

**FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO**



# **Robô Móvel para Vigilância**

**António Maria Amorim Aguiar de Araújo Correia**

Mestrado Integrado em Engenharia Eletrotécnica e de Computadores

Orientador: Professor Doutor Paulo Gomes da Costa

Co-orientador: Doutor Filipe Neves dos Santos

17 de Julho de 2019



# Resumo

Um robô para vigilância de interiores pode trazer diversas mais-valias. Quer seja em situações em que o acesso a humanos seja adverso, como por exemplo “quintas” de servidores que por vezes chegam a temperaturas demasiado altas para humanos, ou mesmo para cobrir espaços grandes de forma simples e eficiente. Não eliminando por completo a necessidade de vigilantes humanos, um robô com este propósito pode monitorizar diversos aspetos de várias instalações diferentes e reportar essa informação a um utilizador, seja por alguma interface ou vídeo.

Foi nesse âmbito que surgiu o projeto “RobVigil” em 2011, sendo este o efetivo antecessor do robô desenvolvido nesta dissertação. Tendo em mente os objetivos alcançados pelo projeto original, foi desenvolvido um novo robô vigilante baseado inteiramente no framework ROS.

Para tal, foram analisadas várias técnicas de localização e navegação autónomas e explorado o software já desenvolvido de forma a aplicá-lo a um robô real. Foi também reestruturada a organização de hardware de forma a ser possível utilizar novos componentes e facilitar a integração de outros sensores (*e.g.* detetores de fumo, monóxido de carbono ou humidade).

Numa primeira fase, foram implementadas soluções inteiramente disponibilizadas pelo ROS para o desenvolvimento de um robô que, de uma maneira completamente autónoma, consegue dirigir-se para um ponto especificado num mapa conhecido de forma eficiente, robusta e desviando-se de obstáculos não fatais (*i.e.* que impeçam totalmente o deslocamento do robô ou que apareçam de maneira súbita, levando a uma paragem do mesmo por razões de segurança). Seguidamente, foram testadas outras soluções, nomeadamente a *stack Localization Perfect Match* desenvolvida previamente por Sobreira *et al.* para o projeto original.

Já para efeitos de vigilância propriamente dita, foram exploradas opções de transmissão em tempo-real do vídeo capturado pela câmara com o intuito de serem analisadas por um utilizador.



# Abstract

An indoor surveillance robot has many up-sides. Whether it is used in scenarios where the access to humans is adverse, such as server “farms” where temperatures can sometimes reach values too high for humans, or to cover wide areas in a simple and effective way. Without completely eliminating the need for human surveillance, a robot with the same purpose can monitor many aspects in several different sites and report that information, by means of some interface or video, to a user.

In that regard, the project “RobVigil” was created in 2011 becoming the predecessor of the robot developed with this dissertation. Keeping in mind the objectives reached by the original project, a new surveillance robot was developed based entirely in the ROS framework.

As such, many different localization and navigation techniques were analyzed as well as the already developed software in order to apply it to a real robot. The organization of the hardware was also restructured to allow the usage of new components and facilitate the addition of other sensors (*e.g.* smoke detectors, carbon monoxide or humidity).

In a first stage, solution made available entirely by ROS were implemented for the development of a robot that could autonomously drive itself to a point specified in a known map in an efficient and robust manner avoiding non fatal obstacles (*i.e.* obstacles that completely impede the robot’s path or appear so suddenly that the robot stops as a security measure). Afterwards, other solutions were tested, namely the *Localization Perfect Match* stack developed by *Sobreira et al.* which was previously developed for the original project.

In order to implement surveillance features, real time transmission options were explored to stream the video captured by the camera whose end goal would be to be analyzed by a user.



# Agradecimentos

Aos meus orientadores, Prof. Dr. Paulo Costa e Dr. Filipe Neves dos Santos, por todo o apoio prestado, disponibilidade e orientação durante a realização desta dissertação.

Ao Prof. Dr. António Paulo Moreira pela oportunidade, disponibilidade e ajuda prestada em diversas situações.

A toda a equipa do INESC-TEC pela ajuda prestada, boa disposição e disponibilidade.

Aos meus Pais, Irmão e restante família que, de uma maneira ou de outra, me ajudaram e estiveram sempre ao meu lado.

Aos amigos que me acompanharam durante todos estes anos, partilharam bons e maus momentos comigo e se tornaram numa parte fundamental da minha vida.

À Patrícia, por literalmente tudo.

Obrigado.

António Correia



*“If it’s flipping hamburgers at McDonald’s,  
be the best hamburger flipper in the world.  
Whatever it is you do you have to master your craft.”*

Calvin Cordozar Broadus



# Conteúdo

|          |                                               |           |
|----------|-----------------------------------------------|-----------|
| <b>1</b> | <b>Introdução</b>                             | <b>1</b>  |
| 1.1      | Contexto . . . . .                            | 1         |
| 1.2      | Motivação . . . . .                           | 1         |
| 1.3      | Objetivos . . . . .                           | 2         |
| <b>2</b> | <b>Revisão Bibliográfica e Estado da Arte</b> | <b>3</b>  |
| 2.1      | Localização de robôs autónomos . . . . .      | 3         |
| 2.1.1    | Localização relativa . . . . .                | 3         |
| 2.1.2    | Localização absoluta . . . . .                | 4         |
| 2.1.3    | Localização por ajuste de mapas . . . . .     | 5         |
| 2.1.4    | SLAM . . . . .                                | 5         |
| 2.2      | Navegação . . . . .                           | 8         |
| 2.2.1    | Planeamento de Trajetórias . . . . .          | 8         |
| 2.2.2    | Campos de Potencial . . . . .                 | 9         |
| 2.2.3    | Mapas de grelha . . . . .                     | 10        |
| 2.2.4    | Algoritmo A* . . . . .                        | 10        |
| 2.3      | ROS . . . . .                                 | 12        |
| 2.3.1    | Arquitetura ROS . . . . .                     | 12        |
| 2.3.2    | Software existente . . . . .                  | 13        |
| 2.4      | Visão catadióptrica . . . . .                 | 14        |
| 2.4.1    | Montagens . . . . .                           | 14        |
| 2.4.2    | Processamento de imagem . . . . .             | 14        |
| 2.5      | Transmissão de vídeo . . . . .                | 15        |
| 2.5.1    | Técnicas de Compressão . . . . .              | 16        |
| <b>3</b> | <b>Caracterização do Problema</b>             | <b>17</b> |
| 3.1      | Definição do problema . . . . .               | 17        |
| 3.2      | Solução Proposta . . . . .                    | 17        |
| <b>4</b> | <b>Robô Vigilante</b>                         | <b>19</b> |
| 4.1      | Hardware . . . . .                            | 19        |
| 4.2      | Software . . . . .                            | 22        |
| 4.2.1    | Driver . . . . .                              | 22        |
| 4.2.2    | Mapeamento . . . . .                          | 22        |
| 4.2.3    | Câmara e transmissão de vídeo . . . . .       | 25        |
| 4.2.4    | Referenciais . . . . .                        | 26        |
| 4.2.5    | Odometria . . . . .                           | 28        |
| 4.2.6    | Deteção de obstáculos fatais . . . . .        | 30        |

