Trabalho Final

Anna Caroline Bozzi[1], **Pamella A. de L. Mariano**[1]

[1]Departamento de Informática – Universidade Federal do Paraná (UFPR)
Curitiba – PR – Brazil

Abstract. *This article describes the implementation of Turtlebot3, following Luize and Mateus' tutorial, to navigate between points in the simulated environment of Gazebo. Due to the unavailability of the DINF environment, the World Default of Gazebo was used. Divided into two stages, the work focused on creating a Python code for basic movements and subsequently adapting it to handle obstacles, aiming to achieve autonomy in robot navigation. The methods, challenges, and results obtained while facing the adversities of the new simulation environment are detailed, contributing to the knowledge in autonomous systems for robotics.*

Resumo. *Este artigo descreve a implementação do Turtlebot3, conforme tutorial de Luize e Mateus, para navegar entre pontos no ambiente simulado do Gazebo. Devido àindisponibilidade do ambiente DINF, utilizou-se o World Default do Gazebo. Dividido em duas etapas, o trabalho focou na criação de um código Python para movimentos básicos e, posteriormente, na adaptação para lidar com obstáculos, buscando alcançar a autonomia na navegação do robô. Os métodos, desafios e resultados obtidos ao enfrentar as adversidades do novo ambiente de simulação são detalhados, contribuindo para o conhecimento em sistemas autônomos para robótica.*

1. Introdução

O presente trabalho foi desenvolvido como parte do trabalho final, baseado no tutorial elaborado por Luize e Mateus, estudantes de Iniciação Científica em Robótica, disponibilizado no repositório GitHub. O desafio selecionado consistia em habilitar o Turtlebot3 para navegar de um ponto a outro, escolhido dinamicamente a cada tentativa, no ambiente simulado do Gazebo.

Inicialmente, a execução do desafio estava planejada para ser realizada no ambiente DINF dentro do Gazebo para simulações em robótica. Entretanto, devido a imprevistos com a disponibilidade desse ambiente, buscamos alternativas para prosseguir com o trabalho. Nesse contexto, foi decidido utilizar o ambiente World Default do Gazebo como substituto, possibilitando assim a continuidade do desafio proposto.

O objetivo principal foi dividir o trabalho em duas partes distintas para alcançar o objetivo estabelecido. Na primeira etapa, concentramo-nos na implementação de um código em Python para controlar os movimentos do Turtlebot3 no ambiente padrão do Gazebo. Essa fase inicial foi essencial para garantir a movimentação básica do robô.

Posteriormente, direcionamos nossos esforços para a segunda parte do trabalho, que envolveu a tentativa de tratamento dos obstáculos no ambiente simulado. Essa etapa visava não apenas a movimentação do Turtlebot3 de um ponto a outro, mas também a capacidade de navegar de forma autônoma, evitando obstáculos que surgissem em seu caminho.

1.1. Trabalho Final

Nesta seção, serão apresentados dois códigos Python: o primeiro direcionado à navegação e o segundo focado no tratamento de obstáculos.

1.2. controla_turtlebot3.py

O código controla_turtlebot3.py é a primeira parte do trabalho que controla o Turtlebot3 no contexto do ROS2 (Robot Operating System 2). Em linhas gerais, o código Python inicializa o sistema ROS2, cria um nó chamado 'controlador_turtlebot3' e publica comandos de velocidade (linear e angular) para movimentar o robô. O loop principal é responsável por continuar enviando esses comandos enquanto o sistema ROS2 estiver operacional. Além disso, ele mostra informações sobre as velocidades linear e angular do Turtlebot3. Finalmente, o código encerra o nó e desliga o sistema ROS2 quando concluído. O código pode ser visto na Figura 1.

