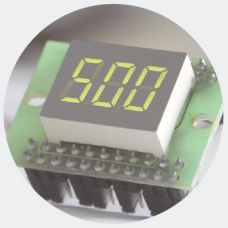
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES






Desenvolvimento de um Sistema de Navegação para Robôs

DAVID JOSÉ LOPES DAS NEVES
novembro de 2020

POLITÉCNICO
DO PORTO

isep Instituto Superior de Engenharia do Porto

Instituto Superior de Engenharia do Porto ⊙ Rua Dr. António Bernardino de Almeida, 431 ⊙ 4249-015 Porto
Tel. +351 228 340 500 ⊙ Fax +351 228 321 159 ⊙ mail@isep.ipp.pt ⊙ www.isep.ipp.pt

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

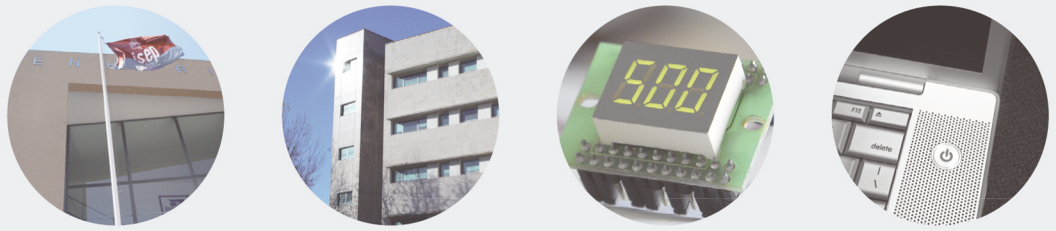
isep

Desenvolvimento de um Sistema de Navegação para Robôs

DAVID JOSÉ LOPES DAS NEVES
Novembro de 2020

INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA ELECTROTÉCNICA E DE COMPUTADORES

isep



Desenvolvimento de um Sistema de Navegação para Robôs

DAVID JOSÉ LOPES DAS NEVES
Novembro de 2020

POLITÉCNICO DO PORTO

DESENVOLVIMENTO DE UM SISTEMA DE NAVEGAÇÃO PARA ROBÔS

David José Lopes das Neves

Mestrado em Engenharia Electrotécnica e de Computadores
Área de Especialização de Automação e Sistemas
Departamento de Engenharia Electrotécnica
Instituto Superior de Engenharia do Porto

2020

Este relatório satisfaz, parcialmente, os requisitos que constam da Ficha de Disciplina de Tese/Dissertação, do 2º ano, do Mestrado em Engenharia Electrotécnica e de Computadores

Candidato: David José Lopes das Neves, Nº 1130673, 1130673@isep.ipp.pt

Orientador da Instituição: Lino Manuel Baptista Figueiredo, lbf@isep.ipp.pt

Mestrado em Engenharia Electrotécnica e de Computadores

Área de Especialização de Automação e Sistemas

Departamento de Engenharia Electrotécnica

Instituto Superior de Engenharia do Porto

28 de novembro de 2020

Agradecimentos

Em primeiro lugar, agradeço ao meu orientador, Professor Lino Figueiredo, por todo o apoio, ajuda e conselhos dados durante a realização deste projeto. Para além da sua disponibilidade e orientação neste trabalho, permitiu-me também evoluir no meu processo de aprendizagem.

À minha família pelo apoio e motivação.

Resumo

O presente documento tem como objetivo relatar o desenvolvimento de um sistema de navegação, baseado na ferramenta de programação para robôs *Robot Operating System* (ROS), com o objetivo de permitir ao robô deslocar-se num determinado espaço.

Inicialmente, realizou-se um estudo teórico sobre plataformas robotizadas, com o intuito de entender melhor as plataformas que há no mercado e como são constituídas. Foi também realizada uma análise sobre a ferramenta ROS, de forma a conhecer o sistema deste e os seus integrantes.

Após o estudo teórico, passou-se à formulação das fases que se consideraram necessárias para a realização do projeto. Numa primeira fase, iam ser desenvolvidos determinados programas no simulador *Turtlesim*, que posteriormente iriam ser adaptados para serem executados pelo simulador *TurtleBot*, sendo que este adiciona uma nova característica aos programas, o uso do sensor laser. Concluídas as fases de simulação, o objetivo seria utilizar o robô comercial Magni para fazer demonstrações reais dos programas anteriormente desenvolvidos.

Em seguida, deu-se início à programação do simulador 2D *Turtlesim*, com o objetivo de desenvolver programas que permitissem ao robô realizar simples operações de movimento, sendo estas as seguintes: mover-se em linha reta, efetuar uma rotação no local e deslocar-se até a um determinado ponto no seu espaço.

Concluída a simulação no *Turtlesim*, procedeu-se para o desenvolvimento de programas no simulador *TurtleBot*, sendo um dos objetivos a utilização de um sensor. Foram reutilizados os programas desenvolvidos no *Turtlesim*, com o acréscimo do sensor laser equipado no *TurtleBot*, permitindo desenvolver os seguintes programas: o robô movimenta-se, alternando entre movimentos lineares e rotacionais; o robô está em constante movimento linear, evitando colisões com as paredes; o robô desloca-se até um determinado ponto, evitando colisões com as paredes, passando por um conjunto de pontos intermédios, para facilitar o deslocamento do mesmo.

A programação do robô Magni fica como um possível trabalho futuro a realizar, assim como explorar outras componentes da ferramenta ROS, para se conseguir desenvolver um sistema de navegação mais robusto.

Palavras-Chave

ROS, robô, *Turtlesim*, *TurtleBot*, sensor laser, Magni.

Abstract

The aim of this document is to describe the process used to develop a navigation system, based on the tool for programming robots Robot Operating System (ROS), with the purpose of making a robot move in a certain space.

In the beginning, a theoretical study about robot platforms was carried out, with the intention of getting a better understanding of the market and its components. Also, it was made an analysis of the tool ROS, to know better its system and its elements.

After the study, the next step was to state the phases that were considered necessary for the development of this project. Primarily, specific programmes were going to be developed on the Turtlesim simulator, then they were going to be adapted to run on the TutleBot simulator, where its possible to add a new feature to the programmes, the laser sensor. Once all the simulations were completed, the next goal was to use the commercial robot Magni for demonstrating real applications of the programmes developed on the simulators.

After that the programming of the 2D Turtlesim simulator started, with the goal of developing programs that would allow the robot to perform simple movement operations, such as: moving in a straight line, rotating on the spot and moving up to a certain point in its area.

After finishing the simulation in Turtlesim, programs were developed in the TurtleBot simulator, one of the purposes being the use of a sensor. The programs developed in Turtlesim were reused, with the addition of the laser sensor equipped in the TurtleBot, allowing the development of the following programs: the robot moves, shifting between linear and rotating movements; the robot is in constant linear motion, avoiding collisions with the walls; the robot moves to a certain point, avoiding collisions with the walls, passing through a set of middle points, to ease its displacement.

The programming of the Magni robot is seen as a possible future task to be carried out, as well as exploring other components of the ROS tool, in order to develop a more robust navigation system.

Keywords

ROS, robot, Turtelsim, TurtleBot, laser sensor, Magni

Índice

Índice de Figuras

Acrónimos

ABS – *Acrylonitrile Butadiene Styrene*

QR – *Quick Response*

ROS – *Robot Operating System*

SAIL – *Stanford Artificial Intelligence Laboratory*

1. INTRODUÇÃO

1.1. CONTEXTUALIZAÇÃO

A robótica é um tema cada vez mais presente nos dias de hoje, possuindo um grande leque no que toca à variedade de robôs existentes no mundo, desde braços robóticos a robôs humanoides. Esta área está em constante evolução, possuindo grande presença em enumeres empresas, com o objetivo de melhorar a produtividade, pois estes são capazes de realizar diversas tarefas, de forma eficaz e eficiente.

Ao longo dos anos, foram desenvolvidos diferente tipos de robôs com diferentes fins, como por exemplo braços robóticos desenhados para atuar em linhas de montagem e plataformas robotizadas para transporte de cargas. Motores e sensores são algumas tecnologias esquipadas nos robôs, que permitem a estes atuar de acordo com a programação que lhes foi aplicada. Existem diferentes tipos de tecnologias para a programação de robôs, sendo alguns proporcionados pelas empresas que os fabricam, assim como existem diferentes ferramentas para a programação dessas tecnologias.

Este projeto surgiu no âmbito de conhecer melhor uma dessas ferramentas para a programação de robôs, denominado de *Robot Operating System* (ROS). Este vem equipado com uma grande coleção de bibliotecas e utensílios, para ajudar o utilizador a simplificar a

criação de programas. O objetivo deste projeto vai consistir em estudar a ferramenta ROS para programação de um robô, sendo que este foi fabricado com esse propósito, fazendo com que este se desloque até determinados locais num espaço predefinido.

1.2. OBJETIVOS

O objetivo principal deste projeto é desenvolver um sistema de navegação para robôs, recorrendo à ferramenta ROS. Para a realização deste projeto foi necessário executar os seguintes passos:

- Estudo de plataformas robotizadas, com o intuito de explorar as tecnologias utilizadas pelos mesmos;
- Estudo da ferramenta ROS;
- Programação da ferramenta ROS a nível de simulação;
- Programação do robô;
- Execução de testes e análise de resultados.

1.3. CALENDARIZAÇÃO

A Figura 1 demonstra o cronograma em que o desenvolvimento deste projeto se vai basear. Este abrange um conjunto de tarefas essenciais para o desenvolvimento de um sistema de navegação para robôs, como por exemplo: Estudo do ROS, isto é, estudar ao detalhe a tecnologia ROS, como é programada, a linguagem que usa, entre outros aspetos; Programação do ROS a nível de simulação, onde vão ser aplicados os conhecimento adquiridos num ambiente virtual; Testes e resultados, ou seja, observação e estudo dos resultados obtidos.

	Março		Abril				Maio				Junho				Julho				Agosto				Setembro				Outubro			
Actividades	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4	S1	S2	S3	S4
Análise de paltaformas robotizadas																														
Estudo do ROS																														
Programação do ROS (simulação)																														
Programação do robô																														
Testes e resultados																														
Relatório																														

Figura 1 – Cronograma

1.4. ORGANIZAÇÃO DO RELATÓRIO

Este relatório encontra-se dividido em 7 capítulos, em que o Capítulo 1 apresenta uma introdução ao projeto.

No capítulo seguinte, 2, é apresentado um estudo sobre algumas plataformas robotizadas comerciais e sobre a ferramenta ROS.

No Capítulo 3 são apresentadas as fases que foram consideras necessárias para a realização do projeto, explicando o contributo de cada uma delas.

O Capítulo 4 apresenta a implementação realizada no simulador *Turtlesim*, assim como excertos de código dos programas desenvolvidos e os resultados obtidos.

No Capítulo 5 é apresentada a implantação efetuada simulador *Turtlebot*, contendo também excertos de código dos programas criados e os resultados obtidos dos mesmos.

O último capítulo, 6, apresenta as conclusões obtidas após a realização do projeto e as ações futuras que se poderão realizar para conseguir alcançar melhores resultados.

2. ESTADO DE ARTE

Neste capítulo vão ser realizados os estudos necessários para o desenvolvimento de um sistema de navegação para robôs. Nele está contido uma análise a diferentes plataformas robotizadas comerciais, assim como o estudo da ferramenta ROS.

2.1. PLATAFORMAS ROBOTIZADAS

Antes de avançar com o estudo da ferramenta ROS, decidiu-se realizar um estudo a plataformas robotizadas, com o principal objetivo de descobrir as tecnologias que os robôs usam para se movimentar, mas também com o intuito de ficar a conhecer melhor os tipos de plataformas robotizadas que há no mercado e como são constituídos.

2.1.1. MAGNI

A plataforma robotizada Magni, desenvolvida pela *Ubiquity Robotics*, é construída com alumínio resistente, tendo a capacidade de suportar cerca de 100 kg de carga. Tem uma avançada capacidade de navegação, permitindo que este se desloque dentro de habitações, evitando obstáculos e melhorando o seu mapa interno, isto tudo graças ao sistema de navegação *ceiling fiducial based navigation* e aos sonares que tem incorporado.

O robô é alimentado com duas baterias, uma de 12V 7A DC e outra de 5V 7A DC, e desloca-se com 2 *hub motors* de 200W, que estão ligados às rodas motoras, fazendo com que este se desloque a uma velocidade máxima de 2 m/s. Com baterias de 10 Ah, consegue durar 8 horas e com baterias de 32 Ah, consegue durar 24 horas.

No que diz respeito às dimensões, este robô tem 43,91 cm de comprimentos, 41,74 cm largura e 26,5 cm de altura. O Magni vem equipado com o controlador Quad-core ARM A9 – Raspberry Pi3 e ROS Kinetic, sendo utilizado o sistema operativo Ubuntu 16.04 para a sua programação. A Figura 2 apresenta o robô anteriormente descrito. [1]

Figura 2 – Magni [1]

2.1.2. *MOBILE ROBOT BASE*

A *Zagros Robotics* apresenta 2 modelos de bases robotizadas e cada modelo possui 2 formatos. As bases são compostas por 2 prateleiras, feitas de material *Acrylonitrile Butadiene Styrene* (ABS), que podem ter formato circular ou quadrangular, 2 motores de 12V 3A, rodas motoras e rodas não motoras, com 15 cm e 7,5 cm de diâmetro, respetivamente.

Como foi referido, as bases dividem-se em 2 modelos: *REX Mobile Robot Bases* e *Max Mobile Robot Bases*, em que as principais diferenças são a capacidade de carga e a velocidade com que se deslocam. Os modelos REX, ao contrário dos Max, possuem um *5545 Dual Channel Enconder with Index Pulse*, têm uma capacidade de carga de 16 kg e consegue-se deslocar a uma velocidade de 0,3 m/s. Os modelos Max têm uma capacidade

de carga de 13,6 kg e consegue-se deslocar a uma velocidade de 0,2 m/s com carga máxima.

No que diz respeito às medidas, ambos os modelos têm a possibilidade de ter as prateleiras com 35,5 cm ou 40 cm de dimensão, mas os modelos Max ainda têm possibilidade de ter 30,5 cm de dimensão. No caso do formato circular, as bases, dependendo do modelo, podem ter 30,5 cm, 35,5 cm ou 40 cm de diâmetro e no caso do formato quadrangular, as bases podem ter 30,5x30,5 cm, 35,5x35,5 cm ou 40x40 cm. [2][3]

Estas bases não possuem um sistema de navegação nem outra qualquer tecnologia, a não ser os motores, ficando à responsabilidade de quem adquirir uma base escolher que tecnologias pretende usar para programar o robô.

A Figura 3 e a Figura 4 apresentam as bases na forma circular e na forma quadrangular, respetivamente, e a Figura 5 demonstra um exemplo de aplicação usando a base quadrangular de 30,5x30,5, criando um robô aspirador.

Figura 3 – *Mobile robot base* (Circular) [2]

Figura 4 – *Mobile robot base* (Quadrangular) [2]

Figura 5 – Exemplo de aplicação (base quadrangular) [4]

2.1.3. *HANGFA DISCOVERY AND HANGFA COMPASS ROBOT PLATFORM*

A empresa *Hangfa Hydraulic Engineering* desenvolveu algumas séries de plataformas robotizadas, como as séries *Hangfa Discovery* e as *Hangfa Compass*. Dentro de cada série, existem plataformas com construções físicas distintas, sendo estas construídas de alumínio. Contudo, estas estão equipadas com as mesmas tecnologias, havendo pequenas diferenças entre estas.

Todas as plataformas usam motores DC *coreless* e possuem sensores ultrassónicos, havendo distinção na quantidade destes equipamentos que são incorporados nos robôs e na

potência dos motores. Como fonte de energia, cada robô possui uma bateria de ião lítio, o que permite que estes consigam ter um funcionamento com duração máxima de 30 horas.

Os robôs vêm equipados com uma placa de desenvolvimento RHF407, que tem incorporada o microcontrolador Cortex M4 STM32F407 e possibilita interfaces com CAN bus e RS232. Na aquisição de qualquer um dos robôs, são fornecidos alguns exemplos e aplicações de código baseados neste microcontrolador, com o objetivo de acelerar o processo de programação das plataformas.

A Figura 6 apresenta um exemplo de um robô da série *Hangfa Discovery*, sendo este denominado de *Hangfa Discovery Q2*. Este robô é composto por 4 rodas omnidirecionais, QMA – 10 *omni wheels*, que permite à plataforma realizar manobras de translação, rotação e as duas em simultâneo. Está equipado com 4 motores DC *coreless* de 17W, 1 bateria de ião lítio 24V 10,4 Ah, 5 sensores ultrassónicos e a sua duração varia entre 8 e 30 horas. Possui uma capacidade de carga de 10 kg e consegue atingir uma velocidade máxima de 0,65 m/s. No que toca às dimensões, tem cerca de 35,9 cm de comprimento, 31,35 cm de largura e 11,4 cm de altura.[5]

Figura 6 – Plataforma *Hangfa Discovery Q2* [5]

Na Figura 7 pode-se ver outro exemplo de uma plataforma robotizada fabricada por esta empresa, sendo que esta pertence à série *Hangfa Compass*. Esta base robotizada usa rodas omnidirecionais, QL-10 *omni wheel*, estando equipado com 3 motores DC *coreless* de 30W. Utiliza 5 sensores ultrassónicos e 1 bateria de ião lítio 24V 18,2 Ah, permitindo a este um funcionamento que varia entre 10 e 30 horas. Tem uma capacidade de carga de 20

kg e consegue alcançar uma velocidade máxima de 1,2 m/s. Em relação às suas dimensões, tem cerca de 36,9 cm de comprimento, 40,05 cm de largura e 11 cm de altura.[6]

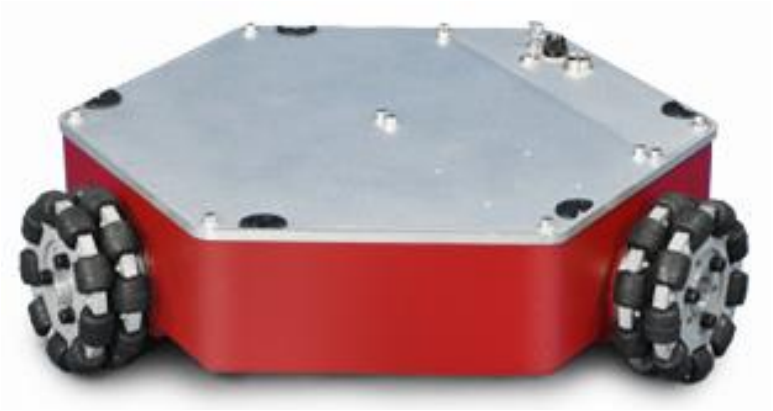


Figura 7 – Plataforma *Hangfa Compass Q1* [6]

2.1.4. *KOBUKI TURTLEBOT 2*

Este robô, presente na Figura 8, possui uma forma física semelhante a uma estante, sendo fabricado pela empresa *Yujin Robot*, com propósitos educacionais e de investigação, conseguindo realizar inúmeras tarefas direcionadas para a robótica, como localização e mapeamento.