|          |                                                                            |           |
|----------|----------------------------------------------------------------------------|-----------|
| 4.2.7    | Navegação . . . . .                                                        | 31        |
| <b>5</b> | <b>Configuração da <i>stack</i> de Navegação</b>                           | <b>33</b> |
| 5.1      | Mapa de custos . . . . .                                                   | 33        |
| 5.2      | Algoritmos de planeamento de trajetórias . . . . .                         | 33        |
| 5.2.1    | Planeadores locais . . . . .                                               | 34        |
| 5.2.2    | Planeadores globais . . . . .                                              | 35        |
| 5.3      | <i>move_base</i> . . . . .                                                 | 37        |
| 5.4      | Parâmetros . . . . .                                                       | 38        |
| 5.4.1    | <i>sim_time</i> e <i>vx/vtheta samples</i> . . . . .                       | 38        |
| 5.4.2    | <i>robot_footprint</i> e <i>inflation_layer</i> . . . . .                  | 38        |
| <b>6</b> | <b>Análise dos resultados obtidos</b>                                      | <b>41</b> |
| 6.1      | Objetivos . . . . .                                                        | 41        |
| 6.2      | Primeira solução . . . . .                                                 | 42        |
| 6.3      | Segunda solução - Adaptação do <i>Localization_Perfect_Match</i> . . . . . | 45        |
| 6.4      | Terceira solução - Implementação da fusão . . . . .                        | 48        |
| 6.4.1    | Terceira solução - Percurso inverso . . . . .                              | 52        |
| <b>7</b> | <b>Conclusões e Trabalho Futuro</b>                                        | <b>55</b> |
| 7.1      | Trabalho realizado e satisfação dos objetivos . . . . .                    | 55        |
| 7.2      | Trabalho futuro . . . . .                                                  | 56        |
| <b>A</b> | <b>Gráfico dos nós e tópicos de ROS durante o funcionamento do robô</b>    | <b>59</b> |
|          | <b>Referências</b>                                                         | <b>61</b> |

# Lista de Figuras

|      |                                                                                                                       |    |
|------|-----------------------------------------------------------------------------------------------------------------------|----|
| 2.1  | Distribuição de probabilidade de posição $l$ dado que $s$ [1]                                                         | 5  |
| 2.2  | Sobreposição do mapa devido às medidas sem incerteza [1]                                                              | 6  |
| 2.3  | Incerteza de associação de dados[1]                                                                                   | 7  |
| 2.4  | Pseudo-código do algoritmo de localização Monte Carlo (MCL)[1]                                                        | 7  |
| 2.5  | Associação de dados de uma Partícula em <i>FastSLAM</i> [1]                                                           | 8  |
| 2.6  | Ponto de destino a azul e um obstáculo a cor-de-rosa[1]                                                               | 9  |
| 2.7  | células fixas à esquerda e Quadtree à direita [1]                                                                     | 10 |
| 2.8  | Pseudo-código do Algoritmo A*[2]                                                                                      | 11 |
| 2.9  | Comparação de complexidade de diferentes estruturas de dados para o Algoritmo A*[1]                                   | 11 |
| 2.10 | Arquitetura Básica do ROS[3]                                                                                          | 13 |
| 2.11 | Cilindro de projeção da uma imagem omnidirecional[4]                                                                  | 15 |
| 2.12 | Mesmo frame de um vídeo em MPEG-2 (esquerda), MPEG-4 (centro) e h.264/AVC (direita) ao mesmo bitrate de compressão[5] | 16 |
| 3.1  | Estrutura mecânica do <i>RobVigil</i>                                                                                 | 18 |
| 4.1  | Diagrama do hardware                                                                                                  | 20 |
| 4.2  | Diagrama das funções de segurança                                                                                     | 20 |
| 4.3  | Vista frontal do robô                                                                                                 | 20 |
| 4.4  | Vista lateral do robô (aberto)                                                                                        | 20 |
| 4.5  | Vista aproximada do <i>laser_front</i>                                                                                | 21 |
| 4.6  | Vista interior do <i>laser_front</i>                                                                                  | 21 |
| 4.7  | Câmara montada verticalmente                                                                                          | 21 |
| 4.8  | espelho concavo diretamente acima da câmara                                                                           | 21 |
| 4.9  | Mapa 2D obtido pelo <i>hector_slam</i>                                                                                | 23 |
| 4.10 | Mapa 3D obtido pelo <i>octomap</i>                                                                                    | 24 |
| 4.11 | Mapa 3D obtido pelo <i>octomap</i> da sala inicial                                                                    | 24 |
| 4.12 | Mapa 2D obtido pelo <i>octomap</i>                                                                                    | 25 |
| 4.13 | Transmissão do vídeo da câmara a cores                                                                                | 26 |
| 4.14 | Transmissão do vídeo da câmara em <i>grayscale</i>                                                                    | 26 |
| 4.15 | Exemplo de relações existentes entre os referenciais ( <i>tf tree</i> )                                               | 27 |
| 4.16 | Referenciais representados no mapa                                                                                    | 27 |
| 4.17 | Exemplo do Robô localizado                                                                                            | 29 |
| 4.18 | Exemplo do robô mal localizado (perdido)                                                                              | 29 |
| 4.19 | Pose dada por odometria (verde), pelo <i>laser_scan_matcher</i> (azul) e o resultado da fusão (vermelho)              | 30 |
| 4.20 | Zona proibida (vermelho)                                                                                              | 31 |

|      |                                                                                       |    |
|------|---------------------------------------------------------------------------------------|----|
| 4.21 | Configuração necessária da stack de navegação do ROS <sup>1</sup>                     | 31 |
| 5.1  | Mapa de custos local (camada roxa)                                                    | 34 |
| 5.2  | Mapa de custos global (camada azul)                                                   | 34 |
| 5.3  | Exemplo ilustrativo das simulações dadas certas velocidades amostradas <sup>2</sup>   | 35 |
| 5.4  | Exemplo de uma trajetória gerada pelo <i>carrot_planner</i> <sup>3</sup>              | 35 |
| 5.5  | Exemplo de uma trajetória global criada pelo <i>navfn</i> [6]                         | 36 |
| 5.6  | Trajetoória com A*[6]                                                                 | 36 |
| 5.7  | Trajetoória com Dijkstra[6]                                                           | 36 |
| 5.8  | Trajetoória (Dijkstra) sem aproximação quadrática do Potencial[6]                     | 37 |
| 5.9  | Trajetoória (Dijkstra) com Grid Path[6]                                               | 37 |
| 5.10 | Diagrama de comportamentos de recuperação do <i>move_base</i>                         | 37 |
| 5.11 | Trajetoória computada (amarelo) com <i>sim_time</i> pré-definido (esquerda)           | 39 |
| 5.12 | Definição do <i>footprint</i> do robô[7]                                              | 39 |
| 6.1  | Ponto de início (vermelho), objetivo final (Azul) e sub-objetivos (Verde 1-4) no mapa | 41 |
| 6.2  | Ponto de início - primeira solução                                                    | 42 |
| 6.3  | Corredor após primeira passagem estreita (porta)                                      | 43 |
| 6.4  | Chegada ao objetivo 2                                                                 | 43 |
| 6.5  | Entrada na zona intermédia - falha na localização                                     | 44 |
| 6.6  | Zona intermédia - falha na localização mais evidente                                  | 44 |
| 6.7  | Zona intermédia - falha na localização permanente                                     | 44 |
| 6.8  | Paragem de funcionamento da primeira solução                                          | 44 |
| 6.9  | Ponto de início - solução 2                                                           | 45 |
| 6.10 | Após objetivo 1 e antes do objetivo 2                                                 | 45 |
| 6.11 | Objetivo 2 cumprido                                                                   | 45 |
| 6.12 | Entrada na zona intermédia                                                            | 46 |
| 6.13 | Zona intermédia - robô localizado (1)                                                 | 46 |
| 6.14 | Zona intermédia - robô localizado (2)                                                 | 46 |
| 6.15 | Início de entrada no corredor                                                         | 46 |
| 6.16 | Entrada no corredor: objetivo 3 cumprido                                              | 47 |
| 6.17 | Início do corredor                                                                    | 47 |
| 6.18 | Corredor - robô ligeiramente mal localizado                                           | 48 |
| 6.19 | Corredor - robô mal localizado                                                        | 48 |
| 6.20 | Corredor - robô efetivamente perdido                                                  | 48 |
| 6.21 | Paragem de funcionamento da solução 2                                                 | 48 |
| 6.22 | Íncio da terceira solução                                                             | 49 |
| 6.23 | Terceira solução: objetivo 2 cumprido                                                 | 49 |
| 6.24 | Terceira solução: objetivo 3 cumprido                                                 | 49 |
| 6.25 | Localização do robô no corredor                                                       | 50 |
| 6.26 | Recuperação, objetivo 4 cumprido                                                      | 51 |
| 6.27 | Continuação da trajetória após recuperar                                              | 51 |
| 6.28 | Chegada ao objetivo final                                                             | 52 |
| 6.29 | Correção no canto do corredor                                                         | 53 |
| 6.30 | Fim do funcionamento - perspetiva global                                              | 53 |

