{2b}\sin\left(\theta + \frac{\Delta\theta}{2}\right) \\ \frac{1}{2}\sin\left(\theta + \frac{\Delta\theta}{2}\right) + \frac{\Delta s}{2b}\cos\left(\theta + \frac{\Delta\theta}{2}\right) & \frac{1}{2}\sin\left(\theta + \frac{\Delta\theta}{2}\right) - \frac{\Delta s}{2b}\cos\left(\theta + \frac{\Delta\theta}{2}\right) \\ \frac{1}{b} & -\frac{1}{b} \end{bmatrix} \tag{6}$$

La matriz (4) es el sistema cinético linealizado en el reto previo así como H_K^T representa esa misma matriz transpuesta esas dos matrices se multiplican con el valor previo de Σ_K que se inicializa en una matriz 3x3 con únicamente ceros , del otro lado de la suma tenemos la matriz (6) ∇_{wk} , esa matriz es proporcionada por el socio formador, siendo ∇_{wk}^T la matriz transpuesta , esas dos matrices se multiplican a la matriz (5) donde los valores *kr* y *kl* deben de ser tuneados mediante pruebas a lazo abierto, este proceso es explicado en la siguiente sección. por lo que el primer valor de cada matriz debería de quedar como se muestran en la matriz (7) y la matriz (8):

$$\boldsymbol{\Sigma}_1 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0.1 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix} \cdot \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0.1 & 1 \end{bmatrix} + \begin{bmatrix} 0.5 & 0.01 & 0.01 \\ 0.01 & 0.5 & 0.01 \\ 0.01 & 0.01 & 0.2 \end{bmatrix} \tag{7}$$

$$\boldsymbol{\Sigma}_1 = \begin{bmatrix} 0.5 & 0.01 & 0.01 \\ 0.01 & 0.5 & 0.01 \\ 0.01 & 0.01 & 0.2 \end{bmatrix} \tag{8}$$

Siendo la matriz (8) el primer valor resultante de Σ_K para posteriormente incorporar ese resultado a la ecuación y ahora multiplicar ese valor al resto de variables lo que genera que conforme avanza el tiempo la matriz crezca representando el error que se va acumulando como se muestra a continuación:

$$\mathbf{\Sigma_1} = \begin{bmatrix} 0.5 & 0.01 & 0.01 \\ 0.01 & 0.5 & 0.01 \\ 0.01 & 0.01 & 0.2 \end{bmatrix} \qquad \mathbf{\Sigma_2} = \begin{bmatrix} 0.998 & 0.0207 & 0.0180 \\ 0.0207 & 1.0040 & 0.0399 \\ 0.0180 & 0.0399 & 0.4000 \end{bmatrix} \tag{9}$$

Representando la matriz (9) los valores que se obtienen con respecto a (x,y, Θ) como se muestra a continuación:

$$\mathbf{\Sigma}_k = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta\theta} \end{bmatrix} \tag{10}$$

Los valores de la matriz (10) se utilizaron sustituyendo los mismos valores en la matriz que se manda a odometría como se ve en la siguiente matriz:

$$\Sigma = \begin{bmatrix} \sigma_{xx} & \sigma_{xy} & \sigma_{xz} & \sigma_{x\varphi} & \sigma_{x\psi} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_{yy} & \sigma_{yz} & \sigma_{y\varphi} & \sigma_{y\psi} & \sigma_{y\theta} \\ \sigma_{zx} & \sigma_{zy} & \sigma_{zz} & \sigma_{z\varphi} & \sigma_{z\psi} & \sigma_{z\theta} \\ \sigma_{\varphi x} & \sigma_{\varphi y} & \sigma_{\varphi z} & \sigma_{\varphi\varphi} & \sigma_{\varphi\psi} & \sigma_{\varphi\theta} \\ \sigma_{\psi x} & \sigma_{\psi y} & \sigma_{\psi z} & \sigma_{\psi\varphi} & \sigma_{\psi\psi} & \sigma_{\psi\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_{\theta z} & \sigma_{\theta\varphi} & \sigma_{\theta\psi} & \sigma_{\theta\theta} \end{bmatrix} \tag{11}$$

La matriz (10) es la que se manda a odometría que contiene 6 valores (x,y,z,φ,ψ, Θ) , al realizar movimientos solo en los valores de la matriz (10) el resto de valores se manda el valor de 0.