controla_turtlebot3.py

```python
import rclpy
from geometry_msgs.msg import Twist

def move_turtlebot3():
    rclpy.init()  # Inicializa o sistema ROS2
    node = rclpy.create_node('controlador_turtlebot3')  # Cria um nó ROS2 com o nome 'controlador_turtlebot3'

    cmd_vel_publisher = node.create_publisher(Twist, '/cmd_vel', 10)  # Cria um publicador para o tópico /cmd_vel

    # Cria uma mensagem Twist para enviar comandos de velocidade
    twist = Twist()

    # Define velocidades linear e angular desejadas
    twist.linear.x = 0.2  # Ajuste a velocidade linear conforme necessário
    twist.angular.z = 0.5  # Ajuste a velocidade angular conforme necessário

    # Loop principal para enviar comandos de velocidade continuamente
    while rclpy.ok():  # Enquanto o sistema ROS2 estiver operacional
        cmd_vel_publisher.publish(twist)  # Publica a mensagem Twist no tópico /cmd_vel
        node.get_logger().info('Movendo Turtlebot3')  # Mensagem de log indicando que o Turtlebot3 está em movimento
        node.get_logger().info('Velocidade Linear: {}, Velocidade Angular: {}'.format(
            twist.linear.x, twist.angular.z))  # Imprime as velocidades linear e angular

        # Desenvolver lógica dos obstáculos

        # Realiza um ciclo do sistema ROS2
        rclpy.spin_once(node, timeout_sec=0.1)  # Aguarda um curto intervalo de tempo para evitar bloqueios

    node.destroy_node()  # Finaliza o nó ROS2
    rclpy.shutdown()  # Encerra o sistema ROS2

if __name__ == '__main__':
    move_turtlebot3()
```

Figure 1. controla_turtlebot3.py

O código utiliza a biblioteca rclpy para inicializar o sistema ROS, criar um nó chamado 'turtlebot3_controller' e um publicador para o tópico /cmd_vel.

O cmd_vel é um tópico muito comum em sistemas de robótica que utilizam o ROS (Robot Operating System). Ele é usado para enviar comandos de velocidade para controlar o movimento de um robô, especialmente no caso de robôs móveis como o Turtlebot3.

A mensagem do tipo Twist é utilizada para definir as velocidades linear e angular desejadas do Turtlebot3. Neste exemplo, twist.linear.x é definido como 0.2 (velocidade linear) e twist.angular.z como 0.5 (velocidade angular). Os valores específicos escolhidos

para twist.linear.x e twist.angular.z podem variar dependendo do contexto e dos requisitos do sistema em que o Turtlebot3 está operando, nesse contexto:

- twist.linear.x = 0.2: Este valor define a velocidade linear desejada do Turtlebot3. O valor 0.2 indica uma velocidade linear moderada, suficiente para que o robô se mova de forma constante, mas não muito rápida, permitindo a navegação sem colisões, especialmente em ambientes com espaço limitado ou onde a precisão no movimento é necessária como é o caso do mundo escolhido.
- twist.angular.z = 0.5: Esse valor determina a velocidade angular desejada do Turtlebot3. A velocidade angular refere-se à taxa de rotação do robô no local. O valor 0.5 indica uma rotação moderada, permitindo que o robô gire a um ritmo considerável, mas não muito rápido para evitar voltas bruscas que possam afetar a estabilidade ou precisão do movimento.

Dentro do loop principal (while rclpy.ok()), a função cmd_vel_publisher.publish(twist) envia continuamente os comandos de velocidade definidos pela mensagem Twist para o tópico /cmd_vel. Além disso, são exibidas informações sobre as velocidades linear e angular no log.

É importante ressaltar que a seção onde está comentado ”Adicione aqui a lógica para lidar com obstáculos, como sensores ou informações de mapa” é o local onde será adicionado um código para que o robô possa reagir a informações do ambiente, como dados de sensores, evitando obstáculos ou seguindo um determinado trajeto.

Por fim, ao finalizar o uso do nó ROS, o código desliga o nó e encerra o sistema ROS com node.destroy_node() e rclpy.shutdown() respectivamente.

1.3. Resultados

A implementação do código permitiu a navegação bem-sucedida do Turtlebot3 por todo o ambiente escolhido. No entanto, os resultados obtidos não foram satisfatórios, pois o robô frequentemente colidia com os obstáculos presentes no ambiente simulado. Essa limitação impactou negativamente na capacidade do robô de completar a trajetória desejada sem intercorrências. Nesse contexto, as Figuras 2, 3 e 4 ilustram os desafios encontrados durante a navegação autônoma no ambiente simulado. Esses resultados ressaltam a necessidade de realizar o tratamento de obstáculos para garantir uma navegação mais eficiente e livre de colisões do Turtlebot3.