# Abreviaturas e Símbolos

|            |                                                              |
|------------|--------------------------------------------------------------|
| ROS        | Robot Operating System                                       |
| AGV        | Automated Guided Vehicle                                     |
| IMU        | Inertial Measurement Unit                                    |
| EKF        | Extended Kalman Filter                                       |
| SLAM       | Simultaneous Localization And Mapping                        |
| EKF-SLAM   | Extended Kalman Filter Simultaneous Localization and Mapping |
| $x_r$      | Coordenada do eixo x do robô                                 |
| $y_r$      | Coordenada do eixo y do robô                                 |
| $\theta_r$ | Orientação do robô                                           |
| FOV        | Field of View                                                |
| MPEG       | Moving Pictures Experts Group                                |
| AVC        | Advanced Video Coding                                        |
| GUI        | Graphical User Interface                                     |
| TR         | Trajectory Rollout                                           |
| DWA        | Dynamic Window Approach                                      |
| MCL        | Monte Carlo Localization                                     |
| PC         | Personal Computer                                            |
| CRIIS      | Centre for Robotics in Industry and Intelligent Systems      |
| DEEC       | Departamento de Engenharia Eletrotécnica e de Computadores   |



# Capítulo 1

## Introdução

### 1.1 Contexto

Em 2010/2011 o INESC TEC desenvolveu um Robô para Vigilância de interiores [8] cuja estrutura mecânica se encontra disponível para investigação e desenvolvimento nas suas instalações. Apelidado de “RobVigil”, este era capaz de receber e enviar informação em tempo-real a partir de sensores e câmaras, bem como funcionar autonomamente ou ser operado remotamente. Além disso, mantinha o seu funcionamento independentemente das condições de iluminação e era capaz de cooperar com outros robôs ou humanos. Tinha como função principal a segurança e vigilância remota de locais fechados permitindo monitorização permanente do estado do local (temperatura, humidade, nível de monóxido de carbono, deteção de fogo, etc...) e deteção de intrusos. Desde então, as tecnologias utilizadas evoluíram bastante, tornando-se pertinente voltar a abordar o problema, agora com novas perspetivas e utilizando tecnologias mais recentes, nomeadamente a incorporação de visão catadióptrica e de ROS (Robot Operating System) como framework de base.

### 1.2 Motivação

O projeto original do Robô Vigilância já em 2011 teve bastante sucesso, tendo sido utilizado como base de vários projetos. O software utilizado na altura também se encontra disponível, tendo este já sido optimizado para o funcionamento do robô. Existem então duas perspetivas possíveis. Numa primeira fase, será necessário explorar, com os novos recursos que foram surgindo, como evoluíram as soluções de localização e navegação autónomas, bem como integrá-las no robô. Seguidamente, será preciso estudar o software antigo desenvolvido e explorar a possibilidade de este ser reutilizado ou melhorado.

Nesse âmbito, o uso de ROS torna-se muito pertinente. Sendo este um framework ainda em crescimento mas já num estado de maturidade, um robô que esteja pronto a funcionar com ROS é uma mais valia num contexto de investigação. Este software traz diversas vantagens como por exemplo reutilização de código base, facilidade de incorporação de código open-source, muito

boas ferramentas de simulação, custo computacional baixo, entre outras. Além disso, como uma das ideologias base é a partilha de trabalho e a facilidade da implementação do mesmo, o ROS é uma ferramenta permanentemente em expansão e que beneficia de uma comunidade investida.

Além desta mudança a nível de software, o hardware do robô também será alvo de melhoramentos. Em termos de sensores, o robô beneficia muito da sua estrutura mecânica pois possui um laser rotativo no seu ponto mais elevado. Esta configuração permite uma boa deteção de paredes e/ou marcos, sendo o ideal para se localizar. Ainda em termos gerais, melhorar o hardware implica o uso de componentes mais modernos e uma reestruturação dos mesmos, permitindo uma boa organização interna do robô.

A incorporação de visão catadióptrica, ou omnidirecional, também surge da configuração exposta pois permite utilizar apenas uma câmara para observar todo o seu ambiente circundante. A junção desses dois componentes permite obter uma só imagem em cada instante de tempo que captura, até um certo limite de distância, o panorama total à volta do robô.

### 1.3 Objetivos

Esta dissertação tem os seguintes quatro objetivos principais:

1. *Benchmarking*, adaptação e posterior integração de algoritmos de localização e navegação autónomos baseados em ROS. Pretende-se uma prova de conceito preliminar da integração de ROS, bem como validação do hardware;
2. Re-implementação e adaptação de software antigo para o robô. Isto é, conseguir aproveitar o trabalho desenvolvido para gerações futuras, indo de encontro com a filosofia de trabalho do framework ROS;
3. Integração de visão catadióptrica para captação de vídeo em tempo real;
4. Escolha e implementação de um sistema de transmissão de vídeo para um computador remoto.

## Capítulo 2

# Revisão Bibliográfica e Estado da Arte

### 2.1 Localização de robôs autônomos

Localização é o nome intitulado ao processo de determinar onde está o robô, relativamente a um referencial. Este é um dos problemas principais em robôs autônomos (AGV) pois, para um robô se conseguir movimentar precisa sempre de saber onde se encontra. Sendo este um problema fundamental, ao longo dos anos surgiram inúmeras abordagens para variadas situações podendo estas ser divididas em dois grandes grupos: localização relativa [9] e localização absoluta [10] .

A localização torna-se possível com o uso de sensores. Tipicamente são utilizados *encoders* incrementais e lasers para efetuar as medidas, podendo a escolha de sensores, dependendo da aplicação, conter variados sensores, como IMUs [11] ou potenciômetros.

De entre os dois grupos, serão mais estudados os métodos de localização absoluta, porém, como a maioria destes integram fundamentos da localização relativa, esta será a primeira a ser exposta.

#### 2.1.1 Localização relativa

A localização relativa é o método mais simples e com menos custo computacional sendo por isso o método, na sua base, mais utilizado para localização de um robô. Tal como o nome indica, este método consiste em calcular a posição (localização), de acordo com um referencial global, relativamente à sua posição anterior.

Para tal, o robô inicia o seu funcionamento com uma dada posição inicial, normalmente correspondente à origem do referencial que vai ser utilizado. De seguida, são utilizadas as medidas dos sensores utilizados para calcular a posição atual. A técnica mais utilizada é a odometria [12] . Na sua forma mais simples, esta técnica utiliza apenas *encoders* incrementais para calcular a pose do robô usando as medidas, sob forma de contagem de impulsos e constantes físicas relativas às propriedades mecânicas do robô.

O grande problema desta metodologia provém de erros mecânicos, *i.e.* derrapagem das rodas, superfícies irregulares, obstáculos inesperados, entre outros. Segundo a ideologia relativista do método, um pequeno erro num dado cálculo, propagar-se-á nos cálculos seguintes, podendo gerar

erros muito grandes. Geralmente, em aplicações com pouca tolerância a erros, são utilizadas abordagens que fundem este método com métodos mais complexos juntamente com mecanismos para mitigar os eventuais erros provenientes das medidas.