Todo lo previamente explicado es representado en el nodo llamado kinematic_puzzlebot que es el encargado de realizar los cálculos en la función *elipse_covarianza* (apéndice **Funcion_Covarianza**) para posteriormente mandarlos al nodo de localización y poder publicarlos mediante un mensaje de tipo */pose* modificando su atributo *covariance* a la odometría.

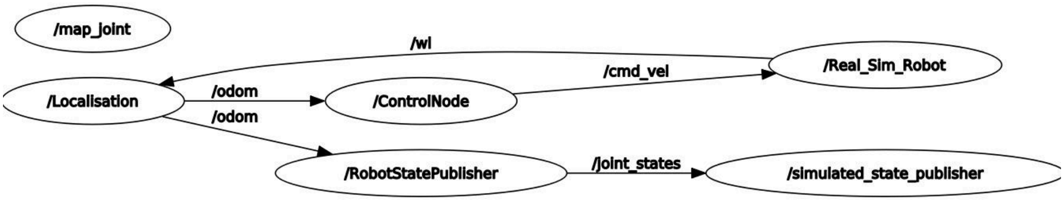

Imagen 1. Gráfico de la arquitectura de ROS del proyecto

Obtención kr y kl:

Para la obtención de estas constantes era necesario la elaboración de una arquitectura de lazo abierto, la arquitectura que se había trabajado durante el desarrollo de los retos del socio formador había sido de lazo cerrado siendo el mensaje */odom* que al conectarse al nodo *ControlNode* el que genera la retroalimentación del sistema de lazo cerrado (Imagen 1). Teniendo esto en mente es que se optó a la creación de una nueva arquitectura únicamente para la obtención de las constantes requeridas, la nueva arquitectura puede apreciarse en la Imagen 2. A diferencia del proyecto anterior se volvió a elaborar una arquitectura de lazo abierto, en esta ocasión se incorpora el puzzle bot mediante una comunicación **SSH**, el diseño de la nueva arquitectura puede observarse en la Imagen 3. Se puede observar que la principal diferencia es que se elimina por completo la utilización de *RVIZ* y por lo tanto el uso de las transformaciones, además, se modificó respecto al anterior proyecto el cálculo de la posición real e ideal del robot, se había mencionado anteriormente que esta aproximación era incorrecta ya que la medición era mediante 2 modelos ideales lo que generaba que no hubiera mucha diferencia entre ambas mediciones, en esta nueva implementación únicamente se calculan las nuevas posiciones reales de acuerdo a las mediciones del *Puzzlebot* (imagen 3) y se comparan con las mediciones obtenidas previamente en el reto anterior. A continuación se detalla el funcionamiento de la arquitectura de lazo abierta implementada.