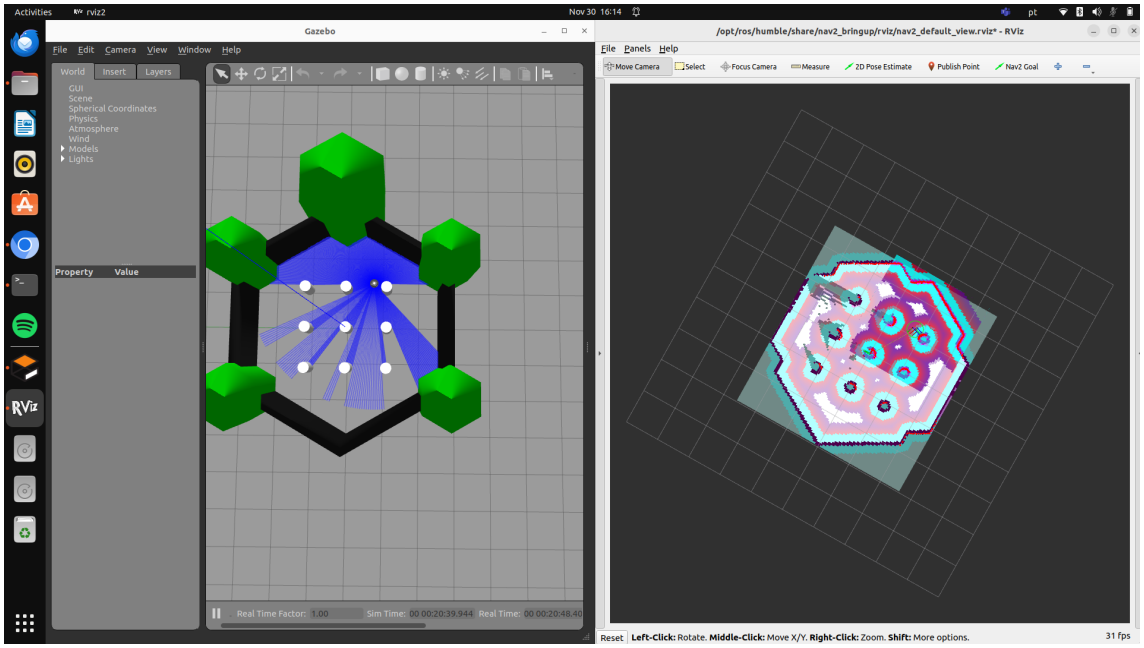


Figure 2.