### 2.1.2 Localização absoluta

A localização absoluta consiste no cálculo da posição do robô, em relação a um referencial, num dado instante de tempo, utilizando apenas as medidas dos sensores. Como não se baseia na posição anterior, comparativamente à localização relativa, o problema de propagação de erros exposto anteriormente é anulado, tornando os erros provenientes das medidas obtidas pelos sensores mais predominantes. Contudo, grande parte das abordagens optam por fundir os dois métodos, pois é muito comum ocorrerem situações que beneficiem de utilizar um método ao invés do outro (*e.g.* mau funcionamento dos sensores).

Existem diversas abordagens para obter a localização absoluta de um robô. A maior distinção entre estes é a necessidade, ou não, de preparação do ambiente, *i.e.* recorrer a marcos cuja localização é conhecida para utilizar como referências no espaço de trabalho do robô [13]. Os métodos que não necessitam de preparação do ambiente consistem geralmente na utilização de visão artificial ou lasers para obter medidas referentes a marcos naturais ou na utilização de mapas. Esta última é a metodologia que será mais alvo de estudo, pois já o projeto original “RobVigil” utilizava este método. Contudo, como todos estes métodos estão sujeitos a erros provenientes dos sensores, opta-se por utilizar a localização relativa e absoluta simultaneamente. Esta abordagem é apelidada de  *fusão* [14]. O critério de decisão entre a utilização de um método ou outro envolve probabilidades. Esta é a base da técnica de localização probabilística que é um ponto fulcral de todos os métodos de localização absoluta.

#### 2.1.2.1 Localização probabilística

As medidas obtidas pelos diversos sensores não são perfeitas devido a erros intrínsecos à movimentação do robô, da sensibilidade dos sensores ou de fatores externos. Pode, então, atribuir-se uma incerteza a cada medida, em cada instante de tempo. Os algoritmos de localização probabilística representam essa incerteza através de distribuições de probabilidade, obtendo assim um grau de confiança quanto aos dados que foram obtidos. Este grau de confiança é o fator decisivo na técnica de  *fusão*. Contudo, existem abordagens baseadas puramente nas distribuições de probabilidade dos sensores que analisam qual é a probabilidade de o robô estar numa determinada posição  $l$  dado que observou uma medida  $s$  (Fig. 2.1). Para que seja possível determinar a sua localização através deste método é necessário que o robô possua três dados: a distribuição de probabilidade da sua posição inicial, o modelo probabilístico completo e um mapa do ambiente circundante do robô.

Existem várias abordagens, nomeadamente o Filtro de Kalman [15] e a localização de Markov [16], cujos conceitos podem ser aplicados para abordagens mais complexas. Existem duas grandes divergências entre os dois métodos: tipo distribuição de probabilidade e peso computacional.

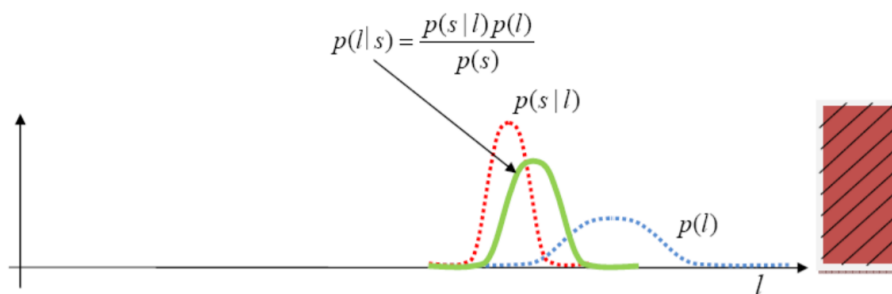


Figura 2.1: Distribuição de probabilidade de posição  $l$  dado que  $s$  [1]

O Filtro de Kalman necessita que o modelo probabilístico do robô seja Gaussiano e de média nula, levando a uma perda de informação dos sensores dado que, na prática, este modelo não coincide com a realidade. Porém, ao simplificar a distribuição de probabilidade este algoritmo tem um custo computacional bastante baixo. Já a Localização de Markov funciona com qualquer tipo de distribuição de probabilidade, não ficando restringido por casos em que a aproximação para uma distribuição Gaussiana de média nula cause demasiadas perdas de informação. No entanto, este algoritmo baseia-se na configuração do espaço de trabalho do robô como sendo células tridimensionais, calculando qual a probabilidade do robô se encontrar em cada uma dessas células, levando a um custo computacional muito elevado.

### 2.1.3 Localização por ajuste de mapas

Este método de localização utiliza um mapa que representa o ambiente onde o robô irá trabalhar. Após obter as medidas dos sensores, o robô tenta validar ou corrigir essas medidas baseando-se no mapa [17]. De uma maneira simples, imaginando que o robô detecta que está a uma certa distância de uma parede, sabendo a posição do robô no mapa, calcula a diferença entre a distância medida e a distância real, podendo assim corrigir a medida, de forma a obter uma localização mais fidedigna. Segundo essa metodologia, existem várias técnicas utilizadas hoje em dia, como o algoritmo *Perfect Match* [18] elaborado por Martin Lauer que demonstra ser uma solução eficiente e robusta.

### 2.1.4 SLAM

Esta abordagem vem unir os tópicos expostos anteriormente. Por um lado, existem técnicas de estimação da pose do robô que necessitam de um mapa do ambiente e por outro, existem técnicas de ajuste de mapas que necessitam da pose do robô. *Simultaneous Localization and Mapping* resolve este problema da seguinte forma: constrói um mapa enquanto estima a pose do robô relativamente a esse mesmo mapa. *SLAM* normalmente é utilizado quando o robô tem um ambiente de trabalho desconhecido assumido como estático. O algoritmo usa como dados de entrada odometria e observações do espaço circundante através de sensores (*e.g.* laser, ultrassom)



Figura 2.2: Sobreposição do mapa devido às medidas sem incerteza [1]

e estima um mapa baseado nessas observações e a posição do robô atual no caso de *Online SLAM* ou a trajetória completa (sendo corrigida com os dados obtidos) no caso de *Offline SLAM*.

Como existe uma incerteza inerente associada às medidas dos sensores, não se pode assumir as medidas como completamente certas, pois o algoritmo iria sobrepor os dados que ia obtendo aos anteriores (Fig. 2.2).

Surgiram então métodos que fundem o algoritmo de *SLAM* com métodos de localização probabilística, nomeadamente o *EKF-SLAM* [19] (*Extended Kalman Filter Simultaneous Localization and Mapping*) e o *FastSLAM* [20] baseados no Filtro de Kalman e no Filtro de partículas (Localização Monte Carlo [21]), respetivamente.

#### 2.1.4.1 EKF-SLAM

O algoritmo *EKF-SLAM* estima a posição do robô, bem como a posição de quaisquer marcos, naturais ou artificiais, detetados até ao momento. Num estado inicial, a matriz de estado tem apenas três componentes ( $x_r$ ,  $y_r$  e  $\theta_r$ ) e apenas adiciona mais componentes referentes a marcos à medida que são observados. Surge, então, um dos grandes problemas deste algoritmo, a associação de dados, *i.e.* quando é observado um marco, se este já foi previamente observado ou se é novo, tendo então de ser adicionado à matriz de estado (Fig. 2.3). Além disso, sendo necessário que os modelos probabilísticos do robô sejam Gaussianos de média nula, acresce a incerteza associada às medidas. Esse acréscimo dificulta o processo de linearização do Filtro de Kalman Extendido.