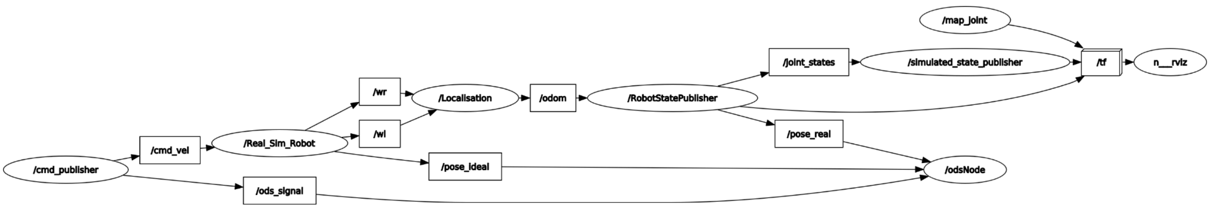

Imagen 2. Gráfico de la arquitectura de ROS de lazo abierto

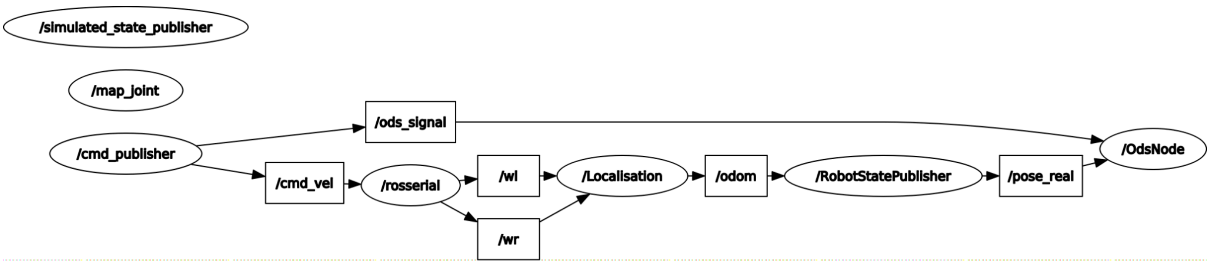

Imagen 3. Gráfico de la nueva arquitectura de ROS de lazo abierto

- CMD Publisher:

Las principales diferencias que se pueden observar respecto a arquitectura de lazo cerrado son la eliminación del nodo de control que deja de ser requerido debido a la nula necesidad de retroalimentación, sin embargo en la arquitectura original el nodo *ControlNode* era requerido ya que este generaba un mensaje de tipo *Twist* que permite al nodo *Real_Sim_Robot* generar las velocidades angulares de cada una de las llantas, al no contar con este nodo se tuvo que generar uno nuevo encargado de generar el mensaje *Twist*.

Este nodo no solamente se limita a la creación del mensaje, recibe parámetros ingresados por el usuario de velocidad (puede ser lineal o angular) y el tiempo que se quiere mantener esta velocidad, esto es necesario para la parametrización de los errores del robot sin retroalimentación, el nodo maneja un rango de tiempo siguiendo el parámetro ingresado, mientras el tiempo transcurrido de la simulación sea menor al tiempo designado, el nodo publicará la velocidad ingresada, al pasar el tiempo designado, se publica un alto asignando a la velocidad un valor de 0. La última funcionalidad de este nodo es la publicación de un mensaje de tipo Booleano que se envía posteriormente al alto del robot, esta bandera es la condicional que indica que el robot se ha detenido,es en este momento que se tiene que realizar el guardado de la posición para la comparación de posición ideal y posición real (este proceso se desarrolla en la siguiente sección).

- Publicación de posiciones:

Al implementar esta arquitectura únicamente en el *Puzzlebot* se busca obtener únicamente la posición final de este, esta posición se encuentra registrada en el tópico */pose_real* que primeramente utiliza la velocidades angulares de cada llantas del *Puzzlebot,* con esas medidas podemos obtener la velocidad lineal y angular del robot (v,w)y finalmente obtener los valores de la posición del robot.

- Nodo almacenador de datos:

Utilizando el nodo *ODSNode* se recibe el mensaje de la posición real utilizando un *callback*, sin embargo, espera la señal enviada por el nodo *CMDNode*, esta señal marca cuando el robot se ha detenido para que sea únicamente esta posición final la que se almacene en txt que el nodo se encarga de escribir o crear en caso de que no se encuentre ningún archivo txt.

- Captación de datos y procesamiento:

Tras la obtención de las posiciones reales son registradas en una hoja de datos, con las condiciones utilizadas en cada caso, recordar que cada caso tiene como variables la velocidad (lineal o angular) y el tiempo de simulación, estas nuevas posiciones reales son comparadas con la posiciones ideales que se obtuvieron en el reto anterior con objetivo de obtener las diferencias entre las posiciones reales y las posiciones ideales en cada escenario propuesto. A continuación se muestra la Tabla (1) que muestra todos los casos de simulación utilizados.

Velocidad Lineal	v	Tiempo
Caso 1	0.5 m/s	2 s
Caso 2	0.5 m/s	5 s
Caso 3	0.7 m/s	2 s
Caso 4	0.3 m/s	2 s
Velocidad Angular	w	Tiempo

Caso 1	0.5 rad/s	3 s
Caso 2	0.5 rad/s	5 s
Caso 3	0.5 rad/s	10 s
Caso 4	0.8 rad/s	3 s
Caso 5	0.2 rad/s	5 s