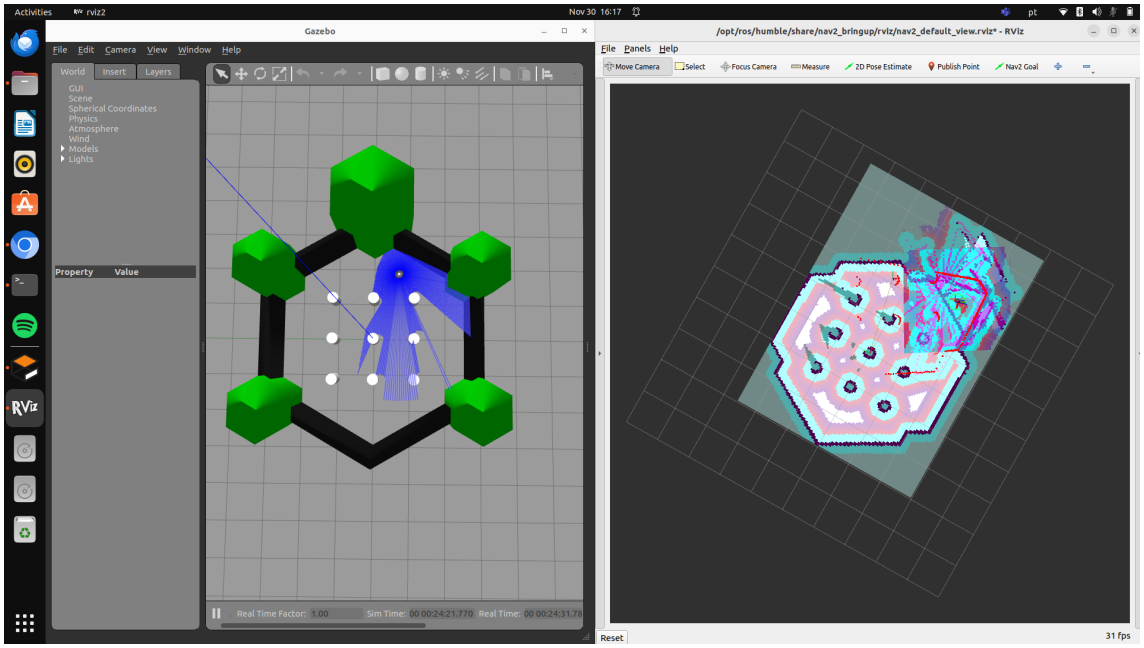


Figure 3.

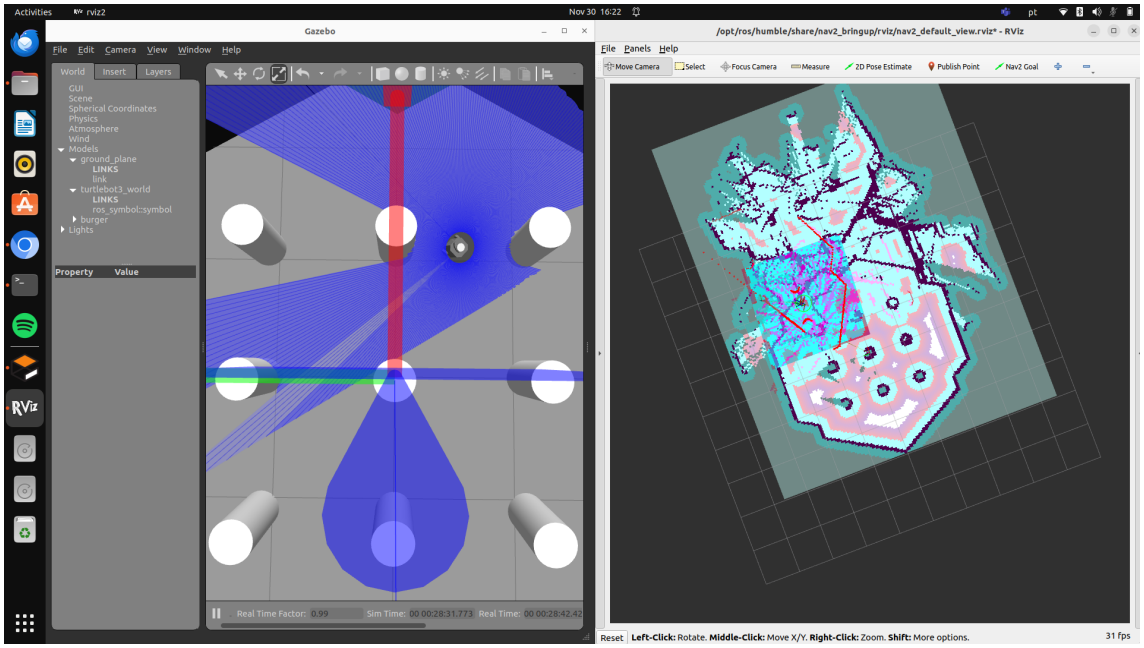


Figure 4.

2. Otimizando o Controlador para o Turtlebot3 com ROS2: Comportamento com obstáculos

O código em questão é um controlador para o Turtlebot3, com implementação para enfrentar obstáculos como versão 2 do código já apresentado, a 5 a seguir compara as duas versões propostas . O principal objetivo da versão 2 é permitir que o robô navegue em um ambiente simulado, evitando obstáculos com base nos dados fornecidos pelos sensores laser. O loop principal controla o comportamento do Turtlebot3 com base nas leituras do sensor laser.