A base do robô denomina-se de *Kobuki*, uma base robótica comercial que consegue atingir uma velocidade linear máxima de 0,7 m/s e uma velocidade rotacional máxima de 180º graus/s, com um tempo de operação de 3 ou 7 horas, dependendo da bateria. São usadas baterias de ião de lítio com 14,8V, diferenciando a corrente com o tamanho da bateria, com possibilidade de carregamento através de *docking station*.

Equipado com motores e com sensores *cliff* e *wheel drop*, o robô é capaz de se mover livremente num determinado espaço, adaptando-se às mudanças de piso, evitando perigo de queda. A sua programação é realizada através da ferramenta ROS, na linguagem C++, recorrendo também ao simulador *Gazebo*, um simulador de robótica 3D, sendo que fica ao critério de quem a adquirir, adicionar o *hardware* necessário para a seu funcionamento.

O *Turtlebot2* é a combinação da base *Kobuki* com plataformas, em que no meio destas é colocada uma câmara para realizar a navegação do robô de forma mais precisa, podendo também acrescentar um componente na plataforma superior, dependendo do propósito robô, como por exemplo um braço robótico. Tem a capacidade de realizar localização e

mapeamento em simultâneo, possibilitando a criação de um mapa do ambiente em que o robô é inserido. O robô tem cerca de 35,4 cm de comprimento e largura, tendo uma altura de 42 cm devido às plataformas, conseguindo ter uma capacidade de carga máxima de 5 kg em chão e 4 kg em tapetes.[7][8]

Figura 8 – *Kobuki Turtlebot2* [8]

2.1.5. *4WD MECANUM WHEEL MOBILE ROBOTIC PLATFORM*

O seguinte robô, desenvolvido pela *King Kong Robot*, consiste numa plataforma robótica com 4 rodas *mecanum*, cada uma com 10 cm de diâmetro, e com uma estrutura robusta de alumínio. Estas rodas permitem que a plataforma consiga ter um movimento omnidirecional, em que cada roda tem um motor associado, sendo estes motores DC de 12V com *encoder*.

Vem equipado com um Arduíno 328 e uma extensão IO, para ser programado nas linguagens C e C++. São fornecidos exemplos de código para ajudar os utilizadores terem uma melhor interação com o Arduino. Também possui 4 sensores ultrassónicos, com detenção de 4 cm a 500 cm, uma bateria de ião de lítio 12V 3500mA, com duração de 1 hora, e um carregador de 12V.

Na base do robô, já existem alguns orifícios previamente desenhados, com o objetivo de facilitar a introdução de equipamentos, como camaras, braços robóticos, entre outros. Para além disso, a plataforma consegue-se movimentar-se em zonas de terreno rugoso. No que diz respeito à sua dimensão, possui 40 cm de comprimento, 30,7 cm de largura e 12,3 cm

de altura, consegue atingir uma velocidade máxima de 0,6 m/s e tem a capacidade de carga de 10 kg. Na Figura 9 é possível visualizar a plataforma robotizada anteriormente descrita.[9]

Figura 9 – Plataforma robotizada *4WD Mecanum Whell* [9]

2.2. *ROBOT OPERATING SYSTEM* (ROS)

O *Robot Operating System* (ROS) é uma ferramenta *open-source* utilizada para o desenvolvimento de aplicações robotizadas. Semelhante a um sistema operativo num computador pessoal, os programas criados no ROS permitem ao utilizador controlar as operações de um robô, recorrendo à reutilização de código presente nestes para outras aplicações, sendo este um dos aspetos principais desta ferramenta. [10][11]

O ROS começou por ser desenvolvido pela *Stanford Artificial Intelligence Laboratory* (SAIL) em 2007, tendo sido continuado pelo laboratório de pesquisa em robótica *Willow Garage*, com uma colaboração de mais de 20 instituições. Esta ferramenta, possuindo um aspeto mais robusto, começou a ser usado por muitos centros de investigação, para o desenvolvimento de vários projetos, e por diversas empresas, sendo que estas adaptaram os seus produtos para poderem ser usados no ROS. Atuadores e sensores são exemplos de componentes que foram adaptados de forma a poderem ser programados pelo ROS.

A arquitetura do ROS encontra-se dividido em três níveis, sendo eles o nível *Filesystem*, o nível *Computation Graph* e o nível *Community*. [12]

2.2.1. NÍVEL *FILESYSTEM*

Este primeiro nível usa um grupo de conceitos para explicar como está formado o ROS internamente, como estão estruturadas as pastas e o número mínimo de ficheiros que necessita para executar uma aplicação. Um programa ROS é dividido em pastas e estas contêm ficheiros, cada um descrevendo a sua funcionalidade. Os componentes que constituem o primeiro nível são os seguintes:

- **Pacotes (*Packages*)**: Um pacote possui o conteúdo e a estrutura mínima para criar um programa em ROS, podendo conter diversos elementos, como por exemplo bibliotecas e ficheiros executáveis;

- **Manifestos (*Manifests*)**: Os manifestos fornecem informações sobre os pacotes, sobre as dependências existentes, entre outros;

- ***Stacks***: As *stacks* são definidas como um conjunto de pacotes, com o objetivo de simplificar o processo da partilha de código;

- **Tipos de mensagem (msg)**: Uma mensagem é a informação que é transmitida de um processo para outro. O ROS já possui muitos tipos de mensagens e cada uma é constituída por dois elementos: campo e constante. O campo define o tipo de dados que a mensagem vai transmitir;

- **Tipos de serviço (srv)**: As descrições dos serviços definem a estrutura dos dados de um pedido e de uma resposta dos serviços em ROS. É um sistema cliente/servidor em que o cliente envia um pedido, permanecendo inativo até obter uma resposta por parte do servidor. [12]

Na Figura 10 pode-se visualizar a estrutura do primeiro nível da arquitetura do ROS.

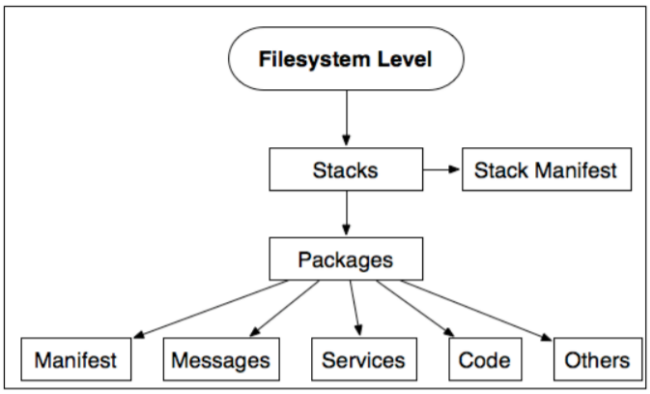


Figura 10 – Estrutura do nível *Filesystem* [12]

2.2.2. NÍVEL *COMPUTATION GRAPH*

O segundo nível da arquitetura do ROS é onde estão estabelecidas as comunicações entre os processos e os sistemas. Faz parte deste nível os conceitos e sistemas que o ROS apresenta para lidar com processos, comunicar com um ou mais computadores, entre outros. O ROS cria uma rede onde todos os processos estão conectados e qualquer nó presente no sistema pode aceder a esta, podendo interagir com outros nós, ver a informação que estes estão a transmitir e enviar dados para a rede. Os conceitos básicos presentes neste nível são os seguintes:

- **Nós (*Node*)**: Os nós são executáveis que comunicam com outros processos através de tópicos ou serviços. Para realizar interação entre um processo e outros nós, é necessário criar um nó com esse processo para conectá-lo à rede do ROS. A prática mais comum consiste em possuir muitos nós com diversas funções, sendo pouco habitual desenvolver um nó que realize todas as operações do sistema. Um nó deve possuir um nome único no sistema e pode ser escrito nas linguagens de programação C++ e *python*;

- ***Master***: O *Master* é responsável por atribuir nomes e registar os nós no sistema ROS, tendo como função permitir que os nós se localizem uns aos outros. Uma vez localizados, estes comunicam entre si num formato de *peer-to-peer* (ponto-a-ponto), um formato em que os nós conseguem partilhar informação sem a necessidade de um servidor central. Sem o *Master* não é possível estabelecer comunicação com os nós, serviços, entre outros;

- **Servidor de Parâmetros (*Parameter Server*)**: O servidor de parâmetros oferece a possibilidade de guardar dados através do uso de palavras chaves numa localização central. Estes parâmetros dão-nos a possibilidade de configurar os nós enquanto estes estão a ser executados ou alterar o seu funcionamento;

- **Mensagens (*Messages*)**: A comunicação entre nós é efetuada através da troca de mensagens. Um nó envia informações para outro nó usando mensagens que são publicadas por um tópico. As mensagens contêm dados que transmitem a informação para outros nós, existindo muitos tipos de mensagens no sistema ROS, com a possibilidade de o utilizador criar o seu próprio tipo de mensagem;

- **Tópicos (*Topics*)**: Os tópicos são usados pelos nós para a transmissão de dados. Diz-se que um nó está a enviar dados quando este faz uma publicação para um tópico e os nós recebem os dados quando fazem uma subscrição a esse mesmo tópico. Um tópico pode ter múltiplas publicações e subscrições, sendo que o tipo de mensagem presente no tópico tenha de ser o mesmo, de forma a ser possível a troca de dados entre os nós. No sistema ROS, os tópicos são transmitidos através do protocolo TCP/IP, ficando assim conhecido como protocolo TCPROS;

- **Serviços (*Services*)**: Os serviços oferecem a possibilidade de interagir com os nós, isto é, os serviços são usados quando se pretende fazer um requisito ou obter uma resposta por parte de um nó. Esta operação não pode ser realizada através de tópicos e não existe serviços base, cabendo ao utilizador criar os seus serviços;

- ***Bags***: As *bags* são um mecanismo de armazenamento de dados do sistema ROS, capaz de guardar toda a informação de mensagens, tópicos, serviços, entre outros. [12]

A figura seguinte apresenta a estrutura deste segundo nível da arquitetura da ferramenta ROS anteriormente descrito.

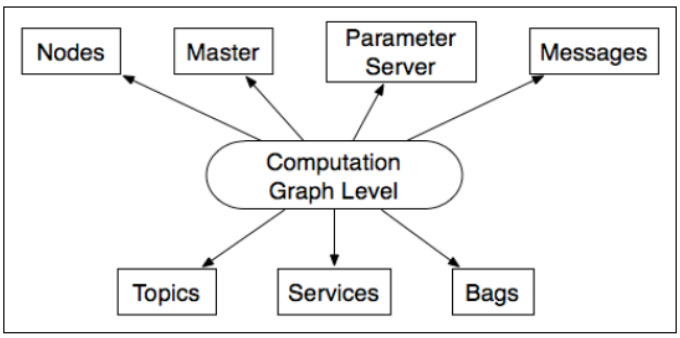


Figura 11 – Estrutura do nível *Computation Graph* [12]

2.2.3. NÍVEL *COMMUNITY*

Este terceiro nível aborda as ferramentas e conceitos para realizar a partilha de conhecimento, algoritmos e código de qualquer programador ROS. A comunidade consegue partilhar conhecimento e *software* através de recursos ROS. Estes recursos ROS são os seguintes:

- **Distribuições**: Distribuições ROS são coleções de versões de *stacks* que podem ser instaladas nas máquinas dos utilizadores;
- **Repositórios**: O ROS depende de redes de repositórios de código, onde diferentes instituições podem desenvolver e lançar os seus próprios componentes de *sofware* para robôs;
- **ROS *Wiki***: O ROS *Wiki* é o principal fórum para encontrar informação sobre o ROS. Cada indivíduo pode contribuir com a sua própria documentação e proporciona outros aspetos, como exemplo tutoriais;
- **Listas de correspondência**: Este é o principal canal de comunicação sobre novas atualizações para a ferramenta ROS, assim como um lugar para fazer questões sobre o mesmo. [12]

2.2.4. COMUNICAÇÃO *PUBLLISHER/SUBSCRIBER*

O principal modo de comunicação presente no ROS é definido como comunicação *publisher/subscriber*, que consiste num modelo de troca de mensagens entre nós, existindo nós com função de publicar mensagens e outros com função de subscrever mensagens. Os

nós utilizam tópicos para a transmissão de dados, podendo existir múltiplas publicações e subscrições para o mesmo tópico, em que o nó que transmite dados é denominado de *publisher* e o nó que lê esses dados é definido como *subscriber*. Na Figura 12 pode-se visualizar uma ilustração de como se pode realizar a comunicação *publisher/subscriber*. [13]

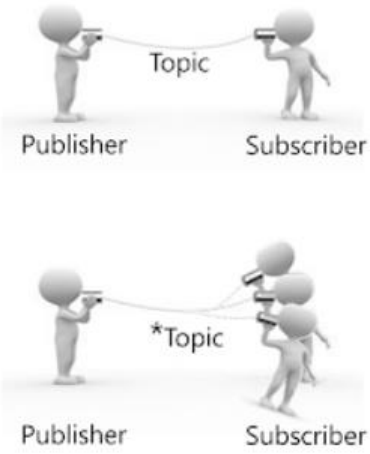


Figura 12 – Ilustração da comunicação *publisher/subscriber* [13]

Para se iniciar qualquer comunicação no sistema ROS é necessário executar o nó principal, ou seja, o *Master*. Este possui um registo de todos os nós do sistema e vai permitir que se encontrem uns aos outros. Cada vez que é criado um novo nó, este vai-se registar no *Master*, permitindo que outros nós o localizem, caso queiram trocar mensagens com este. A Figura 13 apresenta um esquema da comunicação *publisher/subscriber*, onde ambos os nós fazem o registo ao *Master* e começam a trocar mensagens através de um determinado tópico.

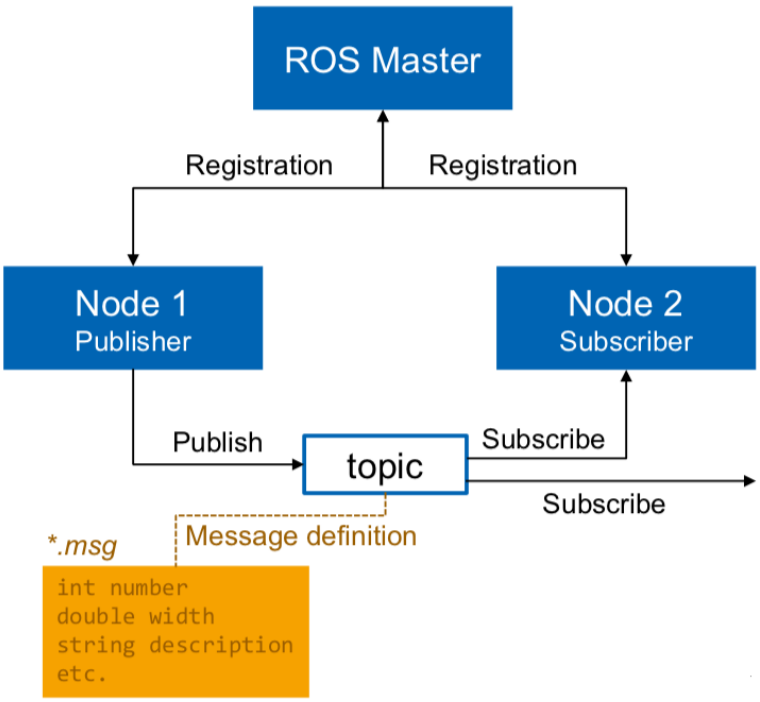


Figura 13 – Esquema da comunicação *publisher/subscriber* [14]

A Figura 14 apresenta um exemplo da comunicação *publisher/subscriber*, em que temos o nó *Camera Driver*, definido como *publisher*, que vai publicar mensagens através o tópico *Images* e os nós *Traffic Light Recognition* e *Pedestrian Detection*, definidos como *subscribers*, que vão receber as mensagens do tópico *Images*, estando estas a serem transmitidas pelo nó *Camera Driver*.

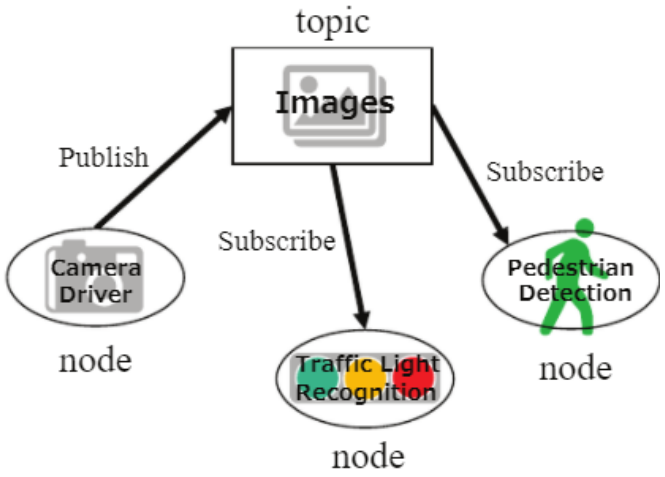


Figura 14 – Exemplo da comunicação *publisher/subscriber* [15]

3. FASES DO PROJETO

Este capítulo vai introduzir as principais fases que foram necessárias para a realização deste projeto, assim como os componentes constituintes dessas fases, fazendo a sua descrição no âmbito do projeto. As fases por onde este projeto se desenrola são as seguintes: programação do simulador *Turtlesim*, utilização de um simulador do robô *TurtleBot* e programação do robô real, estando planeado a utilização do robô Magni, descrito no Capítulo 2. A Figura 15 apresenta um diagrama de blocos que demonstra a ligação das diferentes fases do projeto.

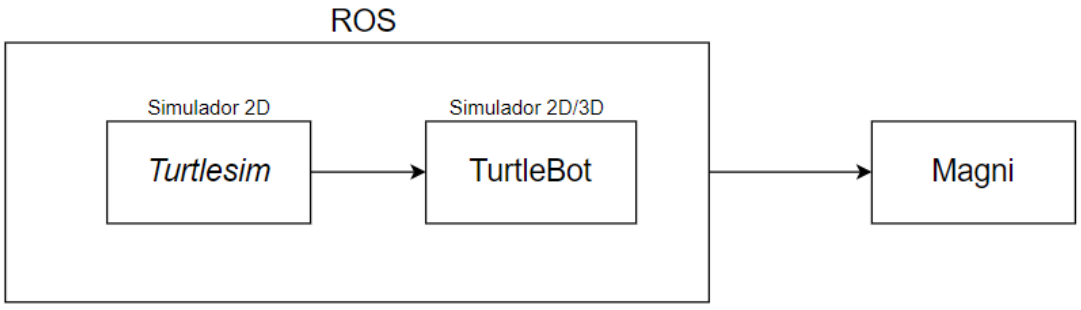


Figura 15 – Diagrama de blocos

Estando compreendido a comunicação *publisher/subscriber*, presente no Capítulo 2, vai se proceder à criação dos programas necessários para o desenvolvimento deste projeto, recorrendo ao simulador robótico 2D *Turtlesim*, onde vão ser testados esses mesmos

programas. Depois dos programas se encontrarem funcionais, estes vão ser modificados para serem novamente testados, sendo que desta vez é utilizado um simulador um pouco mais realista denominado de *TurtleBot Simulator*. Concluindo os testes no ambiente de simulação, vai-se aplicar os programas desenvolvidos, após as devidas alterações, ao robô Magni, de forma a conseguir criar um sistema de navegação mais robusto.