Tabla 1. Casos utilizados en el proyecto

Es necesario mencionar que uno de los escenarios ya no fue considerado para el cálculo de las constantes, se habla específicamente del caso 3 de velocidad angular, esto es debido a que durante las mediciones se lograba apreciar que la diferencia entre las posiciones angulares del *Puzzlebot* respecto al robot ideal eran de un factor de 0.44., esto quiere decir que aproximadamente si en la simulación la posición del robot era de 1 radian la posición real del robot era de 0.4 radianes, se plantea esto debido a que la función *wrap to pi* utilizada en la mayoría de los nodos de este proyecto al momento de redondear las posiciones angulares de este escenario (aproximadamente de un rango de 5.6 radianes a 6 radianes) redondea mediante una operación módulo con 2π obtiene medidas de posición negativas (estas medidas se pueden apreciar en la hoja de datos), para evitar la modificación de la función de redondeo existente que ya había sido usada para todos los casos previamente mencionados se decidió omitir este escenario, eventualmente se transformaron estos datos quitando la opción del módulo y se pudo observar que los valores obtenidos eran bastante similares específicamente a la covarianza de los datos obtenidos en el Caso 1 y Caso 2 de velocidad angular, es por esta razón que se optó definitivamente a ignorar este escenario.

Códigos empleados para la optimización de las constantes usando el gradiente descendiente, a partir del error de los mínimos cuadrados de los experimentos:

Constante kl

```
import numpy as np
import matplotlib.pyplot as plt

def calcular_velocidad(posicion_final_ideal, posicion_final_real, tiempo):
    # Calcula la diferencia de posición
    diferencia_posicion = posicion_final_real - posicion_final_ideal

    # Calcula la velocidad como la diferencia de posición dividida por el
tiempo
    velocidad = diferencia_posicion / tiempo

    return velocidad

def calcular_error(kl, velocidad_ideal, velocidad_real):
    # Calcula la velocidad de cada rueda
    velocidad_l = kl * velocidad_ideal
    velocidad_r = kl * velocidad_ideal

    # Calcula el error cuadrático
    error = np.mean((velocidad_l + velocidad_r - velocidad_real) ** 2)

    return error

def calcular_gradiente(kl, velocidad_ideal, velocidad_real, epsilon=1e-5):
    # Calcula el gradiente utilizando la definición de derivada
    grad = (calcular_error(kl + epsilon, velocidad_ideal, velocidad_real) -
calcular_error(kl, velocidad_ideal, velocidad_real)) / epsilon

    return grad

def descenso_gradiente_kl(velocidad_ideal, velocidad_real, lr=0.01,
num_iter=100):
    # Inicializa kl
    kl = 1.0

    # Listas para almacenar el historial de valores de kl y error
    kl_history = []
    error_history = []

    # Descenso de gradiente
    for i in range(num_iter):
        # Calcula el error y el gradiente
```

```python
        error = calcular_error(kl, velocidad_ideal, velocidad_real)
        grad = calcular_gradiente(kl, velocidad_ideal, velocidad_real)

        # Actualiza kl usando el gradiente descendente
        kl -= lr * grad

        # Almacena el historial
        kl_history.append(kl)
        error_history.append(error)

    return kl, kl_history, error_history

# Datos de ejemplo
posicion_final_ideal = 2.445330811  # Posición final ideal
posicion_final_real = np.array([2.34674])  # Posición final real (para
múltiples experimentos)
tiempo = 5.0  # Tiempo en segundos

# Calcula la velocidad a partir de las posiciones finales
velocidad_ideal = calcular_velocidad(posicion_final_ideal,
posicion_final_real, tiempo)

# Velocidad real (asumiendo que es la misma que la ideal)
velocidad_real = velocidad_ideal

# Descenso de gradiente para encontrar kl
kl_optimo, kl_history, error_history = descenso_gradiente_kl(velocidad_ideal,
velocidad_real)

print("kl óptimo encontrado:", kl_optimo)

# Gráfico de la convergencia del error
plt.plot(error_history)
plt.xlabel('Iteración')
plt.ylabel('Error')
plt.title('Convergencia del Error')
plt.show()
```