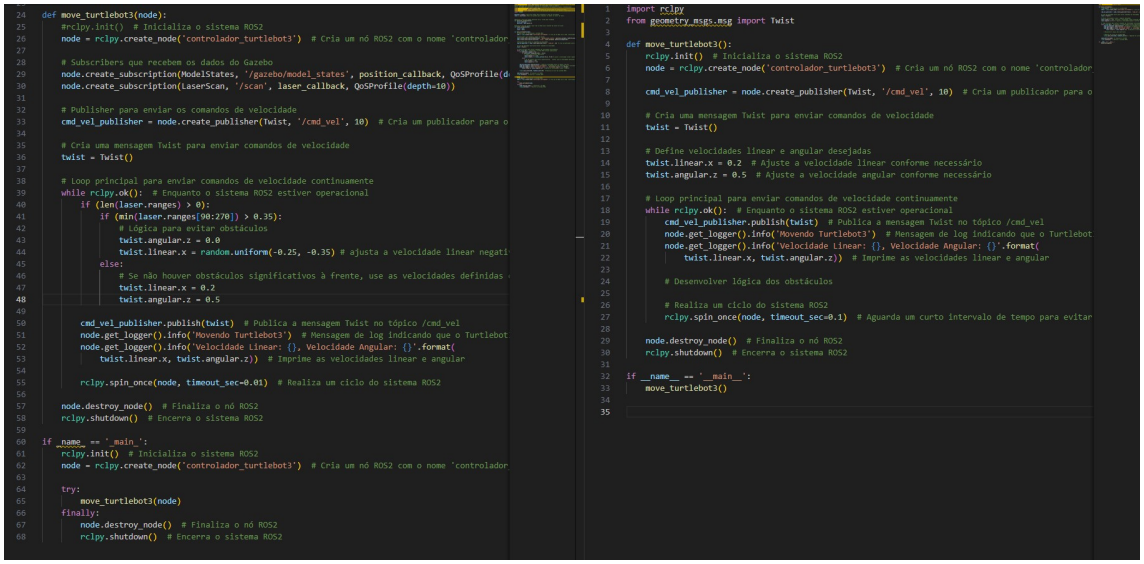
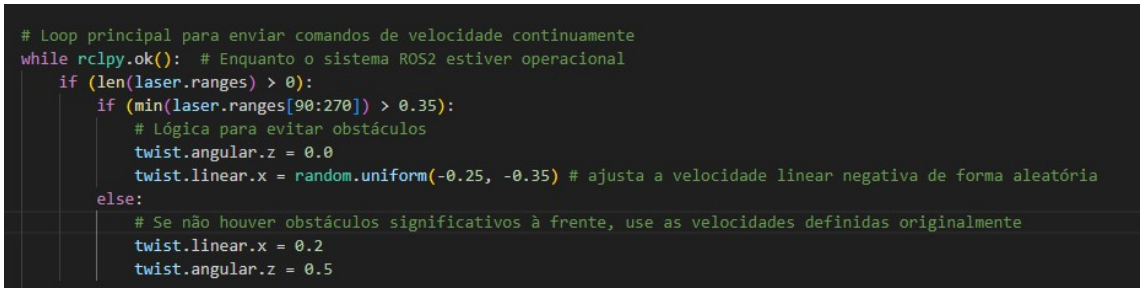


Figure 5. V2 e V1 do código proposto

A seguir a descrição das melhorias propostas:

- Lógica de Controle Aprimorada
  - A lógica de controle no Código 2 foi aprimorada para levar em consideração a presença de obstáculos. Se o sensor laser detectar obstáculos significativos à frente do Turtlebot3, o código ajusta dinamicamente a velocidade linear de forma aleatória, permitindo uma navegação mais flexível ao redor dos obstáculos. A velocidade angular foi mantida constante para garantir uma rotação eficaz quando necessário. Se não houver obstáculos detectados, as velocidades definidas originalmente são mantidas. A 6 a seguir ilustra o descrito.



```
# Loop principal para enviar comandos de velocidade continuamente
while rclpy.ok():  # Enquanto o sistema ROS2 estiver operacional
    if (len(laser.ranges) > 0):
        if (min(laser.ranges[90:270]) > 0.35):
            # Lógica para evitar obstáculos
            twist.angular.z = 0.0
            twist.linear.x = random.uniform(-0.25, -0.35) # ajusta a velocidade linear negativa de forma aleatória
        else:
            # Se não houver obstáculos significativos à frente, use as velocidades definidas originalmente
            twist.linear.x = 0.2
            twist.angular.z = 0.5
```