Para estudar e realizar simulações na ferramenta ROS é usado o sistema operativo Ubuntu, através da criação de uma máquina virtual, no programa *Oracle VM VirtualBox*. Como já foi referido, é possível utilizar as linguagens de programação C++ e *python* para desenvolver programas em ROS, sendo escolhida a linguagem *python* para a realização deste projeto.

3.1. SIMULADOR *TURTLESIM*

No início, vai ser usado o simulador *Turtlesim* para criar os primeiros programas para o desenvolvimento deste projeto. Este simulador é um pacote que vem acompanhado com a instalação do ROS e consiste num robô com forma de tartaruga, como se poder ver na Figura 16, num ambiente 2D, que se pode mover através do teclado ou comandos do ROS. Apesar de ser um simulador um pouco primitivo em relação a um robô real, é uma ferramenta útil para uma aprendizagem inicial do ROS. [16]

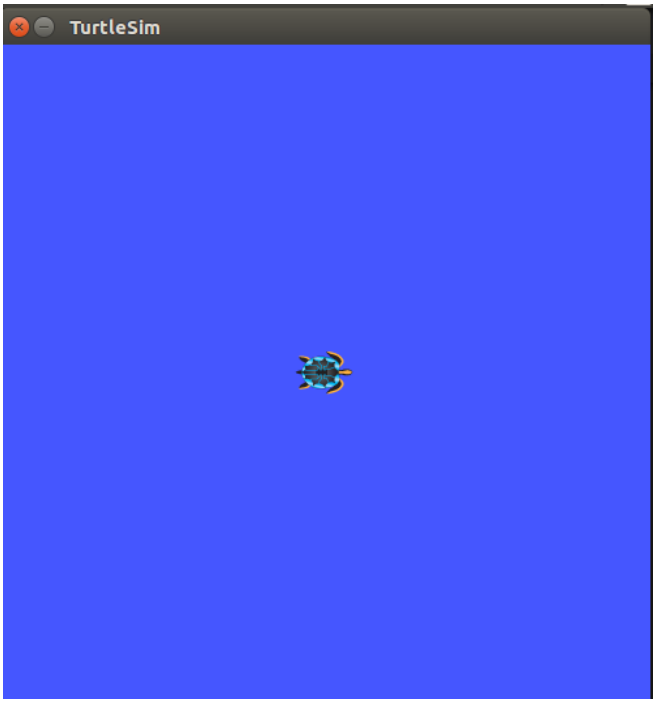


Figura 16 – Simulador *Turtlesim*

Para a realização deste projeto, decidiu-se desenvolver pequenos programas para testar o comportamento do robô, que posteriormente serão agrupados para a criação de aplicações. Os programas que vão ser desenvolvidos são os seguintes:

- **Mover em linha reta**: o robô move-se numa linha reta, para a frente ou para trás, a uma certa distância com uma certa velocidade;
- **Rotação**: o robô gira num certo ângulo, direção dos ponteiros do relógio ou contrária aos ponteiros do relógio, com uma certa velocidade;
- **Ponto destino**: o robô desloca-se para um determinado ponto da sua área, através da indicação de coordenadas.

3.2. SIMULADOR *TURTLEBOT*

Para além do *Turtlesim*, existem outros tipos de simuladores para programação de robôs, sendo os simuladores *Gazebo* e *Stage* os mais utilizados. Para este projeto decidiu-se usar o tipo *Stage* e *Rviz* (*ROS-visualization*) para efetuar as devidas simulações, considerando que a utilização do simulador *Gazebo* seria mais benéfico para a resolução deste projeto. O *Gazebo* oferece um ambiente mais próximo da realidade, mas optou-se por não lhe dar utilidade, devido às limitações tecnológicas que foram detetadas na máquina virtual quando esta executava este simulador.

O *Rviz* apresenta um ambiente 3D, com capacidade de exibir ao utilizador o que o robô está a ver, fazer e pensar, permitindo ao programador ter uma melhor perceção do mudo através do robô. O *Stage*, como o *Turtlesim*, é um simulador 2D, mas com mais semelhanças à realidade, contendo obstáculos, robôs mais robustos, entre outros aspetos. Na Figura 17 e na Figura 18 pode-se visualizar o simulador *Rviz* e na Figura 19 pode-se visualizar o simulador *Stage*, em que o tipo de robô utilizado é o *TurtleBot2*. [17][18]

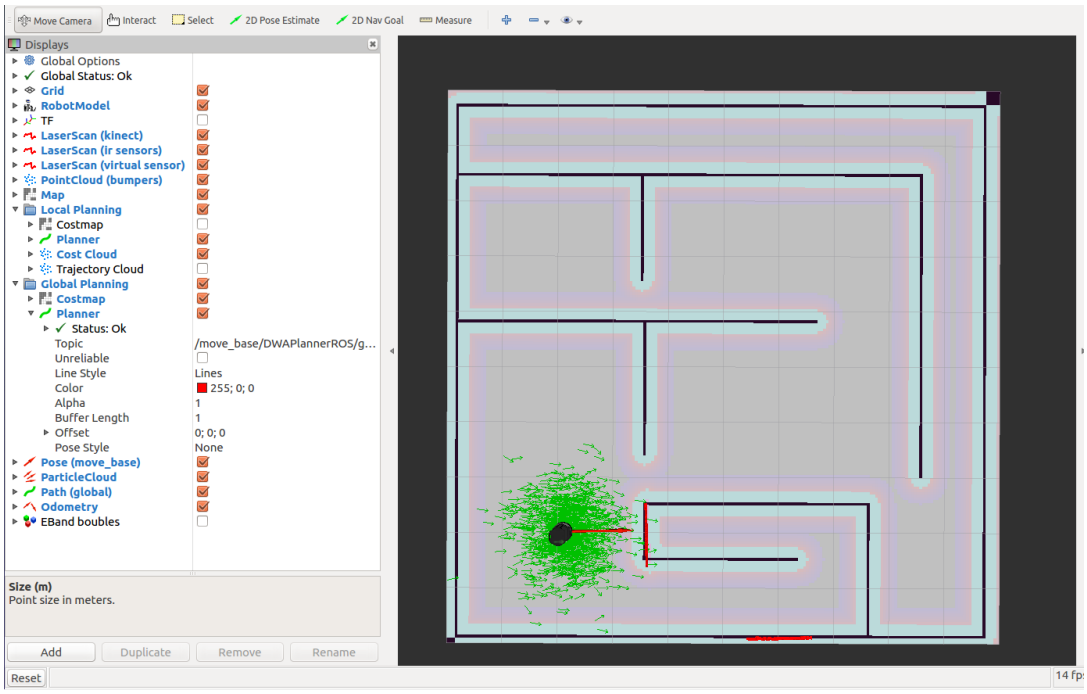


Figura 17 – Simulador *TurtleBot (Rviz_1)*

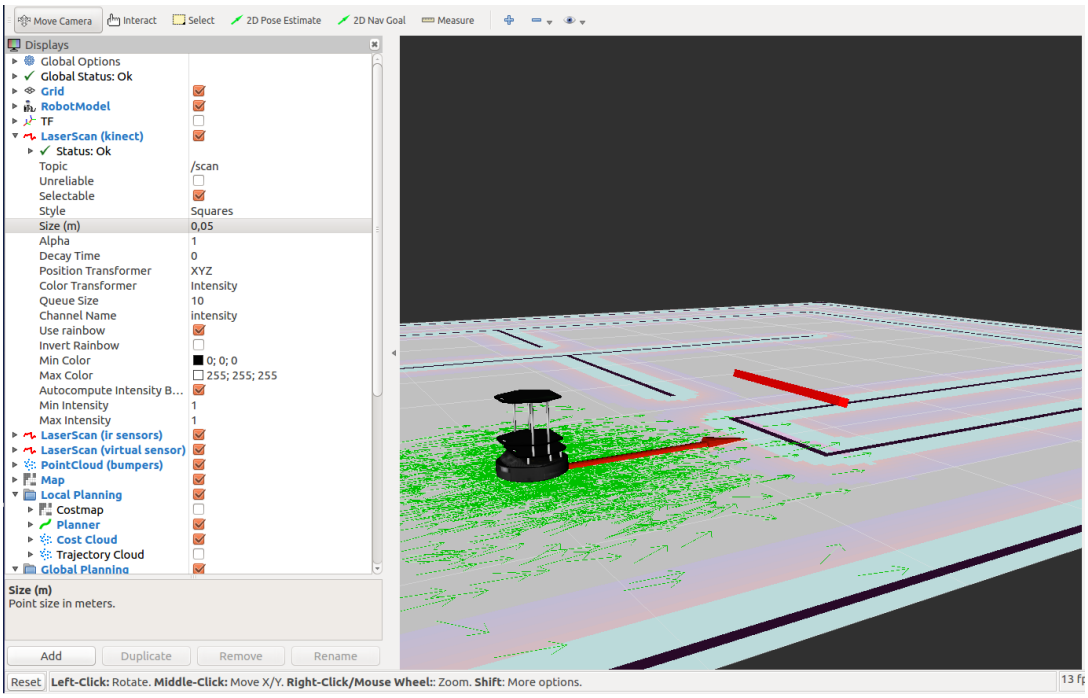


Figura 18 – Simulador *TurtleBot (Rviz_2)*

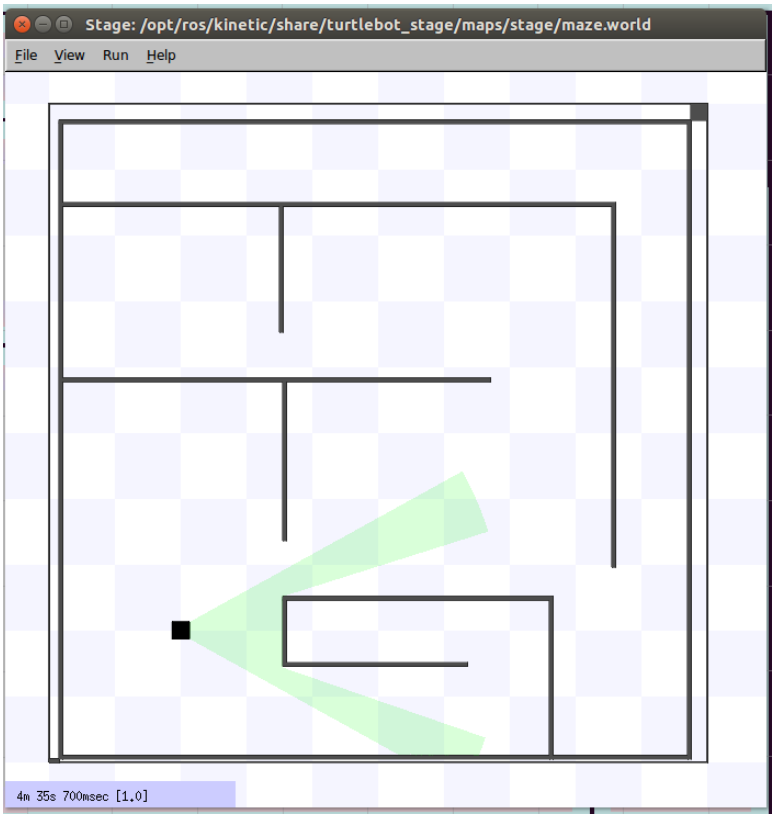


Figura 19 – Simulador *TurtleBot (Stage)*

O código desenvolvido nos programas do simulador *Turtlesim* vai ser reutilizado para criar novos programas no simulador *TurtleBot*, realizando apenas algumas alterações necessárias, como por exemplo mudar o nome dos tópicos. Este simulador, ao contrário do *Turtlesim*, possui sensores incorporados no robô, o que nos permite desenvolver novos programas com acréscimo dos mesmos, ficando um passo mais perto da realidade. Os programas que vão ser criados são os seguintes:

- **Passo a passo**: agrupando os programas **Mover em linha reta** e **Rotação**, o robô desloca-se até um determinado local, alternando entre movimentos lineares e rotacionais;

- **Sensor laser**: o robô vai estar em constante movimento, evitando colisões com as paredes;

- **Destino sensor**: vai ser agrupado ao programa **Ponto destino** o programa **Sensor laser**, permitindo ao robô deslocar-se até a um ponto predefinido, evitando colisões com as paredes.

3.3. MAGNI

Como já foi referido no início deste capítulo, vai ser usado o robô Magni para o desenvolvimento deste projeto, de forma a conseguir transferir os programas criados no ambiente de simulação para o mundo real.

Foi desenvolvida pela empresa deste robô uma máquina virtual especificamente equipada para programar o Magni, sendo esta composta pelo sistema operativo Ubunto, versão 16.04 e pela versão Kinetic do ROS, contendo também alguns programas para mover o robô, como por exemplo controlá-lo através do teclado do computador. Devido à tecnologia incorporada no Magni, este consegue realizar uma ligação *wireless* à máquina que o vai programar. [19]

A câmara Raspberry Pi instalada no robô permite que este detete imagens de código *Quick Response* (QR), possibilitando a construção de um sistema de localização, ou seja, o robô consegue guardar as localizações das imagens de código QR diferentes que deteta. Os sonares que este possui vão permitir que este evite colisão com objetos e paredes que estejam presentes no seu espaço e toda a computação do robô é realizada pelo Raspberry Pi 3. [20]

Os programas criados nos dois simuladores vão ser adaptados para poderem ser executados pelo Magni. Numa fase inicial, vão ser modificados os programas **Mover em linha reta** e **Rotação**, desenvolvidos no simulador *Turtlesim*, para testar os movimentos básicos do robô, e na próxima fase vai-se adaptar os programas **Sensor laser** e **Destino sensor**, desenvolvidos no simulador *TurtleBot*, para testar o comportamento do robô com os sonares. O processo de adaptação dos programas vai ser semelhante ao do *TurtleBot*, que vai consistir em fazer as alterações necessárias ao código, para que este fique apto para ser interpretado e executado pelo robô.

Conseguindo ter sucesso na execução dos programas, pode-se proceder para o desenvolvimento de novos programas, mais direcionados para outras características do Magni, de forma a ter o máximo aproveitamento das capacidades do robô.

4. *TURTLESIM*

Este capítulo vai apresentar a implementação que foi realizada no simulador *Turtlesim*, isto é, o capítulo vai abordar os passos necessários para o desenvolvimento dos programas no *Turtlesim*, assim como partes do código desses mesmos programas. Para além do que já foi descrito, também vão ser apresentados os resultados obtidos através da execução dos programas.

4.1. TÓPICOS

Antes de iniciar a criação dos programas, tem de se descobrir quais os tópicos que vão ser utilizados e explorá-los, de forma a saber o tipo de mensagens que estes transmitem e os parâmetros que possuem. A primeira operação a realizar é iniciar o *Master*, através do comando roscore, porque sem o *Master* estar ativo não é possível realizar operações em ROS. Para ter acesso aos tópicos disponíveis pelo *Turtlesim*, é necessário ativar o nó do mesmo com o comando rosrun turtlesim turlesim_node, podendo-se ver o resultado na Figura 20. Outra forma de certificar que o nó está a ser executado é através do comando rosnode list, em que são listados todos os nós ativos, como se pode visualizar na Figura 21.

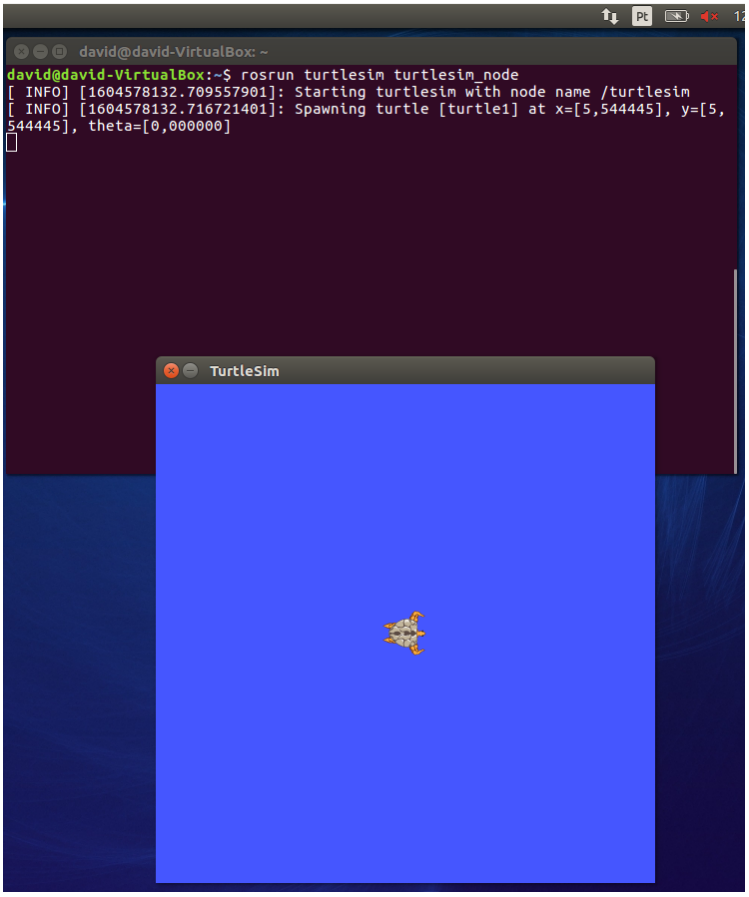


Figura 20 – Nó *Turtlesim*

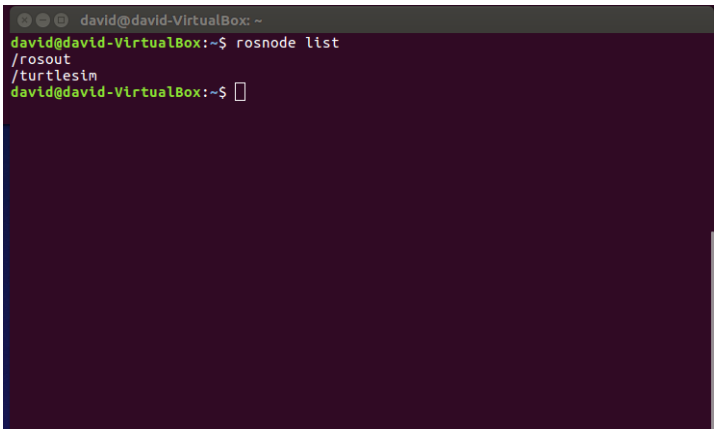


Figura 21 – Lista dos nós

Estando ativo o nó *Turtlesim*, já possível aceder aos tópicos deste simulador, de forma a saber quais os que publicam e subscrevem a este nó. Com o comando rostpoic list pode-se ver todos os tópicos disponíveis, como se pode visualizar na Figura 22. A Figura 23 apresenta um esquema da interligação dos tópicos com o nó *Turtlesim*.