Existe, porém, um conceito que ajuda a mitigar estes problemas chamado de *Loop closure*. Este baseia-se na premissa do encerramento de um ciclo, *i.e.*, o robô já ter detectado todos os marcos existentes e estar de novo a observar marcos agora já presentes na matriz de estado, podendo verificar e/ou corrigir as medidas obtidas. Usufruindo deste conceito, as covariâncias associadas à incerteza da deteção de marcos podem ser muito reduzidas.

Atualmente, este método já não é ótimo, mas ainda consegue ter resultados aceitáveis em certas aplicações.

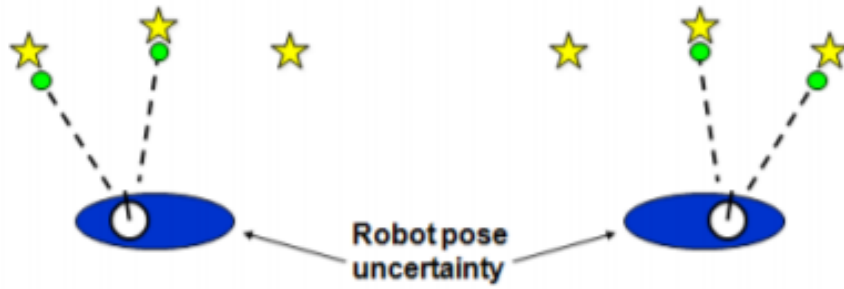


Figura 2.3: Incerteza de associação de dados[1]

### 2.1.4.2 FastSLAM

Esta abordagem ao *SLAM* é baseada no algoritmo do Filtro de Partículas. Este é um algoritmo recursivo baseado na regra de Bayes que representa a função densidade de probabilidade como um conjunto de partículas, com um determinado peso associado, ao longo de um mapa e de acordo com o modelo de movimentação do robô. Cada partícula representa a probabilidade do robô se encontrar numa determinada posição. Desta forma, uma maior concentração de partículas indica uma maior probabilidade de uma dada localização do robô ser a verdadeira [22]. Este algoritmo apresenta vários pontos fortes, como a possibilidade de ser utilizado com qualquer tipo de modelo probabilístico, baixo custo computacional, quando comparado com localização de Markov, e relativa facilidade de implementação. Um algoritmo básico em pseudo-código é apresentado na Fig. 2.4. Ao juntar um determinado número de partículas aleatórias ao conjunto, este método consegue recuperar em casos de *kidnapping* (i.e. movimentação forçada e inesperada do robô por intermédio humano).

```

1:  Algorithm MCL( $\mathcal{X}_{t-1}, u_t, z_t, m$ ):
2:     $\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$ 
3:    for  $m = 1$  to  $M$  do
4:       $x_t^{[m]} = \text{sample\_motion\_model}(u_t, x_{t-1}^{[m]})$ 
5:       $w_t^{[m]} = \text{measurement\_model}(z_t, x_t^{[m]}, m)$ 
6:       $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$ 
7:    endfor
8:    for  $m = 1$  to  $M$  do
9:      draw  $i$  with probability  $\propto w_t^{[i]}$ 
10:     add  $x_t^{[i]}$  to  $\mathcal{X}_t$ 
11:    endfor
12:    return  $\mathcal{X}_t$ 

```

Figura 2.4: Pseudo-código do algoritmo de localização Monte Carlo (MCL)[1]

Deste modo, a utilização de um Filtro de Partículas é muito pertinente para o algoritmo *SLAM*, pois consegue resolver muitos dos seus problemas, nomeadamente os que foram apresentados para o *EKF-SLAM*. Porém, existe um problema relativamente ao tempo de processamento, pois

utilizando o Filtro de Partículas para estimar a pose do robô juntamente com o mapa, requer bastante poder computacional. De forma a resolver este problema, é utilizada a probabilidade posterior fatorizada de acordo com o teorema de Rao-Blackwell [23]. Deste modo, tratando apenas do problema de localização com o Filtro de Partículas e tratando da localização dos marcos (partículas) com um determinado número de Filtros de Kalman Extendidos (EKF), o custo computacional torna-se muito mais leve, dado que, tendo a posição do robô, a construção do mapa é simples permitindo uma computação eficiente das diferentes partículas. Esta abordagem permite ainda resolver o problema de associação de dados, uma vez que a associação passa a ser feita por partícula (Fig. 2.5).

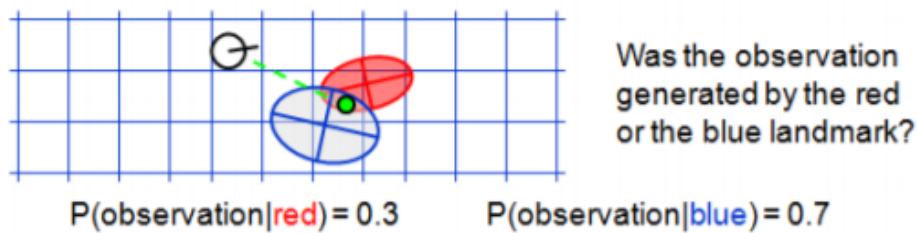


Figura 2.5: Associação de dados de uma Partícula em *FastSLAM*[1]

Os métodos baseados em Filtros tem dominado a literatura de *SLAM* por serem métodos que permitem uma grande autonomia ao robô, apresentam resultados satisfatórios e podem ser implementados sem grande custo computacional, em determinados casos.

## 2.2 Navegação

A Navegação de um robô é o processo de decisão que envolve todo o seu movimento. Existem três conceitos que englobam a navegação:

- **Planeamento do Caminho** — descreve o caminho desde o ponto inicial até ao final, evitando obstáculos, em termos matemáticos ou geométricos;
- **Planeamento de Trajetórias** — representa o caminho em função do tempo;
- **Planeamento de Movimentos** — tem em conta as restrições cinemáticas e dinâmicas do robô.

Geralmente, utiliza-se o termo Planeamento de Trajetórias (*path planning*) no conjunto destas três situações.

### 2.2.1 Planeamento de Trajetórias

Um problema de Planeamento de Trajetórias tem como objetivo principal encontrar um caminho entre dois pontos evitando obstáculos. Face a isto, existem vários critérios adicionais que

podem ser otimizados, como o tempo que leva a completar a trajetória, a distância total, o gasto de energia, entre outras variáveis. Existem várias abordagens descritas na literatura, podendo estas ser divididas em dois grandes grupos: *Offline* e *Online path planning*. O primeiro diz respeito a ambientes em que o robô já tenha acesso à localização de obstáculos estáticos, bem como à trajetória de obstáculos móveis. Esta abordagem também é apelidada de *global path planning*. Quando esta informação não está disponível ou está incompleta, o robô obtém-na através de sensores enquanto do seu movimento. Esta abordagem diz respeito ao segundo grupo, *online path planning*. Este segundo grupo pode ser considerado uma extensão do primeiro, pois, em vários casos, conhecendo alguma informação é possível gerar uma trajetória preliminar que será sujeita a alterações segundo os obstáculos detetados pelo robô.

Ao longo dos últimos anos, surgiram diversos algoritmos através de diferentes técnicas de representação do ambiente. Serão abordados dois deles: Campos de Potencial [24] e o Algoritmo A\* [25], baseado em *grid maps* (Mapas de grelha).

### 2.2.2 Campos de Potencial

No algoritmo de Campos de Potencial, o ambiente é representado da seguinte forma: o ponto de destino cria um potencial de atração e os obstáculos criam um potencial de repulsão (Fig. 2.6). O robô comporta-se como uma partícula dentro do campo de potencial resultante. Este algoritmo é usado tipicamente apenas em ambientes estáticos. Um dos grandes entraves à utilização deste algoritmo é a criação de mínimos locais (por exemplo, se um obstáculo e o ponto de destino estiverem alinhados), levando a que não seja ótimo.