Figure 6. Implementação do loop que trata obstáculos

- Reorganização da Inicialização do ROS2:
  - Na versão aprimorada, a inicialização do ROS2 (rclpy.init()) foi movida para fora da função principal, conforme 7, permitindo uma maior flexibilidade na gestão do nó ROS2.
- Tratamento de Exceções:

  - Introduzimos blocos try e finally, conforme também 7, para garantir a destruição adequada do nó ROS2 mesmo em caso de exceções, contribuindo para uma execução mais robusta.

```python
if __name__ == '__main__':
    rclpy.init()  # Inicializa o sistema ROS2
    node = rclpy.create_node('controlador_turtlebot3')  # Cria um nó ROS2 com o nome 'controlador_turtlebot3'

    try:
        move_turtlebot3(node)
    finally:
        node.destroy_node()  # Finaliza o nó ROS2
        rclpy.shutdown()  # Encerra o sistema ROS2
```

Figure 7. Implementação da MAIN

- Melhorias na Comunicação e Log:
  - O código aprimorado incorporou melhorias na comunicação, utilizando métodos mais eficientes para publicação de mensagens e registro de logs.
- Uso de Constantes Nomeadas:
  - Em vez de valores numéricos diretamente no código, a versão aprimorada adotou o uso de constantes nomeadas, proporcionando uma melhor compreensão e manutenção do código.

2.1. Resultados

Ao aprimorar o código de controle do Turtlebot3, buscamos não apenas otimizar o desempenho, mas também promover boas práticas de programação. A organização, modularidade e clareza do código são cruciais para facilitar a manutenção e a evolução dos sistemas robóticos.

Com as modificações propostas, esperava-se que o código fosse mais eficiente e responsivo.Além de resolver o labirinto. A utilização de argumentos em vez de variáveis globais melhora a organização do código, enquanto a verificação adequada do encerramento do nó garante uma saída mais limpa e controlada. Porém o tempo de espera no loop principal fez com que o sistema ROS2 respondesse bem lentamente aos eventos de detecção de obstáculos.

Foi ajustado o valor de timeout_sec em rclpy.spin_once(node, timeout_sec=0.01) para um valor ainda menor, como 0.001, para garantir que o sistema ROS2 responda mais rapidamente, e ainda assim houve lentidão.

3. Conclusão

A otimização do código de controle do Turtlebot3, incorporando dados do sensor laser, representa um avanço significativo na capacidade de navegação do robô em ambientes complexos. Ao introduzir uma lógica adaptativa que ajusta dinamicamente a velocidade linear em resposta à detecção de obstáculos, o Turtlebot3 demonstra uma maior inteligência na navegação. A capacidade de alterar a velocidade linear de maneira aleatória em resposta à presença de obstáculos permite ao Turtlebot3 contornar barreiras de forma mais eficiente, adaptando-se ao ambiente dinâmico. Entretanto, é importante destacar

que, em alguns cenários, a abordagem de ajuste dinâmico da velocidade linear está resultando em lentidão excessiva. A lentidão pode ser uma preocupação, especialmente em ambientes onde uma resposta mais ágil é necessária. A incorporação de algoritmos de planejamento de trajetória mais avançados pode melhorar a eficiência geral da navegação, permitindo que o Turtlebot3 evite obstáculos de maneira mais preditiva.

Em conclusão, a melhoria no controle do Turtlebot3 representa um passo significativo na direção certa, proporcionando uma navegação mais inteligente e adaptável. Ao enfrentar o desafio da lentidão, é fundamental explorar estratégias adicionais para otimizar o desempenho e atender às demandas específicas do ambiente em que o robô opera. O processo contínuo de refinamento e ajuste garantirá que o Turtlebot3 alcance um equilíbrio ideal entre eficiência e agilidade em sua jornada de navegação.

4. Link para o código

https://github.com/ACBozzi/ROS2/tree/main

References