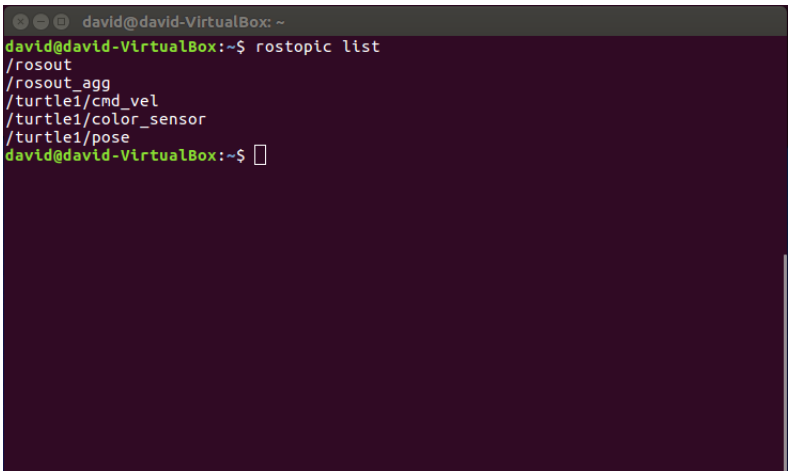


Figura 22 – Lista dos tópicos

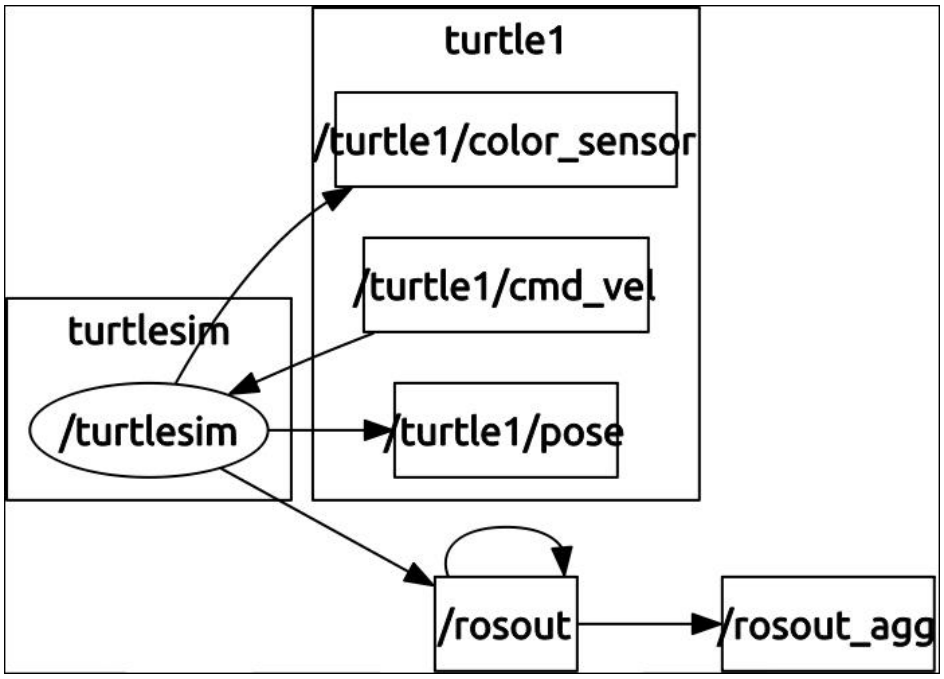


Figura 23 – Esquema do *Turtlesim* (nó e tópicos) [27]

Através do esquema da Figura 23, consegue-se perceber quais os tópicos que publicam e que subscrevem para o nó *Turtlesim*, sendo que os tópicos que publicam são o /turtle1/color_sensor e /turtle1/pose e o único que subscreve é o /turtle1/cmd_vel. Outra forma de descobrir quais os tópicos que publicam e subscrevem, é através da execução do comando rostopic info /turtle1/cmd_vel, usando o tópico /turtle1/cmd_vel como exemplo. Com este comando também conseguimos saber o tipo de mensagem que este tópico utiliza, como se pode confirmar na Figura 24.

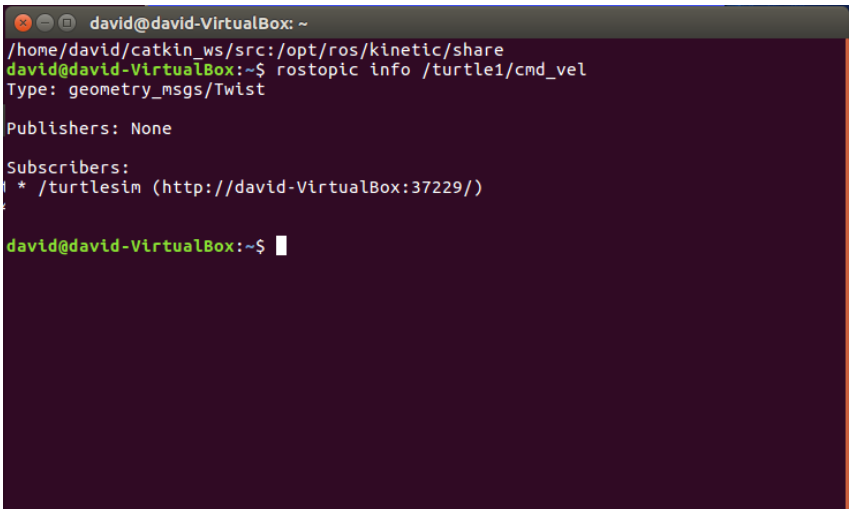


Figura 24 – Informação do tópico /turtle1/cmd_vel

A Figura 24 diz que a mensagem que é transmitida por este tópico é do tipo *Twist*. Utilizando este tipo de mensagem como exemplo, pode-se usar o comando rosmsg show geometry_msgs/Twist para visualizar os parâmetros de qualquer mensagem, podendo ver o resultado na figura seguinte.

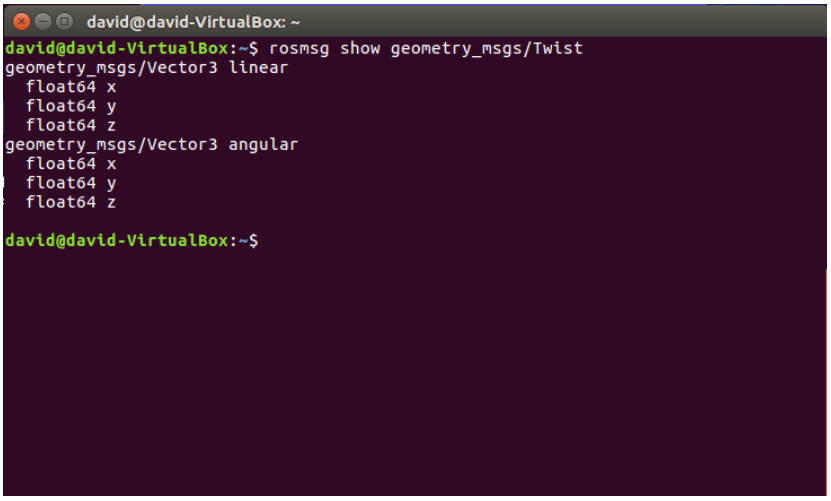


Figura 25 – Parâmetros da mensagem *geometry_msgs/Twist*

Tendo em consideração os objetivos dos programas que se pretendem desenvolver, foram necessários estudar dois tópicos, que são: /turtle1/cmd_vel e /turtle1/pose. O tópico /turtle1/cmd_vel, denominado de *comand velocity*, é utilizado para mover o robô e o tópico /turtle1/pose é usado para detetar a posição do robô.

Dado o uso do tópico /turtle1/cmd_vel como exemplo para demonstrar os comandos descritos anteriormente, já se obteve a informação sobre este. Como foi referido

previamente, o tópico subscreve para o nó *Turtlesim*, com mensagens do tipo *Twist*. A mensagem do tipo *Twist* é composta por dois parâmetros, velocidade linear e velocidade angular, como se pode ver na Figura 25. As coordenadas presentes em cada um dos parâmetros identifica que o robô pode realizar movimentos em três dimensões, quer sejam movimentos lineares ou angulares, de acordo com os eixos presentes na Figura 26. Sendo o *Turtlesim* um simulador em duas dimensões, não faz sentido o uso de determinadas coordenadas, contando que só vão ser utilizadas as coordenadas necessárias para cumprir os objetivos dos programas.

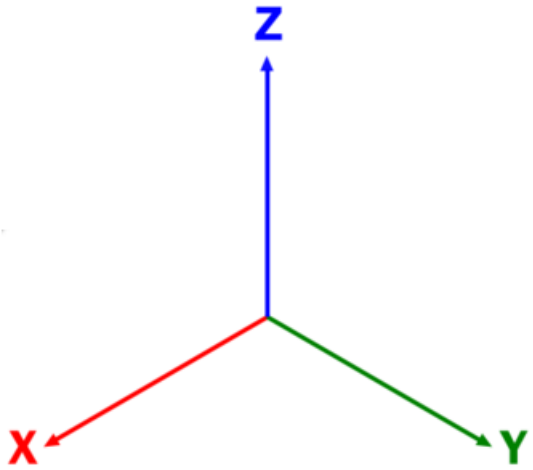


Figura 26 – Eixo de coordenadas 3D

Para descobrir informações sobre tópico /turtle1/pose, aplicaram-se os comandos descritos anteriormente, estando o resultado visível na Figura 27. Tendo já sido mencionado que este tópico publica para o nó *Turtlesim*, a Figura 27 também mostra que a mensagem que este transmite é do tipo *Pose* e este tipo apresenta as coordenadas do robô no seu ambiente, assim como a sua rotação e as suas velocidades.

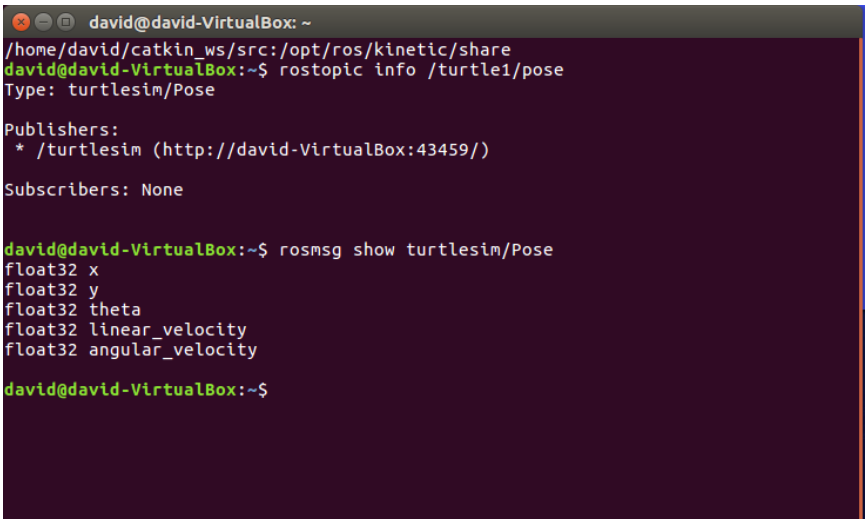


Figura 27 – Informação do tópico /turtle1/pose

4.2. CÓDIGO DESENVOLVIDO

Estando os tópicos definidos, passou-se para o desenvolvimento dos programas planeados. Os três programas possuem características comuns, como os tópicos e as bibliotecas importadas. Cada programa é definido como um nó e todos realizam publicações para o tópico de velocidade /turtle1/cmd_vel, sendo que o programa **Rotação** é o único que não subscreve para o tópico de posição /turtle1/pose.

As bibliotecas são importadas da seguinte forma:

```
import rospy
from geometry_msgs.msg import Twist
from turtlesim.msg import Pose
import math
import time
```

A primeira linha diz respeito à biblioteca necessária para se poder desenvolver um programa em ROS com a linguagem *python*. Faz-se também a importação do tipo de mensagem dos tópicos que vão ser usados, assim como a importação da biblioteca para realizar operações aritméticas.

Para declarar a publicação para tópico /turtle1/cmd_vel são introduzidas as seguintes linhas:

```
topico_velocidade = '/turtle1/cmd_vel'
```

```
publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)
```

Na declaração de um *publisher*, é necessário identificar o nome do tópico que vai ser publicado, o tipo de mensagem que vai ser transmitida e especificar o número de mensagens que consegue guardar antes de serem consumidas.

Para identificar a subscrição para o tópico /turtle1/pose usa-se os seguintes comandos:

```
topico_posicao = '/turtle1/pose'
subscriber_posicao = rospy.Subscriber(topico_posicao, Pose, poseCallback)
```

Como no *publisher*, para declarar um *subscriber* é necessário identificar o nome do tópico e o tipo de mensagem que é transmitida, sendo que este possui uma função de *callback*, uma função que é executada automaticamente assim que é recebida uma nova mensagem pelo *subscriber*.

A função *callback* desenvolvida nos programas **Mover em linha reta** e **Ponto destino** é a seguinte:

```
def poseCallback(posicao):
    global x
    global y, yaw
    x = posicao.x
    y = posicao.y
    yaw = posicao.theta
```

Esta função vai fazer uma atualização constante dos valores da posição e orientação do robô.

4.2.1. MOVER EM LINHA RETA

Este programa vai ter como objetivo o movimento linear do robô, que como já foi referido no capítulo anterior, vai fazer com que o robô execute um movimento em linha reta para uma determinada distancia.

A função criada para realizar o que foi descrito é definida da seguinte forma:

```
def mover(publisher, velocidade, distancia, mover_frente):
```

O utilizador, quando chamar esta função, deve preencher os devidos atributos como pretender, identificando o nome do *publisher* que pretende usar, a velocidade com que o robô se desloque, a distancia que este vai percorrer e a direção do movimento.

A mensagem do tipo *Twis*t é declarada da seguinte forma:

```
mensagem_velocidade = Twist()
```

Dos parâmetros presentes neste tipo de mensagem, o que vai ser utilizado para dar movimento ao robô é o parâmetro da velocidade linear na coordenada X. Na seguinte condição pode-se ver um exemplo de como é programado este parâmetro, que neste caso, este terá um valor positivo ou negativo, dependendo da direção que o robô vai tomar.

```
if (mover_frente):
     mensagem_velocidade.linear.x = abs(velocidade)
else:
     mensagem_velocidade.linear.x = -abs(velocidade)
```

O *publisher* selecionado vai publicar a mensagem da seguinte forma:

```
publisher.publish(mensagem_velocidade)
```

Para verificar que o robô percorre a distância pretendida, é realizado um calculo constante do percurso que este está a fazer, através da equação (1), colocando depois uma condição em que o robô só deixa de realizar movimento quando a distancia deste percurso for maior que a distancia pretendida. Esta condição é apresentada da seguinte forma:

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

```
distancia_movida = abs(math.sqrt(((x-x0) ** 2) + ((y-y0) ** 2)))


if not (distancia_movida < distancia):
     rospy.loginfo("Distanica percorrida")
     break
```

Devido à função *callback*, consegue-se ter uma constante atualização da posição do robô enquanto este se move. Uma vez alcançada a distância desejada, a velocidade linear em X toma o valor 0.

Na função *main*, para além de chamar a função e de declarar os tópicos, também é feita a inicialização deste nó, sendo esta realizada através da seguinte linha de código:

```
rospy.init_node('turtlesim_move_linha_reta', anonymous=True)
```

No anexo 1 está presente o código completo deste programa.

4.2.2. ROTAÇÃO

O foco deste programa é fazer com que o robô realize um movimento rotacional, isto é, o robô vai girar num determinado ângulo na sua posição.

A função criada para este programa é a seguinte:

```
def rotacao(publisher, velocidade_angular_degree, angulo_degree,
ponteiros_relogio):
```

Assim como a função do programa anterior, o utilizador deve preencher o atributos da função quando a chama, identificando o nome do *publisher*, a velocidade angular com que este se desloca, o ângulo que se pretende que ele gire e a direção do movimento.

A mensagem do tipo *Twis*t é declarada da mesma forma como no programa anterior, sendo que o parâmetro utilizado para girar o robô é a coordena Z da velocidade angular. Antes de atribuir o valor da velocidade a este parâmetro, é necessário fazer a conversão de graus para radianos, porque o ROS utiliza esta unidade para os ângulos.

A direção para que o robô vai girar é definida pela seguinte condição:

```
if (ponteiros_relogio):
     mensagem_velocidade.angular.z = -abs(velocidade_angular)
else:
     mensagem_velocidade.angular.z = abs(velocidade_angular)
```

A mensagem publicada pelo *publisher* é feita da mesma forma que no programa anterior, sendo também aplicado uma condição semelhante ao primeiro programa, com o objetivo de certificar que o robô faz a rotação do ângulo selecionado e que termina o movimento após essa rotação estar concluída. Vai ser constantemente calculado o ângulo atual do robô, que é feito através do produto da diferença de tempo, em segundos, desde que o robô o inicia o movimento com a velocidade angular, até este ser maior ou igual ao ângulo pretendido. Esta descrição é apresentada da seguinte forma:

```
angulo_atual_degree = (t1-t0) * velocidade_angular_degree


if (angulo_atual_degree >= angulo_degree):
      rospy.loginfo("Rotacao concluida")
      break
```

Quando a rotação desejada for atingida, a velocidade angular em Z toma o valor 0.

Este nó é inicializado da seguinte forma:

```
rospy.init_node('turtlesim_rotacao', anonymous=True)
```

O anexo 2 apresente o código deste programa na sua totalidade.

4.2.3. PONTO DESTINO

Este programa tem como objetivo a programação do robô para que este realize um movimento até a um determinado ponto no seu espaço. Este ponto é indicado através de coordenadas em X e em Y. Para saber as coordenadas iniciais do robô, usou-se o comando rostopic echo /turtle1/pose, comando usado para mostrar as mensagens publicadas por um tópico, como se pode visualizar na Figura 28. Através desta figura, pode-se observar que as coordenadas iniciais do robô são aproximadamente (5.5,5.5), tendo a Figura 29 como reforço visual.