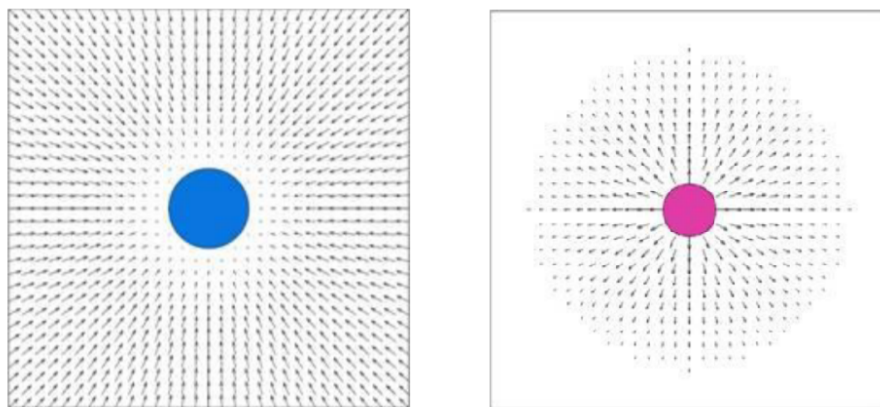


Figura 2.6: Ponto de destino a azul e um obstáculo a cor-de-rosa[1]

Contudo, existem técnicas que tentam resolver, ou mitigar, o problema dos mínimos locais dado que, quando este não existe, o algoritmo apresenta bons resultados e tem um custo computacional muito mais leve que outras soluções. Uma dessas técnicas envolve o algoritmo ser usado em ambientes não estáticos, na esperança de que, com o movimento dos obstáculos, a situação de

mínimo local se dissipe. Outra solução admite que, aquando de uma situação de mínimo local, o robô circunda o obstáculo até que o potencial criado permita chegar ao ponto de destino. Existe um *trade-off* entre a utilização destas técnicas e a eficiência do algoritmo, pelo que são apenas utilizadas em casos muito específicos [26].

### 2.2.3 Mapas de grelha

A decomposição de mapas em Mapas de grelha apresenta diversas utilidades para o planeamento de trajetórias. Esta técnica consiste em dividir o mapa em células, todas estas formando o *Cspace* (ou *espaço de configuração*) sendo que cada uma destas pode estar “ocupada” caso faça parte de um obstáculo, “parcialmente ocupada” (dependendo da resolução das células) ou “livre”. Cabe então ao algoritmo selecionado encontrar uma trajetória para o robô chegar de um ponto de origem ao ponto de destino dentro do conjunto de células livres apelidado de *Clivre* (ou *free space*). Existem várias maneiras de representar as células geometricamente, sendo que o quadrado é o método mais utilizado pela sua simplicidade. Dentro destes, as mais usadas para espaços 2D são a decomposição em células fixas e em Quadtree (Fig. 2.7). A decomposição em Quadtree começa por dividir o *Cspace* em 4 células iguais e, recursivamente, se uma célula não fizer parte do *Clivre*, dividir novamente em 4 células iguais até todas fazerem parte do mesmo.

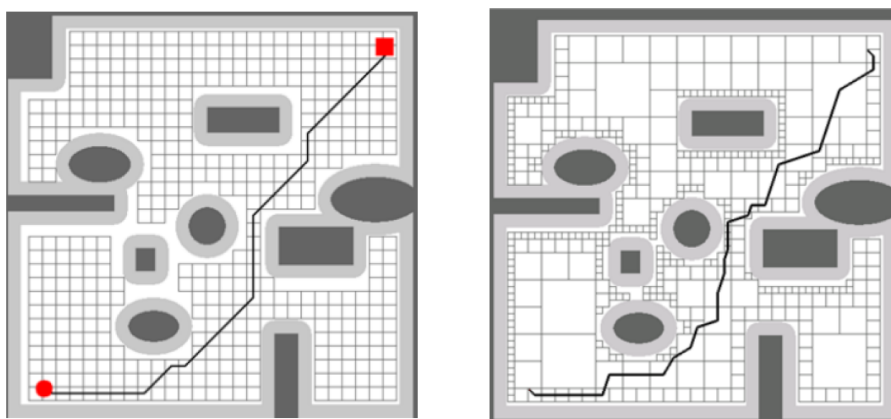


Figura 2.7: células fixas à esquerda e Quadtree à direita [1]

### 2.2.4 Algoritmo A\*

O Algoritmo A\* é globalmente aceite como sendo o mais eficiente para a maioria das aplicações. Isto deve-se ao facto de ser completo e ótimo quando tem acesso a um mapa de grelha do espaço de trabalho. Este algoritmo enquadra-se no grupo de algoritmos de pesquisa em grafos, mais precisamente com heurística. A heurística é a principal distinção entre métodos, podendo a heurística do algoritmo A\* ser considerada uma extensão da heurística do algoritmo de *Dijkstra*. Isto é, a heurística do A\* entra em conta com a informação dos pontos que estão mais perto do

ponto inicial, bem como o quanto falta para alcançar o ponto de destino. É apresentado na Fig. 2.8 uma versão básica do algoritmo.

Mesmo sendo considerado o algoritmo mais eficiente, pode não ser possível ser utilizado em algumas aplicações devido ao elevado custo computacional que caracteriza o algoritmo. Porém, a escolha da heurística, bem como outras alterações a nível de software ( estrutura de dados, alterações no *Cpspace* ou escolha do tamanho das células), altera a complexidade do algoritmo de forma a que seja mais adequado ao hardware do robô, tendo em conta certas diferenças ao nível da performance [27] .

1. Adicionar o nó de partida a  $O$
2. Repetir
  - 2.1. Escolher o  $n_{\text{melhor}}$  (nó melhor) de  $O$  tal que  $f(n_{\text{melhor}}) \leq f(n)$ ,  $\forall n \in O$
  - 2.2. Remover o  $n_{\text{melhor}}$  de  $O$  e adicionar a  $C$
  - 2.3. Se  $n_{\text{melhor}}$  = nó final , terminar o algoritmo
  - 2.4. Para todos  $n \in \text{Star}(n_{\text{melhor}})$  fazer o seguinte:
    - 2.4.1. Se  $(n \notin O)$  e  $(n \notin C)$  então adiciona o nó  $n$  a  $O$
    - 2.4.2. Se  $(n \in O)$  então se  $g(n_{\text{melhor}}) + c(n_{\text{melhor}}, n) < g(n)$  então alterar o pai do nó  $x$  para  $n_{\text{melhor}}$
3. Até que  $O$  esteja vazio

Figura 2.8: Pseudo-código do Algoritmo A\*[2]

Dentro das alterações possíveis, com o objetivo de diminuir o tempo computacional, a escolha da heurística e das estruturas de dados são predominantes. Pode observar-se na Fig. 2.9 a complexidade associada a diferentes tipos de estruturas de dados, sendo notável que com o uso de *Binary Heaps* seja possível diminuir o tempo de processamento até 50% em comparação com *arrays* ordenados.

|                 | <i>Array<br/>Ordenados</i> | <i>Listas<br/>ligadas<br/>ordenadas</i> | <i>Tabelas<br/>Hash</i> | <i>Binary<br/>heaps</i> |
|-----------------|----------------------------|-----------------------------------------|-------------------------|-------------------------|
| <b>Procura</b>  | $\Theta(1)$                | $\Theta(n)$                             | $\Theta(n)$             | $\Theta(1)$             |
| <b>Inserir</b>  | $\Theta(n)$                | $\Theta(1)$                             | $\Theta(1)$             | $\Theta(\log n)$        |
| <b>Eliminar</b> | $\Theta(n)$                | $\Theta(1)$                             | $\Theta(1)$             | $\Theta(\log n)$        |

Figura 2.9: Comparação de complexidade de diferentes estruturas de dados para o Algoritmo A\*[1]

Com o intuito de manter um bom nível de performance do algoritmo mas ainda assim diminuir a carga computacional, são utilizadas alterações ao *Cspace*. Abordagens como o cálculo de pontos

de choque, zonas de folga ou criação de rasto servem para definir a maneira como o robô se comporta face a obstáculos estáticos ou móveis [27].