Constante kr

```python
import numpy as np
import matplotlib.pyplot as plt

def calcular_velocidad_angular(posicion_final_ideal, posicion_final_real,
tiempo):
    # Calcula la diferencia de posición angular
    diferencia_posicion_angular = posicion_final_real - posicion_final_ideal
```

```
    # Calcula la velocidad angular como la diferencia de posición angular
dividida por el tiempo
    velocidad_angular = diferencia_posicion_angular / tiempo

    return velocidad_angular

def calcular_error(kr, velocidad_angular_ideal, velocidad_angular_real):
    # Calcula la velocidad angular de cada rueda
    velocidad_angular_l = kr * velocidad_angular_ideal
    velocidad_angular_r = kr * velocidad_angular_ideal

    # Calcula el error cuadrático medio
    error = np.mean((velocidad_angular_l + velocidad_angular_r -
velocidad_angular_real) ** 2)

    return error

def calcular_gradiente(kr, velocidad_angular_ideal, velocidad_angular_real,
epsilon=1e-5):
    # Calcula el gradiente utilizando la definición de derivada
    grad = (calcular_error(kr + epsilon, velocidad_angular_ideal,
velocidad_angular_real) - calcular_error(kr, velocidad_angular_ideal,
velocidad_angular_real)) / epsilon

    return grad

def descenso_gradiente_kr(velocidad_angular_ideal, velocidad_angular_real,
lr=0.01, num_iter=100):
    # Inicializa kr
    kr = 1.0

    # Listas para almacenar el historial de valores de kr y error
    kr_history = []
    error_history = []

    # Descenso de gradiente
    for i in range(num_iter):
        # Calcula el error y el gradiente
        error = calcular_error(kr, velocidad_angular_ideal,
velocidad_angular_real)
        grad = calcular_gradiente(kr, velocidad_angular_ideal,
velocidad_angular_real)

        # Actualiza kr usando el gradiente descendente
        kr -= lr * grad

        # Almacena el historial
```

```
        kr_history.append(kr)
        error_history.append(error)

    return kr, kr_history, error_history
# Datos de ejemplo
posicion_final_ideal = 2.312493295 # Posición final ideal en radianes
posicion_final_real = np.array([1.06419])  # Posición final real en radianes
(para múltiples experimentos)
tiempo = 3.0  # Tiempo en segundos

# Calcula la velocidad angular promedio a partir de las posiciones finales
velocidad_angular_ideal = calcular_velocidad_angular(posicion_final_ideal,
posicion_final_real, tiempo)

# Velocidad angular real (asumiendo que es la misma que la ideal)
velocidad_angular_real = velocidad_angular_ideal

# Descenso de gradiente para encontrar kr
kr_optimo,              kr_history,              error_history              =
descenso_gradiente_kr(velocidad_angular_ideal, velocidad_angular_real)

print("kr óptimo encontrado:", kr_optimo)

# Gráfico de la convergencia del error
plt.plot(error_history)
plt.xlabel('Iteración')
plt.ylabel('Error')
plt.title('Convergencia del Error')
plt.show()
```

Graficación de ambas posiciones

Con las posiciones reales e ideales obtenidas es que se realizó nuevamente modificaciones a la arquitectura del proyecto añadiendo la generación de las posiciones esta vez en lazo cerrado implementando el *Puzzlebot*, de esta manera obteniendo asi un proyecto de ROS que pueda imprimir en RVIZ los 2 robots con posiciones diferentes, igualmente implementando en el robot ideal la elipse de confidencia para así observar los resultados de esta. La nueva arquitectura puede observarse en la Imagen 4.