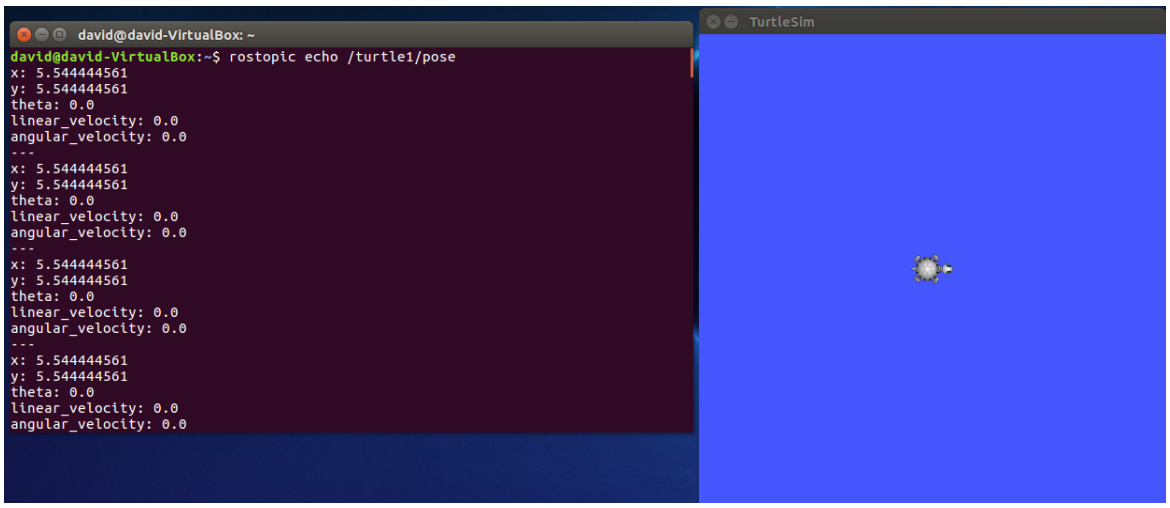


Figura 28 – Comando *echo* no tópico /turtle1/pose

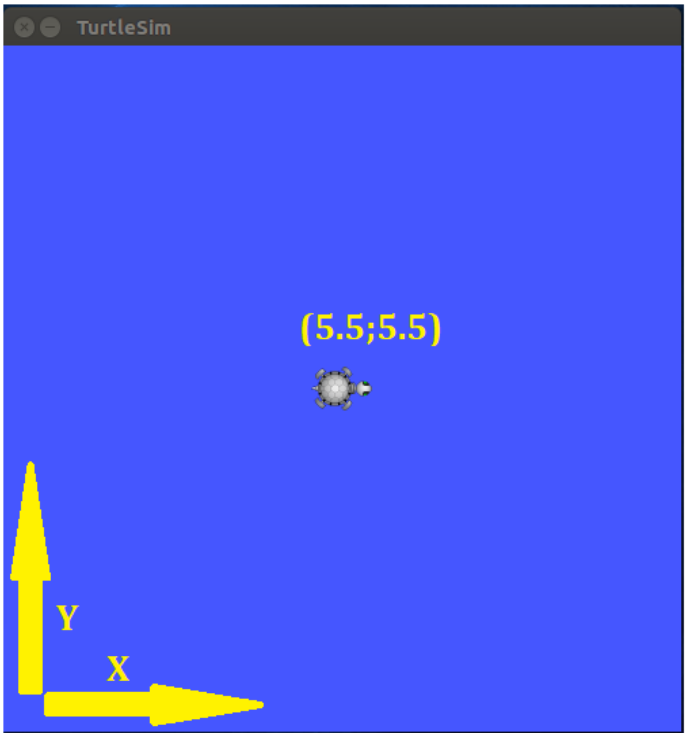


Figura 29 – Coordenadas iniciais do *Turtlesim*

A função estabelecida para este programa é a seguinte:

```
def ponto_destino(publisher, coordenada_x, coordenada_y):
```

Semelhante aos programas anteriores, o utilizador, ao chamar a função, deve identificar o nome do *publisher*, as coordenadas X e as coordenadas Y, sendo estas as coordenadas que o robô vai alcançar.

Como no programa **Mover em linha reta**, este possui a função *callback*, obtendo uma constante atualização dos valores da posição do robô, tendo também a mesma declaração do tipo de mensagem *Twist*.

Neste programa tem que se usar os dois parâmetros da mensagem *Twist* que foram usados nos programas anteriores, ou seja, vai se usar o parâmetro da velocidade linear em X e o parâmetro da velocidade angular em Z. Para o cálculo de ambas as velocidades, é implementado um controlador proporcional, com o intuito de tornar mais suave o movimento do robô.

A velocidade linear é obtida através do produto da distância, calculada através da equação (1), com o ganho proporcional. Esta descrição é feita da seguinte forma:

```
K_linear = 0.5
```

```
distancia=abs(math.sqrt(((coordenada_x-x)**2)+((coordenada_y-
y)**2)))
velocidade_linear = distancia * K_linear
```

O mesmo caso é aplicado para o cálculo da velocidade angular, sendo que esta é o produto do ganho proporcional com o ângulo desejado, que é adquirido através da função matemática arco tangente da diferença de coordenadas. Com o arco tangente consegue-se obter o ângulo entre dois vetores. O que foi descrito foi realizado da seguinte forma:

```
K_angular = 4.0
angulo_desejado = math.atan2(coordenada_y-y, coordenada_x-x)
velociade_angular = (angulo_desejado-yaw) * K_angular
```

A escolha do ganho proporcional tem de ser feita com cuidado, pois pode afetar o desempenho do robô.

Os parâmetros das velocidades são realizados pelas seguintes linhas de código:

```
mensagem_velocidade.linear.x = velocidade_linear
mensagem_velocidade.angular.z = velociade_angular
```

A inicialização deste nó é a seguinte linha de código:

```
rospy.init_node('turtlesim_ponto_destino', anonymous=True)
```

O anexo 3 possui todo o código deste programa.

4.3. TESTES E RESULTADOS

Para testar os programas, foram previamente definidos parâmetros pelo utilizador, sendo que se poderia alterar o código para que o programa pergunta-se ao utilizador que valores pretende inserir.

No programa **Mover em linha reta**, decidiu-se que o robô percorria uma distância de 5.0 m a uma velocidade de 5.0 m/s, com direção frontal. Na Figura 30 pode-se visualizar o resultado obtido deste teste.

```
mover(publisher_velocidade, 5.0, 5.0, True)
```

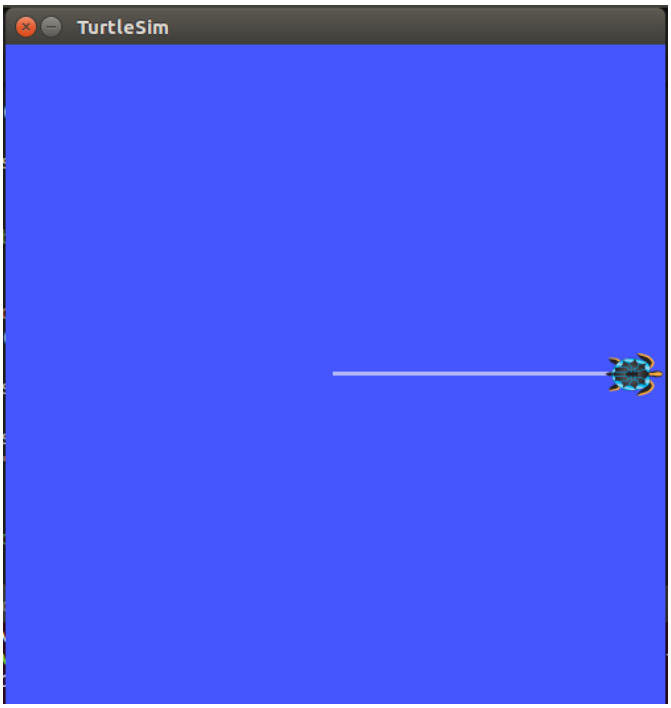


Figura 30 – Teste **Mover em linha reta**

No programa **Rotação**, definiu-se o robô vai girar 90º na direção contraria dos ponteiros do relógio, a uma velocidade angular de 15º. A Figura 31 apresenta o resultado obtido deste programa.

```
rotacao(publisher_velocidade, 15, 90, False)
```

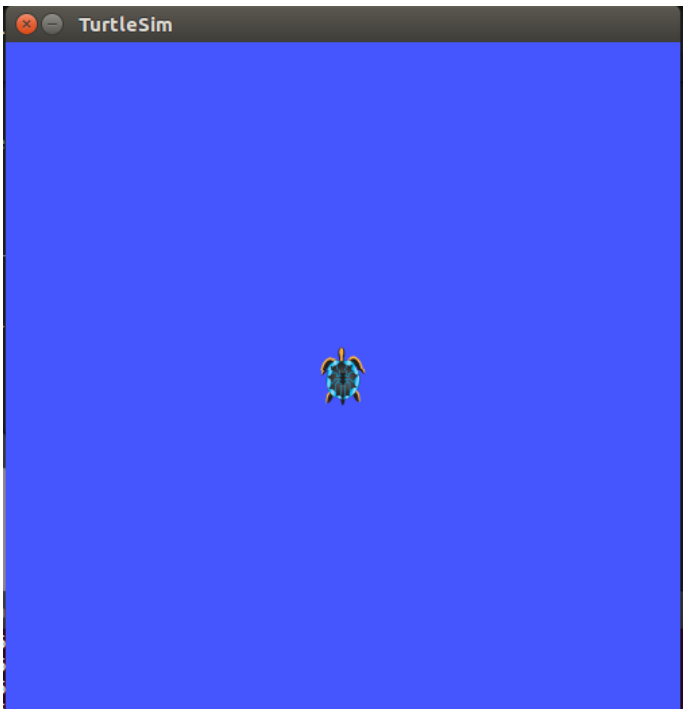


Figura 31 – Teste **Rotação**

Para testar o programa **Ponto destino**, foi decido que o robô tinha que se deslocar ao ponto (9.0,1.0) e o resultado que se obteve está presente na figura seguinte.

```
ponto_destino(publisher_velocidade, 9.0, 1.0)
```

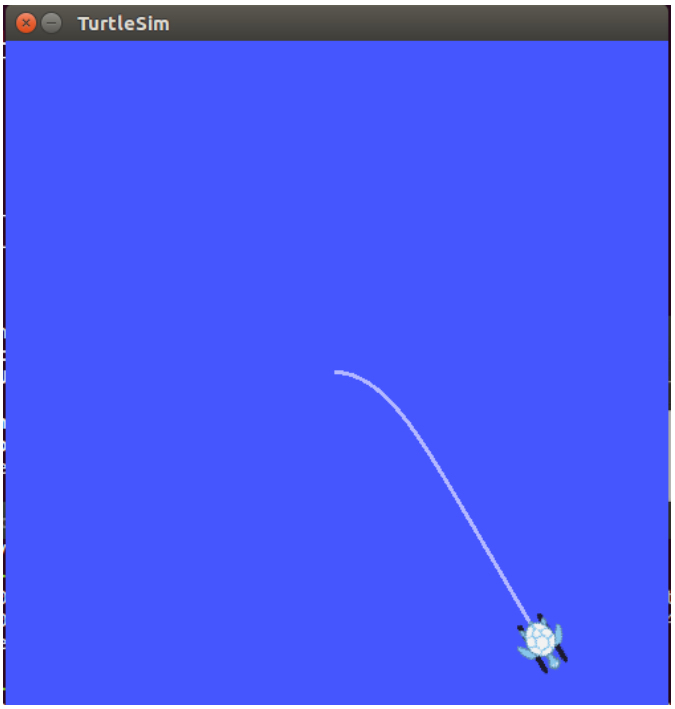


Figura 32 – Teste **Ponto destino**

Através da observação das Figuras 30, 31 e 32, pode-se concluir que todos os programas foram bem-sucedidos nas suas execuções, podendo passar-se para a próxima fase do projeto.

5. *TURTLEBOT*

Semelhante ao Capítulo 4, este novo capítulo vai apresentar a implementação que foi realizada no simulador *TurtleBot*, abordando o caminho percorrido para o desenvolvimento dos programas, descritos no Subcapítulo 3.3, e as alterações que foram feitas ao código dos programas do *Turtlesim*, para estes poderem serem compatíveis com o *TurtleBot*. No fim deste capítulo, vão ser apresentados os resultados obtidos pela execução dos programas.

5.1. TÓPICOS

Assim como no *Turtlesim*, teve de se saber quais os tópicos do *TurtleBot* que vão ser utilizados para o desenvolvimento dos programas. O procedimento é exatamente o mesmo que foi feito no *Turtlesim*, isto é, tem que se iniciar o *Master* e depois executar o simulador, sendo que é utilizado o comando roslaunch turtlebot_stage turtlebot_in_stage.launch, como se pode ver na Figura 33. Este *launch* inicia o simulador *Rviz* e o simulador *Stage*, estando isto visível na Figura 34.

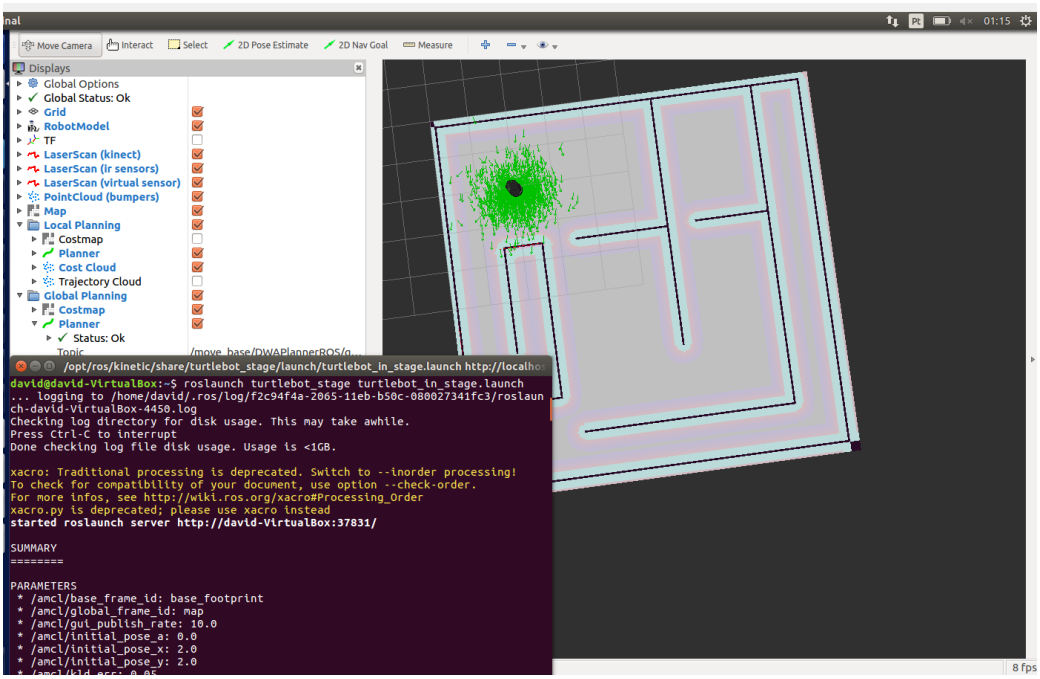


Figura 33 – Executar o comando para iniciar o simulador

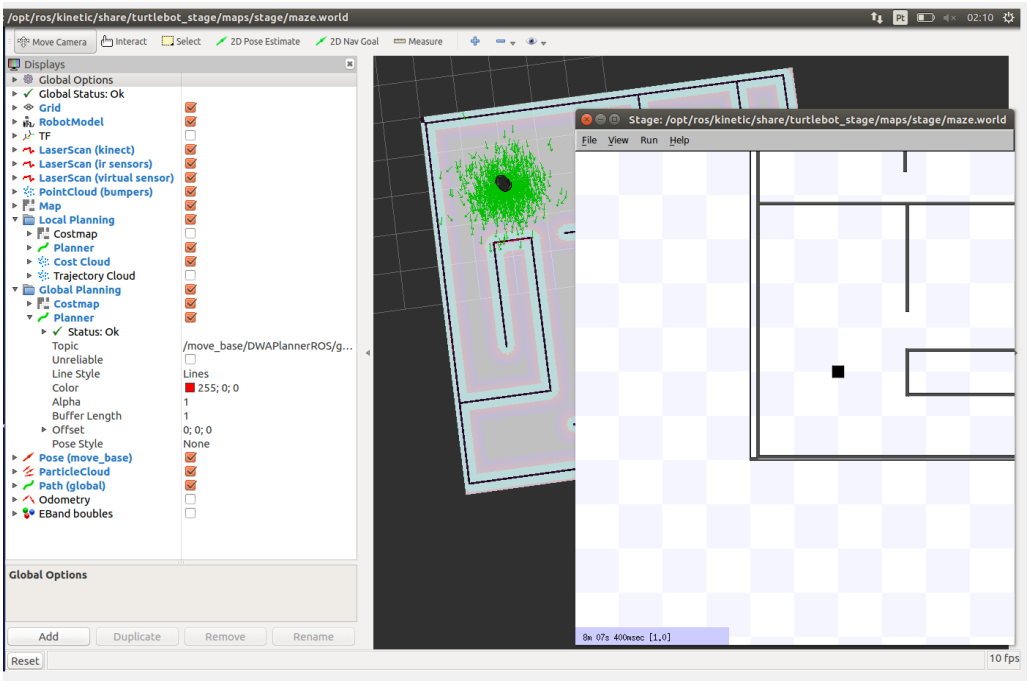


Figura 34 – Simulador *Rviz* e o simulador *Stage*

Com o simulador ativo, é possível visualizar os tópicos deste, recorrendo ao comando rostopic list, mas como vão ser reutilizados os programas desenvolvidos no *Turtlesim*, já se tem conhecimento dos tópicos que se pretende usar. Para mover o robô vai ser utilizado o tópico /cmd_vel_mux/input/teleop, para detetar a posição é usado o tópico /odom e para usufruir do sensor laser vai ser usado o tópico /scan. Para obter informações sobre os

tópicos foram usados os comandos rostopic info e rostopic echo, com os seus funcionamentos descritos no Capítulo 4.

Na figura seguinte são fornecidas informações sobre o tópico /cmd_vel_mux/input/teleop, podendo ver que este subscreve para o nó *mobile_base_nodelet_manager* e transmite mensagens do tipo *Twist*, sendo que os parâmetros deste tipo de mensagem já foram vistos no simulador *Turtlesim*.

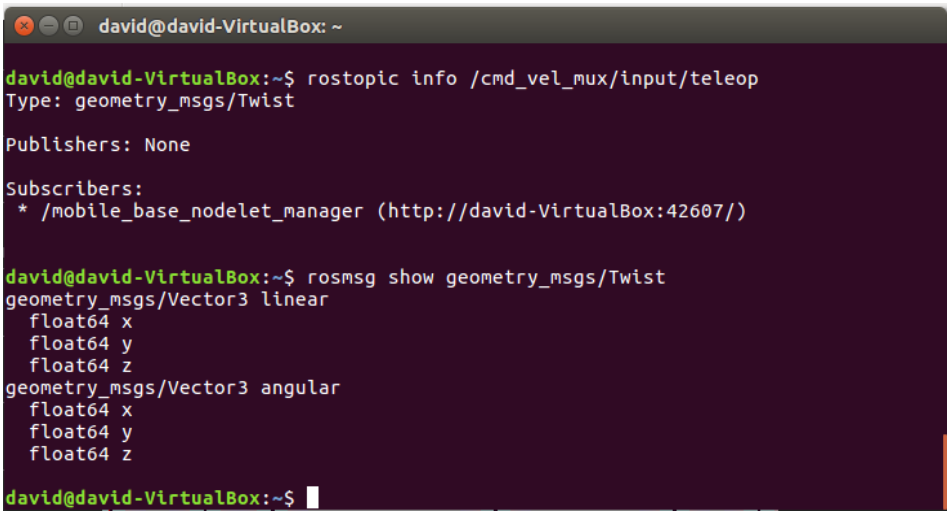


Figura 35 – Informação do tópico /cmd_vel_mux/input/teleop

O tópico /odom, do termo odometria, a estimativa da posição do robô através dos dados recolhidos por sensores de movimento, é publicado pelo nó *stageros* e subscrito pelos nós *mobile_base_nodelet_manager* e *move_base*, enviando mensagens do tipo *Odometry*, com se pode ver na Figura 36. A Figura 37 apresenta os parâmetros do tipo de mensagem *Odometry*, sendo que só vão ser utilizados os que estão selecionados, de forma a controlar a posição e orientação do robô.