Existindo esta flexibilidade quanto aos parâmetros do A\*, as inúmeras variações existentes são referidas como parte da “família” de algoritmos A\*, como por exemplo os algoritmos IDA\*, ARA\* ou AD\* [28] são ainda hoje alvo de desenvolvimento e investigação de forma a conseguir a melhor performance para todos os diferentes casos possíveis.

## 2.3 ROS

O *Robot Operating System* é um framework de robótica open-source desenvolvido pela Open Source Robotics Foundation. Surgiu com o objetivo de fornecer serviços e ferramentas para o desenvolvimento de projetos no âmbito da robótica, como bibliotecas e simuladores, bem como uma arquitetura modular unificada que promove a investigação e a partilha de trabalho [29]. O conceito que diferencia o ROS de outros frameworks é esta arquitetura pois, estando um robô habilitado para conseguir trabalhar segundo a mesma, torna-se possível a reutilização de código desenvolvido e disponibilizado por outros utilizadores. Deste modo, o ROS beneficia de uma comunidade ativa e empenhada que cresce continuamente.

### 2.3.1 Arquitetura ROS

Até ao momento, o desenvolvimento em ROS é feito unicamente em plataformas baseadas em Unix, em que os ficheiros individuais são escritos em C++ ou Python. Segundo [1], na arquitetura de ROS existem os seguintes conceitos:

- **Nós** — programas executáveis que desempenham cálculos, processamento ou Funções específicas;
- **Mensagens** — mecanismo de comunicação entre nós por estruturas de dados;
- **Tópicos** — Nome utilizado para identificar uma mensagem. Podem ser publicados e/ou subscritos;
- **Serviços** — Utilizados para interações *request/reply*;
- **Bags** — Formato para guardar registos de dados em ficheiros reproduzíveis;
- **Master** — Regista os nomes, publicação, subscrição dos nós. Inclui também um servidor que permite guardar parâmetros centralmente.

Pode observar-se na Fig. 2.10 uma arquitetura simples: um nó publica informação sob forma de uma mensagem a um tópico e outro nó, que se encontra subscrito ao tópico, recebe essa informação também sob forma da mesma mensagem. Quando um serviço é invocado, este pode executar funcionalidades de um outro nó.

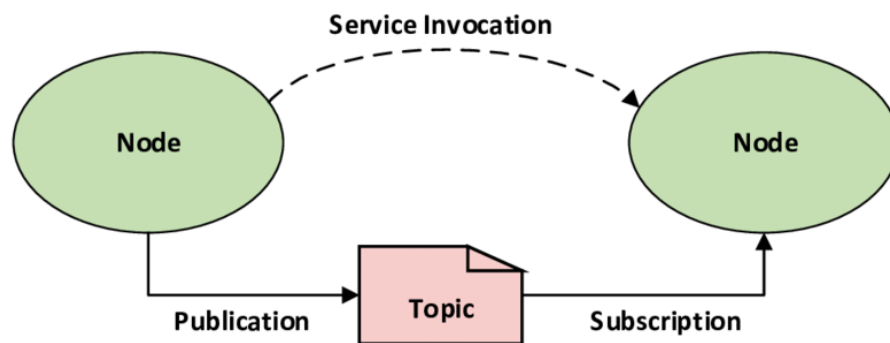


Figura 2.10: Arquitetura Básica do ROS[3]

### 2.3.2 Software existente

Para a implementação de um sistema de localização e navegação no robô, existem neste momento quatro *packages* em ROS pertinentes ao projeto: *robot\_localization*, *navigation*, *hector\_slam* e *gmapping*.

O primeiro, *robot\_localization* [30], é um conjunto de nós que estimam a pose do robô no espaço tridimensional baseados no *EKF*. Além disso, foi desenvolvido de forma a não ter restrições quanto ao número de *inputs*, *i.e.* sensores, podendo até integrar dados obtidos a partir de GPS. A estimativa é obtida através da fusão entre os diversos *inputs*.

O *package navigation*[31] foi desenvolvido com o intuito de ser utilizado para qualquer tipo de aplicações, sendo possível interpretar o “mundo” visto pelo robô em 2D ou 3D conforme o necessário. São recebidas informações da odometria e dos sensores do robô de forma a calcular comandos de velocidade seguros para os motores. Foi também desenvolvido um algoritmo de espaço desconhecido que permite ao robô reagir inteligentemente quando precisa de obter mais informações sobre um determinado espaço, *e.g.* quando se movimenta perto de cantos. Uma grande vantagem desta *stack* é o facto de suportar *SLAM* para casos em que não existe conhecimento prévio do ambiente.

O *package hector\_slam* [32], desenvolvido por Stefan Kohlbreche e Johannes Meyer, permite a um robô gerar um mapa, bem como localizar-se dentro dele utilizando apenas dados provenientes de um LIDAR. A localização obtida por este *package* é bastante precisa e não necessita de odometria proveniente das rodas do robô.

Por fim, o *package gmapping*[33] permite a utilização de *SLAM* baseado no Filtro de Partículas, construindo um *gridmap* a partir de dados obtidos através de lasers. Foi desenvolvida uma abordagem que tem em consideração o movimento do robô, bem como as observações mais recentes o que leva a uma redução da incerteza da pose do robô na implementação do Filtro de Partículas.

## 2.4 Visão catadióptrica

Intitula-se de visão catadióptrica um sistema de visão que engloba tanto a reflexão como a refração de luz. Assim sendo, este tipo de sistemas é caracterizado pela obtenção de visão omnidirecional, *i.e.* um campo de visão de  $360^\circ$ , tornado possível pela inclusão de um ou mais espelhos na configuração da câmara presente no robô. Além disso, são alternativas mais económicas, pois requerem apenas uma câmara, que permitem ao robô observar todo o seu espaço circundante.

### 2.4.1 Montagens

Segundo [34], existem diversas montagens possíveis no que toca a sistemas de visão em robótica:

- **Câmara fixa a apontar para a frente** — Configuração típica e a mais simples. Tem a grande desvantagem de obter informação limitada pelo campo de visão (*FOV*) e pela possibilidade da imagem ser obstruída por obstáculos;
- **Câmara motorizada** — A estrutura da câmara é acoplada a um motor. O *FOV* aumenta com o movimento do motor (cima/baixo - *tilt* e esquerda/direita - *pan*). Um problema provém de encontrar formas eficientes de coordenar os movimentos do robô e da câmara;
- **Múltiplas câmaras** — O uso de múltiplas câmaras é a solução mais simples para obter um *FOV* completo e não estar sujeito aos problemas expostos anteriormente. Porém, é uma solução muito mais dispendiosa e que também traz problemas de consumo de energia associados ao uso das várias câmaras;
- **Câmara fixa apontada para um ou mais espelhos** — Esta é a montagem considerada para um sistema de visão catadióptrica. A configuração mais utilizada e, conseqüentemente, a que será utilizada no robô Vigilante, consiste numa câmara apontada verticalmente para cima em direção a um espelho convexo. A desvantagem deste método é a distorção associada às observações da câmara, porém, pode controlar-se estes efeitos processando as imagens obtidas.

### 2.4.2 Processamento de imagem

Estando já pré-determinada a configuração da câmara e do espelho que serão utilizados, torna-se pertinente explorar a possibilidade de transformar as imagens obtidas em imagens panorâmicas para facilitar a sua interpretação.