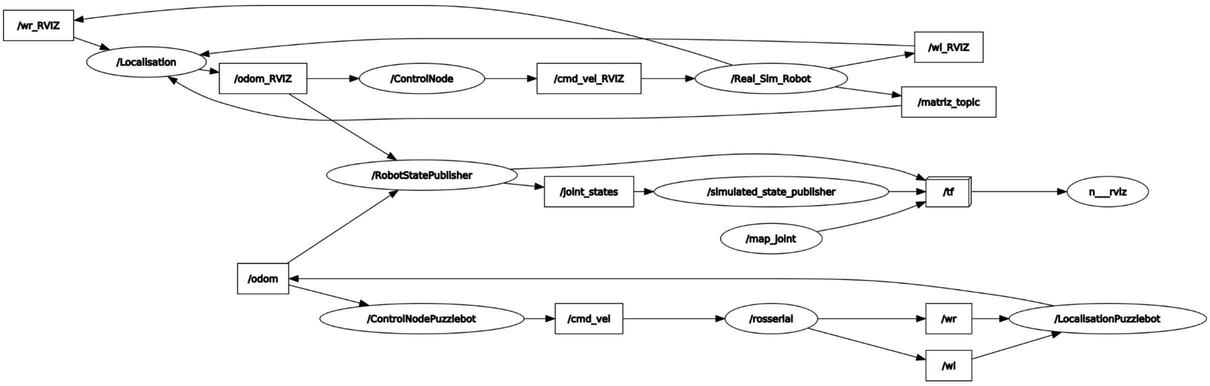

Imagen 4. Gráfico de arquitectura lazo cerrado completa

La mayoría de los nodos suelen repetirse o ser bastante similares, algunas diferencias que se pueden encontrar entre ellos son:

- *LocalisationPuzzlebot* no cuenta con la función que permite la obtención de la matriz de covarianza ya que únicamente el robot ideal tiene que generarla para así saber si la posición del robot real es adecuada o se encuentra dentro de esta matriz.
- *RealSimRobot* no tiene un modo análogo en la sección del proyecto real debida a que las funciones de este nodo (recibir un mensaje *cmd_vel* y la publicación de las velocidades wl y wr) son reemplazadas por el robot físico cumpliendo con estos objetivos.
- Se crea un segundo archivo URDF, estoas no se muestran en la Imagen 4, sin embargo se tuvo que desarrollar un segundo archivo que es una copia del anterior debido a que al momento de publicar en *RVIZ* mediante transformadas es que se llegaba a apreciar que había una sobreescritura ya que se llegaban a repetir el ID de las juntas y de los enlaces lo que ocasionó malfuncionamientos en el simulador llegando asi a la solución de un segundo modelo 3D.

Problemas encontrados

En el desarrollo del mini challenge se encontraron una gran cantidad de problemas principalmente en la comunicación con gazebo/fisico debido a que el controlador angular genera un error más grande de lo debido , por más que se intentó encontrar la causa de esto no se pudo encontrar nada que generará este sobre error , se intentó calcular de 4 diferentes maneras el error y ninguna funciono , tambien se reviso todos los mensajes que se mandan y se reciben , lo único que se encontró es que según odometría del robot está llegando a los puntos deseados como si realmente cumpliera el objetivo , se realizaron pruebas sin odometría y el resultado no cambiaba por lo que después de muchas modificaciones también descartamos eso y por último notamos que todos los códigos que se generaron el año pasado no funcion en el nuevo gazebo generando el mismo sobre error angular. Después de todas estas pruebas llegamos a la conclusión de que se necesita mandar y recibir valores de gazebo de una manera diferente por lo que no pudimos completar esa parte debido a que en gazebo y en físico los resultados eran los mismos.

Resultados obtenidos

Los resultados se pueden observar en el siguiente video:
https://youtu.be/EJsC4OBjaZo

Conclusión y análisis de resultados

En el video se pueden observar los 2 robots funcionando en el simulador de *RVIZ*, sin embargo, el modelo del robot real no puede mostrarse debido a inconsistencias que se encontraron al momento de configurar el simulador, sin embargo, en el video se pueden apreciar la línea respecto al punto de referencia odom la posición del robot.. La elipse de confidencialidade es muy grande debido al valor de los constantes que llegan a ser así debido a la diferencia de valores de las posiciones que se llegaron a medir, estas podrían tunearse más para obtener mejores resultados sin embargo está siempre permanecera asi de grande o de tamaño similar por la diferencia entre modelo ideal y modelo real.

En cierto momento se llega a detener el robot final siendo este uno de los problemas en los que aún se sigue buscando solución, no es el único problema referente a la velocidad angular y su posición, se están buscando actualmente alternativas de solución analizando exhaustivamente la arquitectura desarrollada.

Referencias

Interpretación de la matriz de Covarianza - FasterCapital. (s. f.). FasterCapital. https://fastercapital.com/es/tema/interpretaci%C3%B3n-de-la-matriz-de-covarianza.html#:~:text=Es%20una%20matriz%20cuadrada%20que%20muestra%20las%20varianzas%20de%20cada,las%20relaciones%20lineales%20entre%20variables.

nav_msgs/Odometry Documentation. (s. f.). https://docs.ros.org/en/diamondback/api/nav_msgs/html/msg/Odometry.html