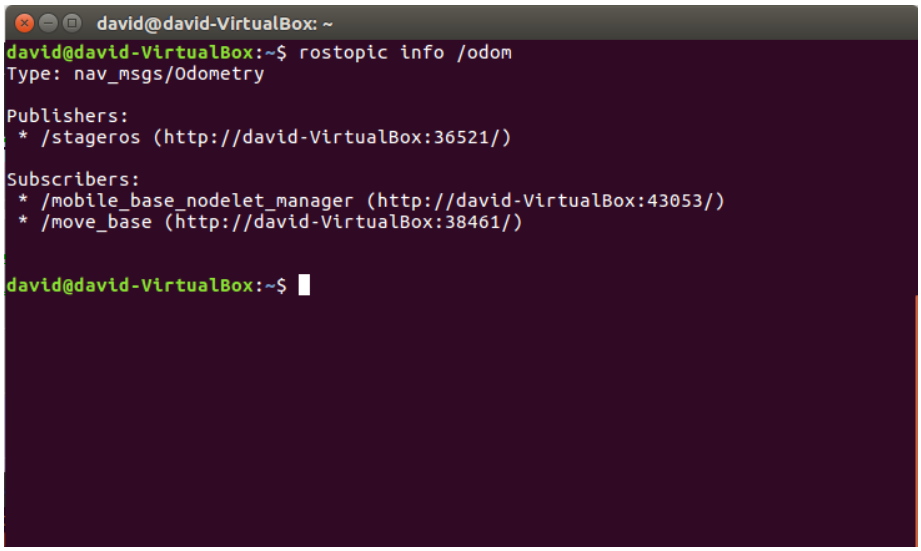


Figura 36 – Informação do tópico /odom

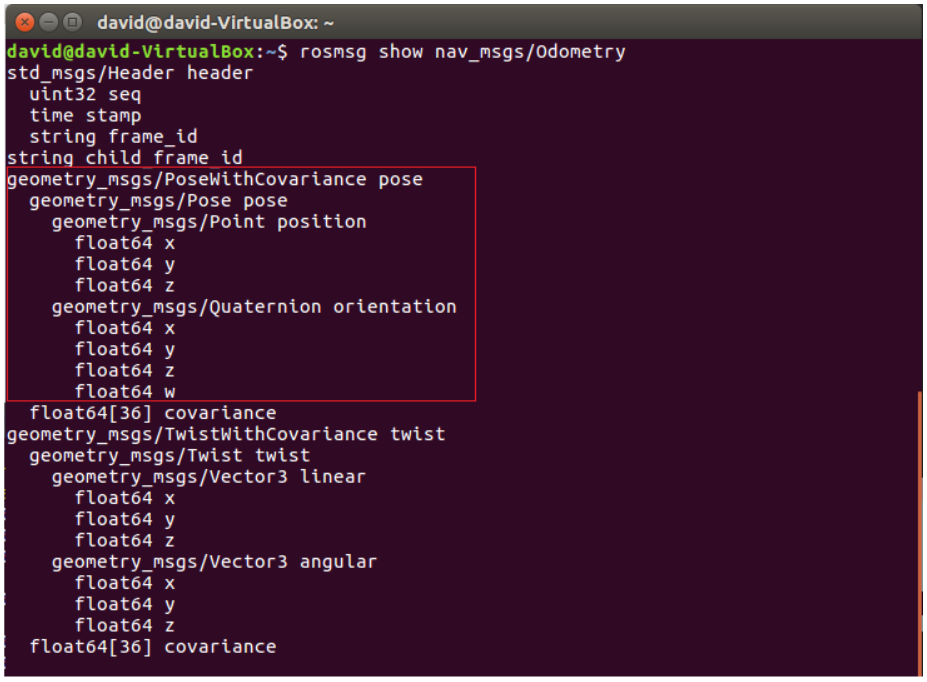


Figura 37 – Parâmetros da mensagem *nav_msgs/Odometry*

A Figura 38 demonstra os nós que publicam e subscrevem para o tópico /scan, assim como o tipo de mensagens que este transmite para esses nós, sendo do tipo *LaserScan*. Na Figura 38 também se pode ver os parâmetros constituintes do sensor laser, como por exemplo o ângulo máximo do sensor, o ângulo mínimo, o alcance máximo, entre outros.

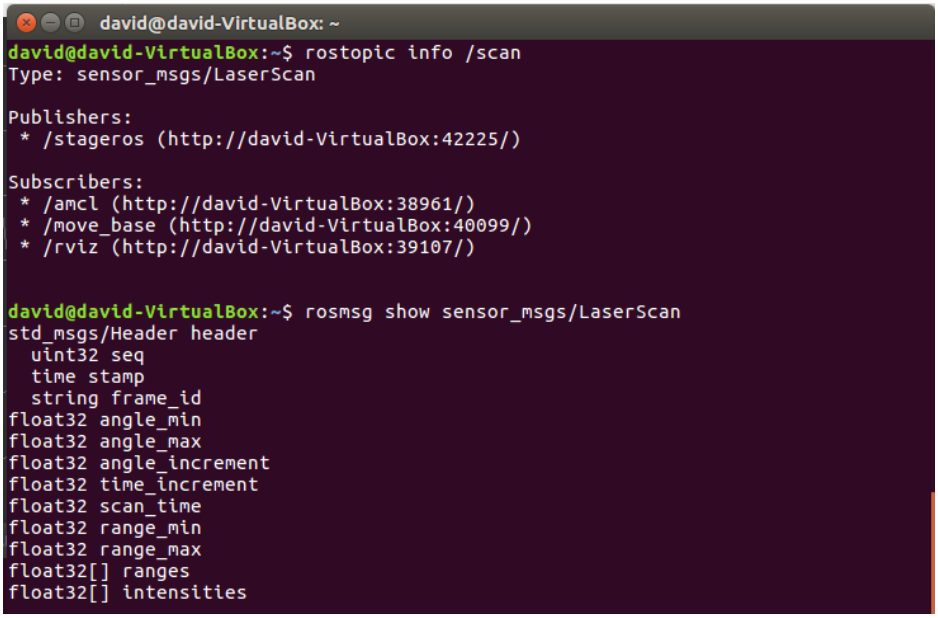


Figura 38 – Informação do tópico /scan

Recorrendo ao comando rostopic echo /scan consegue-se visualizar os valores dos parâmetros do sensor, obtendo assim a informação das características do sensor laser. A Figura 39 apresenta o resultado obtido pela aplicação do comando, sendo possível observar o seguinte: o sensor tem um alcance máximo de 5 m, não tem alcance mínimo, o ângulo mínimo do sensor é aproximadamente -0.5061 rad e o ângulo máximo é aproximadamente 0.5061 rad, estes valores equivalem a aproximadamente 29º.

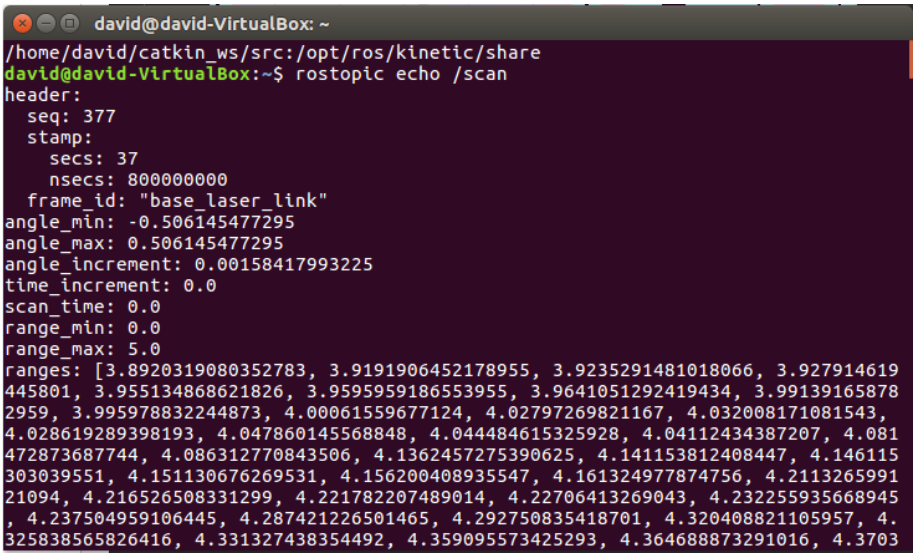


Figura 39 – Comando *echo* no tópico /scan

Um sensor laser permite saber a distância entre o robô e um obstáculo, através de raios laser, que são lançados em diferentes direções e ao atingirem um objeto ou uma superfície, ressaltam de volta. Na Figura 40 pode-se visualizar o sensor laser do *TurtleBot* no simulador *Stage*.

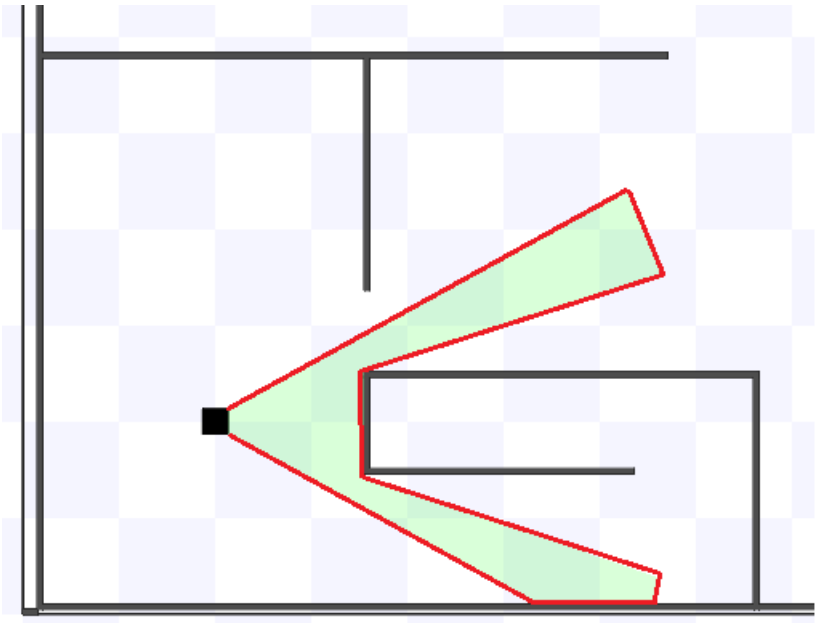


Figura 40 – Sensor laser no simulador *TurtleBot*

5.2. CÓDIGO DESENVOLVIDO

Com a definição dos tópicos, procedeu-se ao desenvolvimento dos programas projetados. Semelhante aos programas do *Turtlesim*, os programas desenvolvidos neste simulador possuem características idênticas, sendo estas as bibliotecas importadas e os tópicos. Todos os programas efetuam publicações para o tópico velocidade /cmd_vel_mux/input/teleop, mas só dois é que subscrevem para o tópico de posição /odom, sendo estes o **Passo a passo** e o **Destino sensor**, e também só dois realizam subscrição para o tópico de sensor /scan, sendo o caso do **Sensor laser** e **Destino sensor**.

A importação das bibliotecas é feita da seguinte forma:

```
import rospy
from geometry_msgs.msg import Twist
import math
import time
from nav_msgs.msg import Odometry
```

```
from sensor_msgs.msg import LaserScan
```

A declaração do *publisher* para o tópico /cmd_vel_mux/input/teleop é realizada através da introdução das seguintes linhas:

```
topico_velocidade = '/cmd_vel_mux/input/teleop'
publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)
```

Para fazer a subscrição para o tópico /odom usa-se o seguinte código:

```
topico_posicao = '/odom'
subscriber_posicao = rospy.Subscriber(topico_posicao, Odometry, poseCallback))
```

A função *callback* desenvolvida para atualizar os valores da posição e orientação do robô é a seguinte:

```
def poseCallback(posicao):
    global x
    global y
    global z
    x = posicao.pose.pose.position.x
    y = posicao.pose.pose.position.y
    z = posicao.pose.pose.orientation.z
```

A criação de um *subscriber* para o tópico /scan é feita da seguinte forma:

```
topico_sensor = '/scan'
subscriber_sensor = rospy.Subscriber(topico_sensor, LaserScan, scanCallback)
```

A função *callback* criada para a subscrição do tópico /scan é realizada pelas seguintes linhas:

```
def scanCallback(scan_data):
    global min_value
    min_value = min(scan_data.ranges)
```

As distâncias obtidas por cada raio laser são guardadas numa lista, denominada de *ranges*, e esta função vai buscar o valor mínimo dessa lista de *ranges*.

5.2.1. PASSO A PASSO

O programa **Passo a passo** foi criado com o propósito de demonstrar como um programa desenvolvido num simulador, pode ser reutilizado e aplicado noutro ambiente. Neste caso, são combinadas as funções dos programas do *Turtlesim* **Mover em linha reta** e **Rotação**, para fazer com que o robô se desloque até a um determinado local. A Figura 47 apresenta o percurso que ficou definido para o robô realizar.

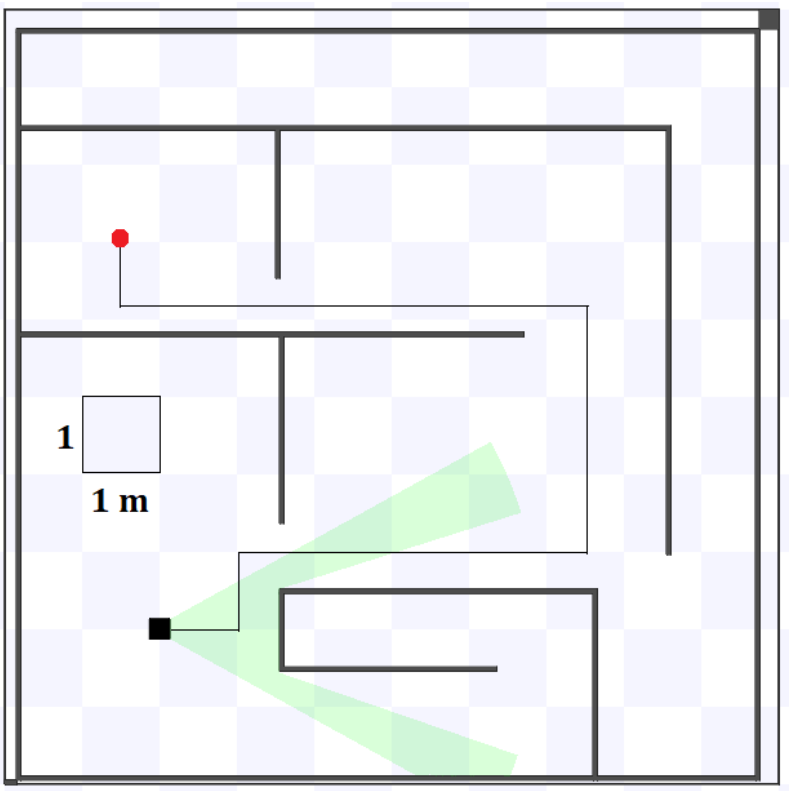


Figura 41 – Percurso do programa **Passo a passo**

Para a concretização do percurso, foi criada a seguinte função:

```
def passo_a_passo(publisher):
    mover(publisher, 1.0, 1.0, True)
    rotacao(publisher, 7.5, 90, False)
    mover(publisher, 1.0, 1.0, True)
    rotacao(publisher, 7.5, 90, True)
    mover(publisher, 1.0, 4.5, True)
    rotacao(publisher, 7.5, 90, False)
    mover(publisher, 1.0, 3.2, True)
    rotacao(publisher, 7.5, 90, False)
```

```
    mover(publisher, 1.0, 6.0, True)
    rotacao(publisher, 7.5, 90, True)
    mover(publisher, 1.0, 0.75, True)
    rotacao(publisher, 7.5, 90, True)
    print ("O robo chegou ao destino!")
```

Esta função vai invocar as outras funções, estando estas com valores atribuídos de acordo com o estudo que foi feito do percurso.

Para além da criação da função anterior, não foram realizadas quais quer alterações nas funções que tinham sido previamente desenvolvidas no *Turtlesim*.

O nó deste programa é iniciado da seguinte forma:

```
rospy.init_node('turtlebot_movimento_passo_a_passo',
anonymous=True)
```

No anexo 4 pode-se visualizar todo o código deste programa.

5.2.2. SENSOR LASER

Este programa tem como objetivo o uso do sensor laser presente no robô simulado, de forma a evitar que este colida com as paredes. O robô vai ser programado para estar em constante movimento linear até se aproximar de uma parede. Ao atingir uma certa distância entre ele e a parede, este vai parar de se mover e vai iniciar um movimento rotacional, até detetar a falta de parede, voltando a iniciar o movimento linear. A distância entre o robô e a parede é detetada pelo laser.

A função desenvolvida para realizar esta tarefa é a seguinte:

```
def mover_laser(publisher):
```

Esta função simplesmente vai receber o *publisher* que permite ao robô movimentar-se.

A mensagem do tipo *Twis*t é declarada da mesma forma como foi efetuada nos programas do simulador *Turtlesim*. De seguida desenvolveu-se a seguinte condição:

```
distancia_minima = min_value
        if distancia_minima > 0.6:
            velocidade_linear = 2
```

```
            velocidade_angular = 0
        else:
            velocidade_linear = 0
            velocidade_angular = -0.5
```

A condição diz que enquanto o valor mínimo dos *ranges* for superior a 0,6 m, o robô vai executar um movimento linear. Caso isso não se verifique, este vai efetuar uma rotação até poder retomar o movimento linear.

Os parâmetros da velocidade linear e angular são realizados da mesma forma como nos programas anteriores, assim como a publicação da mensagem.

É inicializado o nó deste programa através da seguinte linha:

```
rospy.init_node('turtlebot_sensor', anonymous=True)
```

O anexo 5 apresenta o código deste programa na sua totalidade

5.2.3. DESTINO SENSOR

O programa **Destino sensor** vai agrupar o programa **Ponto destino**, desenvolvido no *Turtlesim*, com o programa **Sensor laser**, de forma a criar um programa que possibilite a deslocação do robô até um determinado ponto, sem este colidir com obstáculos.

Através do comando rostopic echo /odom, conseguiu-se descobrir as coordenadas iniciais do robô, sendo elas (0,0). Na Figura 42 pode-se visualizar o resultado do comando e na Figura 42 é possível ver as coordenadas do robô no simulador *Stage*.

```
/home/david/catkin_ws/src:/opt/ros/kinetic/share
david@david-VirtualBox:~$ rostopic echo /odom
header:
  seq: 370
  stamp:
    secs: 37
    nsecs: 100000000
  frame_id: "odom"
child_frame_id: ''
pose:
  pose:
    position:
      x: 0.0
      y: 0.0
      z: 0.0
    orientation:
      x: 0.0
      y: 0.0
      z: 0.0
      w: 1.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
  twist:
    linear:
      x: 0.0
      y: 0.0
      z: 0.0
    angular:
      x: 0.0
      y: 0.0
      z: 0.0
  covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
```

Figura 42 – Comando *echo* no tópico /odom

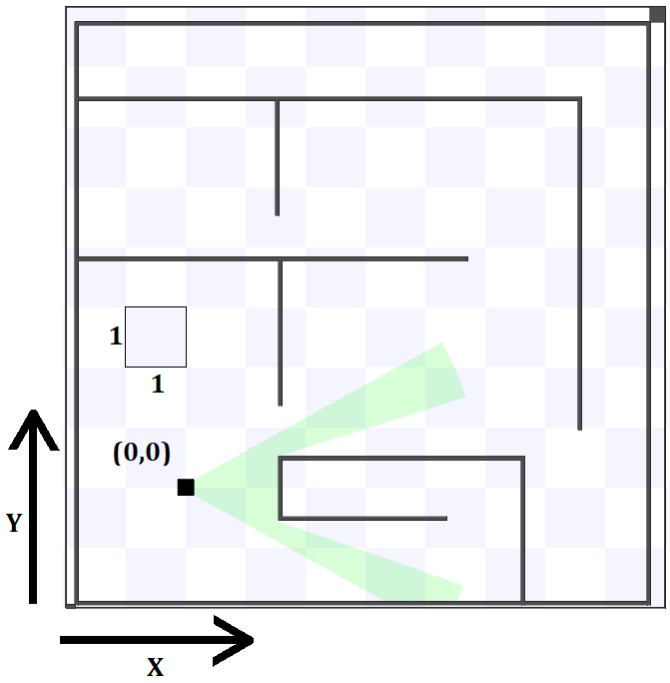


Figura 43 – Coordenadas iniciais do *TurtleBot*

Foi definida a seguinte função:

```
def destino_sensor(publisher, coordenada_x, coordenada_y):
```

Esta função tem o mesmo objetivo que a função desenvolvida no programa **Ponto destino**, possuindo no seu conteúdo a condição que é usada no programa **Sensor laser**. Assim como nos outros programas, a mensagem *Twist* é a mesma. A condição anteriormente descrita, é apresentada da seguinte forma:

```
distancia_minima = min_value
```

```
K_linear = 0.5
distancia    =    abs(math.sqrt(((coordenada_x-x)    **    2)    +
((coordenada_y-y) ** 2)))

K_angular = 4.0
angulo_desejado = math.atan2(coordenada_y-y, coordenada_x-x)

if distancia_minima > 0.5:
velocidade_linear = distancia * K_linear
velocidade_angular = (angulo_desejado-z) * K_angular
else:
velocidade_linear = 0
velocidade_angular = -0.5
```

Desta forma, o robô consegue deslocar-se até determinados pontos, com o auxílio do sensor laser. Nesta situação, optou-se por tornar mais fácil o descolamento do robô, criando pontos intermédios, pelos quais o robô tem de passar antes de alcançar o destino final. Para isso definiu-se uma nova função, estando esta descrita da seguinte forma:

```
def ponto_a_ponto(publisher):
    destino_sensor(publisher_velocidade, 1.0, 1.0)
    destino_sensor(publisher_velocidade, 5.0, 1.0)
    destino_sensor(publisher_velocidade, 6.0, 3.0)
```

A próxima figura apresenta as coordenadas dos pontos intermédios que o robô tem de passar até chegar ao ponto final (6,3).

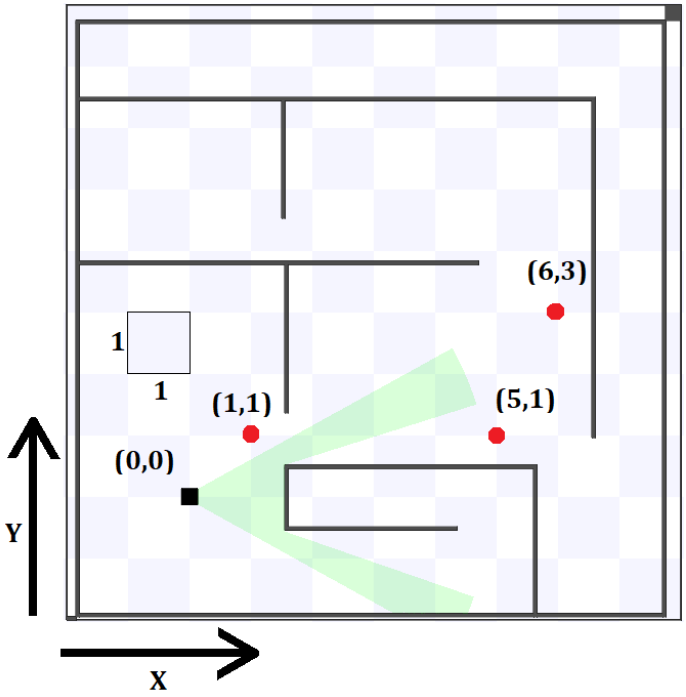


Figura 44 – Coordenadas dos pontos intermédios

A inicialização deste nó é realizada através da seguinte linha de código:

```
rospy.init_node(' 'turtlebot_movimento_com_sensor', anonymous=True)
```

No anexo 6 é demonstrado todo o código desenvolvido para a criação deste programa.

5.3. TESTES E RESULTADOS

Na Figura 45 e na Figura 46 pode-se visualizar o resultado obtido pela execução do programa **Passo a passo**, no simulador *Rviz* e *Stage* respetivamente.

```
passo_a_passo(publisher_velocidade)
```

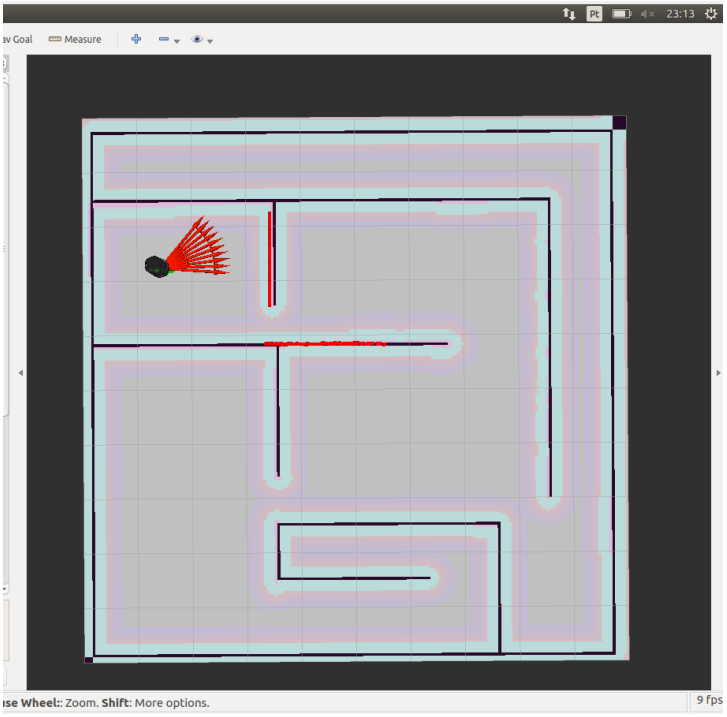


Figura 45 – **Passo a passo** (*Rviz*)

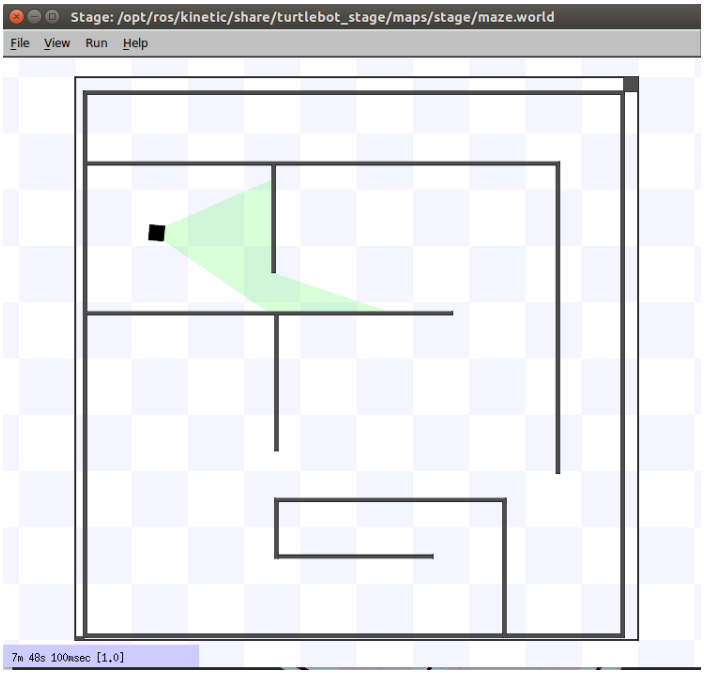


Figura 46 – **Passo a passo** (*Stage*)

A Figura 47 e a Figura 48 apresentam o resultado obtido pela execução do programa **Sensor laser**, em ambos os simuladores.

```
mover_laser(publisher_velocidade)
```

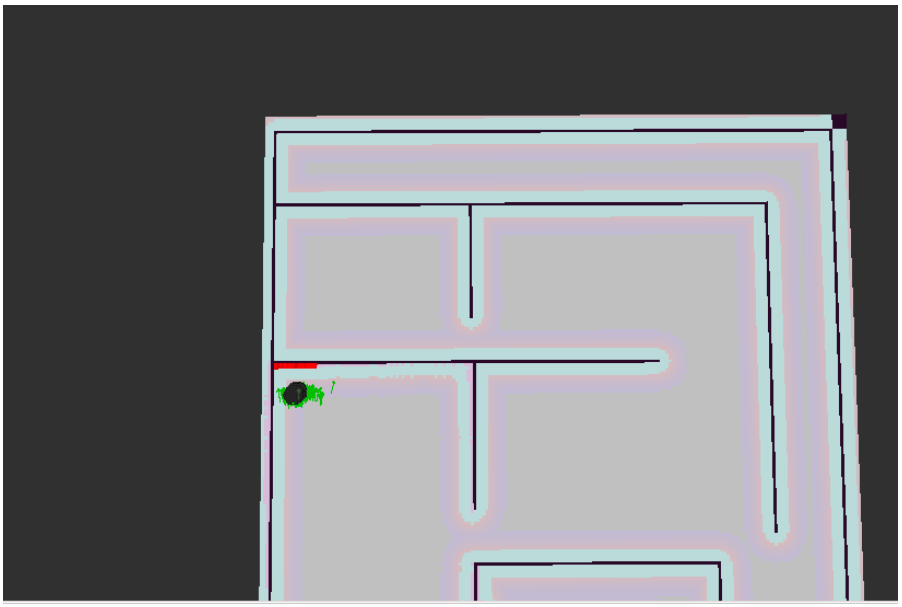


Figura 47 – **Sensor laser** (*Rviz*)

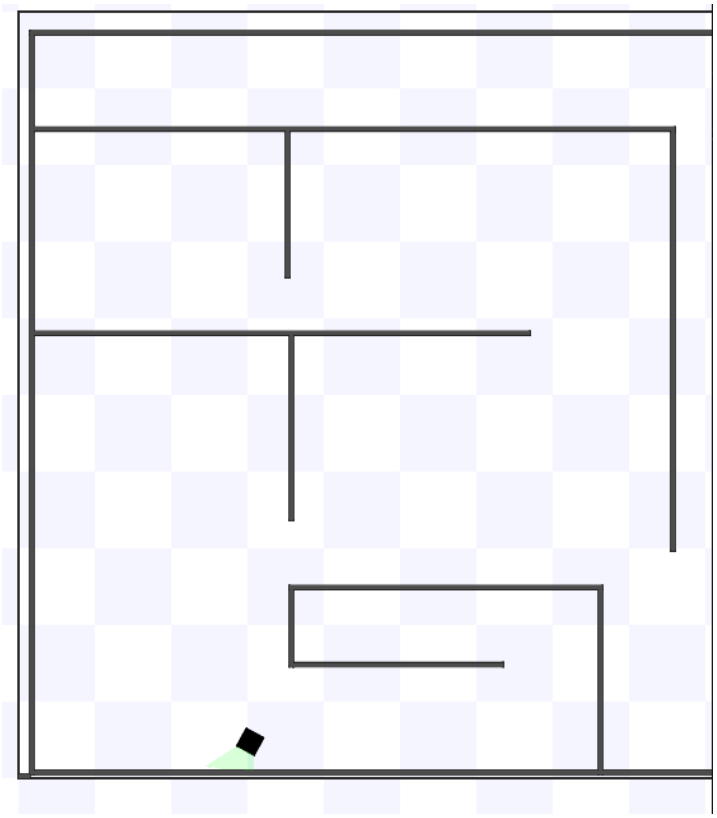


Figura 48 – **Sensor laser** (*Stage*)

As próximas figuras apresentam os resultados que se obtiveram através da execução do programa **Destino sensor**, mostrando o percurso que o robô realizou, sendo a Figura 49 o

primeiro ponto intermédio, a Figura 50 o segundo ponto intermedio e a Figura 51 o ponto final.

```
ponto_a_ponto(publisher_velocidade)
```

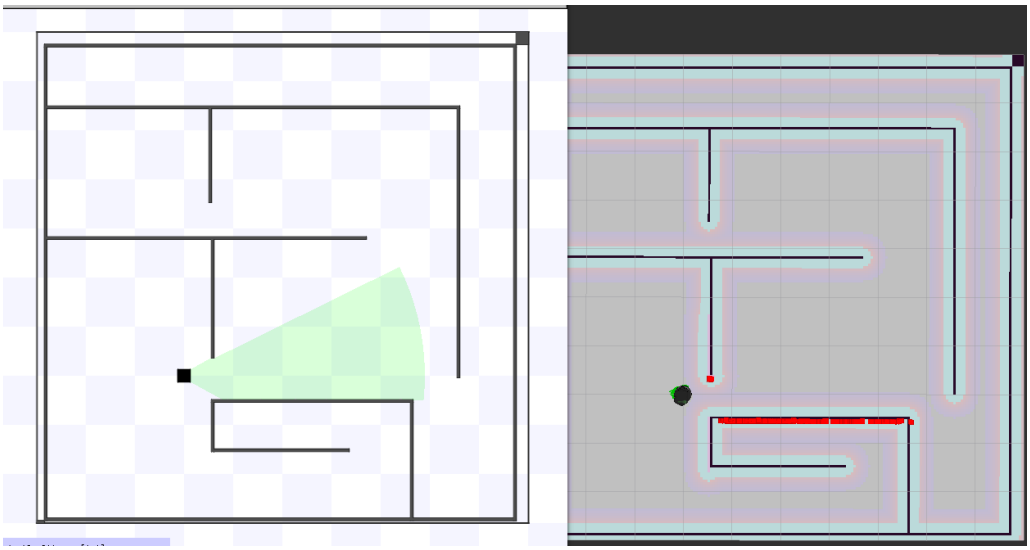


Figura 49 – Primeiro ponto intermédio

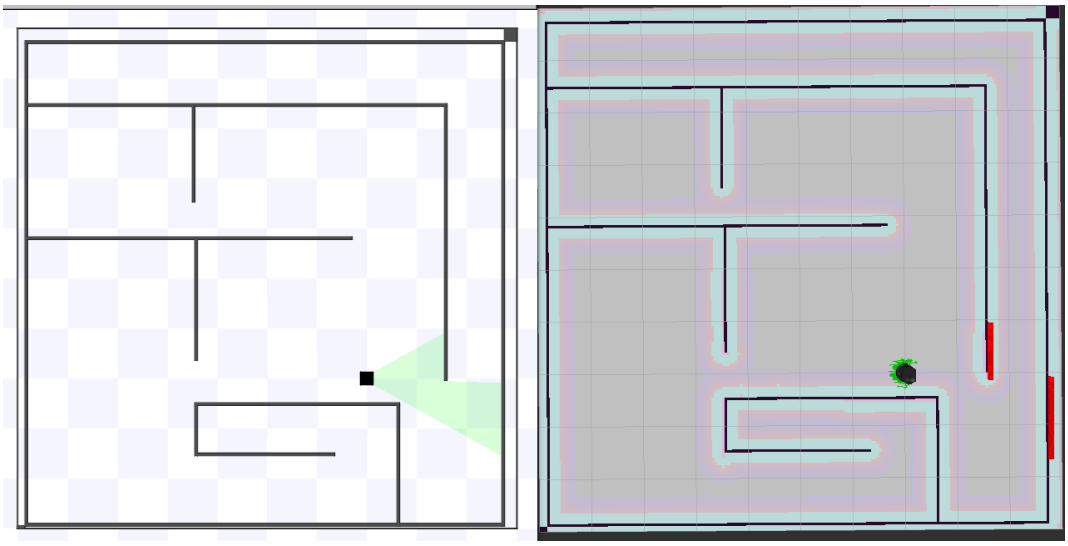


Figura 50 – Segundo ponto intermédio

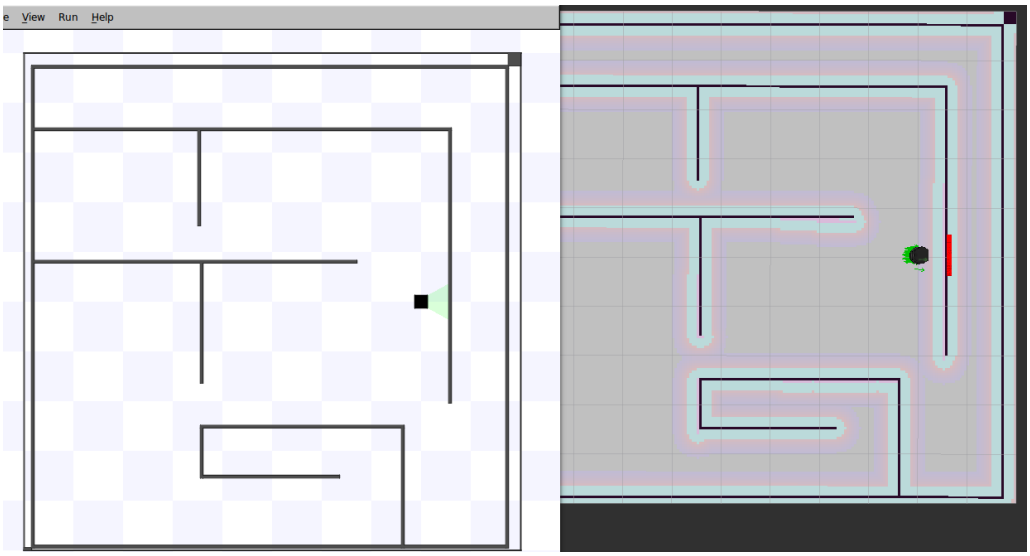


Figura 51 – Ponto final do percurso

Após uma análise dos resultados dos programas, desde a Figura 45 à Figura 51, pode-se concluir que estes for a executados com êxito. Apesar disso, é necessário ter em conta que falta robustez nos programas que envolvem o sensor laser. Estando os pogramas funcionais, pode-se proceder à programação do robô Magni.

6. CONCLUSÕES

Através da realização deste trabalho, ficou-se a conhecer melhor as plataformas robotizadas que existem no mercado. Também possibilitou a aquisição de conhecimentos sobre a ferramenta ROS, o que esta engloba e como é feita a sua programação.

No que toca aos programas desenvolvidos no simulador *Turtlesim*, pode-se dizer que estes foram bem-sucedidos nas suas programações, sendo que o mesmo aconteceu nos programas criados no simulador *TurtleBot*, havendo sempre possibilidade para melhoria.

Em relação aos objetivos propostos no início deste projeto, pode-se concluir que estes não foram totalmente atingidos. Devido à pandemia causada pelo vírus COVID-19 não foi possível ter acesso à plataforma Magni, existindo assim uma componente importante em falta.

Como trabalhos futuros, poder-se-á realizar as devidas programações no robô Magni, assim como realizar novas simulações, recorrendo ao simulador *Gazebo*, e explorar outras funcionalidades do ROS.

Referências Documentais

[1] Ackerman, E. (12 de Março de 2018). *Ubiquity Robotics Launches Beefy ROS Development Platform*. Obtido de IEEE Spectrum: https://spectrum.ieee.org/automaton/robotics/diy/ubiquity-robotics-launches-beefy-ros-development-platform

[2] Robotics, Z. (2018). *Max Mobile Robot Bases*. Obtido de Zagros Robotics: https://www.zagrosrobotics.com/shop/category.aspx?catid=23

[3] Robotics, Z. (2018). *REX Mobile Robot Bases*. Obtido de Zagros Robotics: https://www.zagrosrobotics.com/shop/category.aspx?catid=24

[4] Robotics, Z. (2018). *Item Details - Max '97 Robot Base (12in x 12in)*. Obtido de Zagros Robotics: https://www.zagrosrobotics.com/shop/item.aspx?itemid=521

[5] RobotShop. (2019). *Hangfa Discovery Q2 Robot Platform*. Obtido de RobotShop: https://www.robotshop.com/en/hangfa-discovery-q2-robot-platform.html

[6] *Compass Q1*. (2019). Obtido de Chengdu Hangfa Robotics Co.,Ltd: http://www.hangfa.com/EN/robot/CompassQ1.html

[7] *iClebo Kobuki - about*. (2015). Obtido de iClebo Kobuki: http://kobuki.yujinrobot.com/about2/

[8] *ROS Components - Mobile robots - Turtlebot2*. (2016). Obtido de ROS Components: https://www.roscomponents.com/en/mobile-robots/9-turtlebot-2.html#/3d_sensor-no/controller-no/docking_station-no/power_cods-europa_cee_7_16/additional_battery-no/disk_extra_nuc no/turtlebot_2_assembly_and_configuration-no/arm_robot-no/courses-no/arm_robot

[9] RobotShop. (2019). *4WD Mecanum Wheel Mobile Robotic Platform*. Obtido de RobotShop: https://www.robotshop.com/en/4wd-mecanum-wheel-mobile-robotic-platform.html

[10] *Ubunto - Robotics - What is ROS*. (2020). Obtido de Canonical: https://ubuntu.com/robotics/what-is-ros

[11] Nickels, J. K. (13 de Maio de 2012). *Robot operating systems: Bridging the gap between human and robot.* Obtido de ResearchGate: https://www.researchgate.net/publication/254031552_Robot_operating_systems_Bridging_the_gap_between_human_and_robot

[12] Fernández, A. M. (2013). *Learning ROS for Robotics Programming.* BIRMINGHAM - MUMBAI: Packt Publishing Ltd.

[13] Peter, R. (02 de julho de 2019). *Ros Topics : Ros Tutorial*. Obtido de ROBIN ROBOTICS: https://robinrobotic.blogspot.com/2019/07/ros-topics-ros-tutorial.html

[14] Obtido de https://www.google.com/imgres?imgurl=https%3A%2F%2Fuser-images.githubusercontent.com%2F37708330%2F53758673b1404b803ebe11e993bb2d64de41fc02.png&imgrefurl=http%3A%2F%2Fmasyfood.com%2Flwqj%2Fros2-publisher.html&tbnid=02yB3QW9T14ZUM&vet=12ahUKEwiW67OD_-H

[15] https://www.researchgate.net/publication/327198306_Autoware_on_Board_Enabling_Autonomous_Vehicles_with_Embedded_Systems#pf3

[16] Carol Fairchild, D. T. (novembro de 2017). *ROS Robotics By Example - Second Edition*. Obtido de Packt: https://subscription.packtpub.com/book/hardware_and_creative/9781788479592

[17] *Turtlebot Simulators*. (2016). Obtido de Gaitech's Education Portal: https://edu.gaitech.hk/turtlebot/turtlebot-simulators.html

[18] Woodall, W. (16 de maio de 2018). *rviz*. Obtido de ROS.org: http://wiki.ros.org/rviz

[19] Ubiquity Robotics. *Driving a Robot with a Keyboard*. Obtido de Learn Ubiquity Robots and ROS: https://learn.ubiquityrobotics.com/keyboard_teleop

[20] Ubiquity Robotics. *Fiducial-Based Localization*. Obtido de Learn Ubiquity Robots and ROS: https://learn.ubiquityrobotics.com/fiducials

Anexo 1 – Programa Mover em linha reta

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

from turtlesim.msg import Pose

import math

import time


def poseCallback(posicao):

    global x

    global y, yaw

    x = posicao.x

    y = posicao.y

    yaw = posicao.theta


def mover(publisher, velocidade, distancia, mover_frente):

        #declarar uma mensagem Twist para enviar comandos de velocidade

        mensagem_velocidade = Twist()

        #buscar o posicao inicial
```

```
global x, y
x0 = x #guardar a localizacao inicial da coordenada x
y0 = y #guardar a localizacao inicial da coordenada y

if (mover_frente):
    mensagem_velocidade.linear.x = abs(velocidade)
else:
    mensagem_velocidade.linear.x = -abs(velocidade)

distancia_movida = 0.0
loop_rate = rospy.Rate(10) #a velocidade e publicada a 10Hz (10 vezes por segundo)

while True :
        rospy.loginfo("Turtlesim em movimento")
        publisher.publish(mensagem_velocidade) #comeca o movimento
        loop_rate.sleep()
        distancia_movida = abs(math.sqrt(((x-x0) ** 2) + ((y-y0) ** 2)))
        print distancia_movida
        if  not (distancia_movida < distancia):
            rospy.loginfo("Distanica percorrida")
            break
```

```
        #o robo para quando a distancia e concluida
        mensagem_velocidade.linear.x = 0
        publisher.publish(mensagem_velocidade)

if __name__ == '__main__':
    try:

        #inicializar o no
        rospy.init_node('turtlesim_move_linha_reta', anonymous=True)

        #declarar o publisher da velocidade
        topico_velocidade = '/turtle1/cmd_vel'
        publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)

        #declarar o subscriber da posicao
        topico_posicao = '/turtle1/pose'
        subscriber_posicao = rospy.Subscriber(topico_posicao, Pose, poseCallback)
        time.sleep(2)

        mover(publisher_velocidade, 5.0, 5.0, True)
```

```
except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```

Anexo 2 – Programa Rotação

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

import math

import time


def rotacao(publisher, velocidade_angular_degree, angulo_degree, ponteiros_relogio):

    #declarar uma mensagem Twist para enviar comandos de velocidade

    mensagem_velocidade = Twist()

    #conversao de graus para radianos

    velocidade_angular = math.radians(abs(velocidade_angular_degree))


    if (ponteiros_relogio):

        mensagem_velocidade.angular.z = -abs(velocidade_angular)

    else:

        mensagem_velocidade.angular.z = abs(velocidade_angular)


    loop_rate = rospy.Rate(10) #a velocidade e publicada a 10Hz (10 vezes por segundo)
```

```
    t0 = rospy.Time.now().to_sec() #instantes antes de comecar o movimento

    while True :
        rospy.loginfo("Turtlesim em rotacao")
        publisher.publish(mensagem_velocidade)

        t1 = rospy.Time.now().to_sec()
        angulo_atual_degree = (t1-t0) * velocidade_angular_degree
        loop_rate.sleep()

        if  (angulo_atual_degree >= angulo_degree):
            rospy.loginfo("Rotacao concluida")
            break

    #o robo para quando a rotacao e concluida
    mensagem_velocidade.angular.z =0
    publisher.publish(mensagem_velocidade)

if __name__ == '__main__':
    try:
```

```
    #inicializar o no
    rospy.init_node('turtlesim_rotacao', anonymous=True)

    #declarar o publisher da velocidade
    topico_velocidade = '/turtle1/cmd_vel'
    publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)
    time.sleep(2)

    rotacao(publisher_velocidade, 15, 90, False)

except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```

Anexo 3 – Programa Ponto destino

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

from turtlesim.msg import Pose

import math

import time

from std_srvs.srv import Empty


def poseCallback(posicao):

    global x

    global y, yaw

    x = posicao.x

    y = posicao.y

    yaw = posicao.theta


def ponto_destino(publisher, coordenada_x, coordenada_y):

    global x

    global y, yaw
```

```
loop_rate = rospy.Rate(10) #a velocidade e publicada a 10Hz (10 vezes por segundo)
#declarar uma mensagem Twist para enviar comandos de velocidade
mensagem_velocidade = Twist()

while (True):
    K_linear = 0.5 #a escolha do ganho proporcional tem de ser feito com cuidado
    distancia = abs(math.sqrt(((coordenada_x-x) ** 2) + ((coordenada_y-y) ** 2)))
    velocidade_linear = distancia * K_linear #controlador proporcional

    K_angular = 4.0 #a escolha do ganho proporcional tem de ser feito com cuidado
    #atan2 fornece o angulo entre dois vetores
    angulo_desejado = math.atan2(coordenada_y-y, coordenada_x-x)
    velociade_angular = (angulo_desejado-yaw) * K_angular  #controlador proporcional

    mensagem_velocidade.linear.x = velocidade_linear
    mensagem_velocidade.angular.z = velociade_angular

    publisher.publish(mensagem_velocidade)
    print ('x = ', x, ', y = ', y, ', distancia ate ao destino:', distancia)
```

```
        if (distancia < 0.001):
            break

if __name__ == '__main__':
    try:

        #inicializar o no
        rospy.init_node('turtlesim_ponto_destino', anonymous=True)

        #declarar o publisher da velocidade
        topico_velocidade = '/turtle1/cmd_vel'
        publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)

        #declarar o subscriber da posicao
        topico_posicao = '/turtle1/pose'
        subscriber_posicao = rospy.Subscriber(topico_posicao, Pose, poseCallback)
        time.sleep(2)

        ponto_destino(publisher_velocidade, 9.0, 1.0)

    except rospy.ROSInterruptException:
```

```
rospy.loginfo("node terminated.")
```

Anexo 4 – Programa Passo a passo

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

import math

import time

from nav_msgs.msg import Odometry


def poseCallback(posicao):

    global x

    global y

    global z

    x = posicao.pose.pose.position.x #Posicao-posicao-x

    y = posicao.pose.pose.position.y #Posicao-posicao-y

    z = posicao.pose.pose.orientation.z #Posicao-orentacao-z


def mover(publisher, velocidade, distancia, mover_frente):

        #declarar uma mensagem Twist para enviar comandos de velocidade

        mensagem_velocidade = Twist()
```

```
#buscar o posicao inicial

global x, y

x0 = x #guardar a localizacao inicial da coordenada x

y0 = y #guardar a localizacao inicial da coordenada y


if (mover_frente):

   mensagem_velocidade.linear.x = abs(velocidade)

else:

   mensagem_velocidade.linear.x = -abs(velocidade)


distancia_movida = 0.0

loop_rate = rospy.Rate(10) #a velocidade e publicada a 10Hz (10 vezes por segundo)


while True :

      rospy.loginfo("Turtlesim em movimento")

      publisher.publish(mensagem_velocidade) #comeca o movimento

      loop_rate.sleep()

      distancia_movida = abs(math.sqrt(((x-x0) ** 2) + ((y-y0) ** 2)))

      print  distancia_movida

      if  not (distancia_movida < distancia):
```

```
            rospy.loginfo("Distanica percorrida")
            break

    #o robo para quando a distancia e concluida
    mensagem_velocidade.linear.x = 0
    publisher.publish(mensagem_velocidade)

def rotacao(publisher, velocidade_angular_degree, angulo_degree, ponteiros_relogio):
    #declarar uma mensagem Twist para enviar comandos de velocidade
    mensagem_velocidade = Twist()
    #conversao de graus para radianos
    velocidade_angular = math.radians(abs(velocidade_angular_degree))

    if (ponteiros_relogio):
        mensagem_velocidade.angular.z = -abs(velocidade_angular)
    else:
        mensagem_velocidade.angular.z = abs(velocidade_angular)

    loop_rate = rospy.Rate(10) #a velocidade e publicada a 10Hz (10 vezes por segundo)
    t0 = rospy.Time.now().to_sec() #instantes antes de comecar o movimento
```

```
    while True :
        rospy.loginfo("Turtlesim em rotacao")
        publisher.publish(mensagem_velocidade)

        t1 = rospy.Time.now().to_sec()
        angulo_atual_degree = (t1-t0) * velocidade_angular_degree
        loop_rate.sleep()

        if  (angulo_atual_degree >= angulo_degree):
            rospy.loginfo("Rotacao concluida")
            break

    #o robo para quando a rotacao e concluida
    mensagem_velocidade.angular.z = 0
    publisher.publish(mensagem_velocidade)

def passo_a_passo(publisher):

    mover(publisher, 1.0, 1.0, True)
    rotacao(publisher, 7.5, 90, False)
    mover(publisher, 1.0, 1.0, True)
```

```
    rotacao(publisher, 7.5, 90, True)
    mover(publisher, 1.0, 4.5, True)
    rotacao(publisher, 7.5, 90, False)
    mover(publisher, 1.0, 3.2, True)
    rotacao(publisher, 7.5, 90, False)
    mover(publisher, 1.0, 6.0, True)
    rotacao(publisher, 7.5, 90, True)
    mover(publisher, 1.0, 0.75, True)
    rotacao(publisher, 7.5, 90, True)

    print ("O robo chegou ao destino!")

if __name__ == '__main__':
    try:

        #inicializar o no
        rospy.init_node('turtlebot_movimento_passo_a_passo', anonymous=True)

        #declarar o publisher da velocidade
        topico_velocidade = '/cmd_vel_mux/input/teleop'
        publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)
```

```
    #declarar o subscriber da posicao
    topico_posicao = '/odom'
    subscriber_posicao = rospy.Subscriber(topico_posicao, Odometry, poseCallback)
    time.sleep(2)

    passo_a_passo(publisher_velocidade)

except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```

Anexo 5 – Programa Sensor laser

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

import math

import time

from sensor_msgs.msg import LaserScan


min_value = 0


def scanCallback(scan_data):

    global min_value

    min_value = min(scan_data.ranges) #vai buscar o menor valor da lista dos ranges


def mover_laser(publisher):

    global min_value

    mensagem_velocidade = Twist()

    loop_rate = rospy.Rate(20) #a velocidade e publicada a 20Hz (20 vezes por segundo)
```

```
    while True:
        distancia_minima = min_value

        if distancia_minima > 0.6:
            velocidade_linear = 2
            velocidade_angular = 0
            rospy.loginfo("TurtleBot em movimento linear")
        else:
            velocidade_linear = 0
            velocidade_angular = -0.5
            rospy.loginfo("TurtleBot em rotacao")

        mensagem_velocidade.linear.x = velocidade_linear
        mensagem_velocidade.angular.z = velocidade_angular
        publisher.publish(mensagem_velocidade)
        loop_rate.sleep()

if __name__ == '__main__':
    try:
```

```
    #inicializar o no
    rospy.init_node('turtlebot_sensor', anonymous=True)

    #declarar o publisher da velocidade
    topico_velocidade = '/cmd_vel_mux/input/teleop'
    publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)

    #declarar o subscriber do sensor
    topico_sensor = '/scan'
    subscriber_sensor = rospy.Subscriber(topico_sensor, LaserScan, scanCallback)
    time.sleep(2)

    mover_laser(publisher_velocidade)

except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```

Anexo 6 – Programa Destino sensor

```
#!/usr/bin/env python

import rospy

from geometry_msgs.msg import Twist

import math

import time

from nav_msgs.msg import Odometry

from sensor_msgs.msg import LaserScan


min_value = 0


def poseCallback(posicao):

    global x

    global y

    global z

    x = posicao.pose.pose.position.x #Posicao-posicao-x

    y = posicao.pose.pose.position.y #Posicao-posicao-y

    z = posicao.pose.pose.orientation.z #Posicao-orentacao-z
```

```
def scanCallback(scan_data):


    global min_value

    min_value = min(scan_data.ranges) #vai buscar o menor valor da lista dos ranges




def destino_sensor(publisher, coordenada_x, coordenada_y):

    global x

    global y

    global z

    global min_value


    loop_rate = rospy.Rate(20) #a velocidade e publicada a 20Hz (20 vezes por segundo)

    #declarar uma mensagem Twist para enviar comandos de velocidade

    mensagem_velocidade = Twist()


    while True:

        distancia_minima = min_value

        K_linear = 0.5

        distancia = abs(math.sqrt(((coordenada_x-x) ** 2) + ((coordenada_y-y) ** 2)))
```

```
K_angular = 4.0
angulo_desejado = math.atan2(coordenada_y-y, coordenada_x-x)

if distancia_minima > 0.5:
    velocidade_linear = distancia * K_linear #controlador proporcional
    velocidade_angular = (angulo_desejado-z) * K_angular #controlador proporcional
    rospy.loginfo("TurtleBot em movimento")
else:
    velocidade_linear = 0
    velocidade_angular = -0.5
    rospy.loginfo("TurtleBot em rotacao")

mensagem_velocidade.linear.x = velocidade_linear
mensagem_velocidade.angular.z = velocidade_angular

publisher.publish(mensagem_velocidade)

if (distancia < 0.01):
    break
```

```
def ponto_a_ponto(publisher):

    destino_sensor(publisher_velocidade, 1.0, 1.0)
    destino_sensor(publisher_velocidade, 5.0, 1.0)
    destino_sensor(publisher_velocidade, 6.0, 3.0)
    rospy.loginfo("TurtleBot chegou ao destino!")

if __name__ == '__main__':
    try:

        #inicializar o no
        rospy.init_node('turtlebot_movimento_com_sensor', anonymous=True)

        #declarar o publisher da velocidade
        topico_velocidade = '/cmd_vel_mux/input/teleop'
        publisher_velocidade = rospy.Publisher(topico_velocidade, Twist, queue_size=10)

        #declarar o subscriber da posicao
        topico_posicao = '/odom'
        subscriber_posicao = rospy.Subscriber(topico_posicao, Odometry, poseCallback)
```

```
    #declarar o subscriber do sensor
    topico_sensor = '/scan'
    subscriber_sensor = rospy.Subscriber(topico_sensor, LaserScan, scanCallback)
    time.sleep(2)

    ponto_a_ponto(publisher_velocidade)

except rospy.ROSInterruptException:
    rospy.loginfo("node terminated.")
```