De forma a obter uma imagem panorâmica partindo de uma imagem omnidirecional, utiliza-se a abordagem intitulada de *mapas geométricos*, *i.e.* mapear os pixels da imagem obtida na perspetiva original para a imagem na perspetiva que se procura. Assim sendo, segundo [35], partindo da equação do espelho convexo (parabólico), é possível interpolar as intersecções dos diferentes raios de visão, chegando às equações:

$$x_i = \frac{h}{z_p + \sqrt{x_p^2 + y_p^2 + z_p^2}} \times x_p \quad (2.1)$$

$$y_i = \frac{h}{z_p + \sqrt{x_p^2 + y_p^2 + z_p^2}} \times y_p \quad (2.2)$$

As equações 2.1 e 2.2 relacionam, de forma generalizada, um pixel  $P(x_p, y_p, z_p)$  com o seu ponto de intersecção dado por  $x_i$  e  $y_i$ , em que  $h$  é o parâmetro da parábola. Para o caso específico da transformação em imagem panorâmica, estas equações podem ser reescritas da seguinte forma:

$$x_i = \frac{h}{z_p + \sqrt{w^2 + f^2}} \times x_p \quad (2.3)$$

$$y_i = \frac{h}{z_p + \sqrt{w^2 + f^2}} \times y_p \quad (2.4)$$

em que  $w$  é a altura do pixel no cilindro de projeção e  $f$  é o comprimento focal (raio do cilindro)[35]. Uma representação do cilindro de projeção pode ser vista na Fig. 2.11. As equações 2.3 e 2.4 representam uma otimização das equações 2.1 e 2.2 dado que o termo  $\sqrt{x_p^2 + y_p^2 + z_p^2}$ , equivalente à distância entre o pixel e o foco da parábola, é agora constante, o que leva a uma computação muito mais rápida, sendo possível utilizar esta transformação para vídeos em tempo real [35].

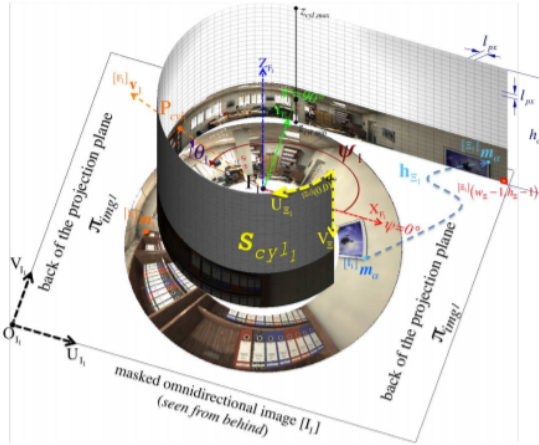


Figura 2.11: Cilindro de projeção da uma imagem omnidirecional[4]

## 2.5 Transmissão de vídeo

Relativamente a qualquer tipo de transmissão de dados, existem dois fatores predominantes que influenciam o produto final: *delay* e qualidade de transmissão. Neste caso específico, os dados a transmitir são um vídeo contínuo capturado pelo robô. Assim sendo, o *delay* de transmissão apresenta-se como latência, *i.e.* o *delay* entre o momento em que o vídeo é capturado e efetivamente recebido no destino. Já a qualidade de transmissão apresenta uma relação direta com a resolução do vídeo. Desta forma, uma transmissão ideal minimiza o *delay* e maximiza a qualidade.

Na prática, existe sempre um *tradeoff* entre estes dois fatores pois transmitir com mais qualidade implica um *delay* maior. Isto deve-se ao facto de vários fatores influenciarem os efeitos

expostos, como o hardware, protocolos de transmissão e o algoritmo de compressão de dados utilizado.

### 2.5.1 Técnicas de Compressão

Em relação aos algoritmos de compressão para vídeo, desde 1990, as famílias de algoritmos MPEG-x e h.26x dominam a literatura. Contudo, a última adição aos standards internacionais de codificação de vídeo, h.264/AVC é aceite como sendo o algoritmo mais potente até ao momento [36]. Este algoritmo providencia o melhor balanço entre eficiência de compressão, complexidade de implementação e custo. Como a captura de vídeo do robô tem como objetivo ser supervisionada por um utilizador para fins de vigilância, a latência é um fator de grande importância.

É, então, necessário um algoritmo que apresente um bom nível de compressão dado que este tem uma relação direta com a latência (quanto mais comprimida estiver a informação, menos tempo demora a ser transmitida), sem nunca sacrificar demasiado a qualidade do vídeo pois este tem de ser interpretável para um utilizador humano. Com esses objetivos em mente, o uso de h.264/AVC torna-se muito pertinente dado que é o algoritmo que apresenta melhor eficiência de compressão mantendo uma boa qualidade. Comparativamente a MPEG-2 e MPEG-4 Visual, o algoritmo h.264/AVC é capaz de melhorar a qualidade da imagem ao mesmo *bitrate* de compressão ou, mantendo a mesma qualidade de imagem, diminuir o *bitrate* de compressão [5]. Um exemplo pode ser constatado na Fig. 2.12.



Figura 2.12: Mesmo frame de um vídeo em MPEG-2 (esquerda), MPEG-4 (centro) e h.264/AVC (direita) ao mesmo *bitrate* de compressão[5]

## Capítulo 3

# Caracterização do Problema

### 3.1 Definição do problema

Tendo como objetivo revitalizar o projeto "RobVigil" de forma a aproveitar o máximo possível o software já desenvolvido e mantendo as suas funcionalidades, surgiu uma oportunidade para reapropriar o trabalho efetuado com a ajuda do framework ROS. Mantendo o inerente espírito do próprio, seria necessário adaptar o robô de forma a que conseguisse funcionar com este e de seguida tentar transportar o software antigo novamente para o robô. É tido em consideração que os próprios repositórios de ROS já contêm várias ferramentas com as quais seria possível desenvolver um projeto similar ao original. A nível de hardware existem também dois melhoramentos que se pretende alcançar: complexidade e custo. A estrutura mecânica original (Fig. 3.1) encontra-se desorganizada e pouco escalável. Sendo assim, será necessário reestruturar e reorganizar o hardware de modo a que este seja menos complexo, mais simples de interpretar e de fácil integração de novos componentes.

### 3.2 Solução Proposta

Foi, então, feito um estudo das diferentes técnicas de localização e navegação de robôs autônomos bem como dos *packages* existentes já desenvolvidos em ROS. Tendo como finalidade a incorporação dos algoritmos desenvolvidos para o projeto original, será também revisto todo o seu software de forma a poder incorporá-lo no novo framework.

Sendo a integração da *stack* de navegação do ROS o componente principal para o funcionamento do robô, é necessário desenvolver um driver para movimentar o robô, boas leituras dos sensores e transmissão apropriada desses dados, definir as relações físicas entre os diferentes referenciais do robô e criar de um mapa onde seja possível testar o seu funcionamento.

Após o desenvolvimento de uma primeira versão totalmente baseada em software *open source* como prova de conceito, serão explorados diversos melhoramentos, nomeadamente no tipo de localização e na incorporação do *Localization Perfect Match* com o objetivo de obter os melhores resultados possíveis. Adicionalmente, de forma a que o robô se consiga desviar de todo o tipo

de obstáculos, será também explorada a possibilidade de criação de mapas 3D. Desta forma, será possível obter toda a informação possível sobre um dado local e incorporá-la no mapa final.

A estrutura mecânica do robô terá, numa primeira fase, apenas os motores, um laser e uma câmara catadióptrica juntamente com um espelho convexo. Deste modo, terá uma configuração simples e capaz de transmitir por vídeo, capturado apenas por uma câmara, toda a sua área circundante. Após alcançar resultados satisfatórios, será implementado um novo laser na frente do robô apontado para baixo com uma ligeira inclinação. Este permitirá detetar obstáculos que não são visíveis pelo laser montado na parte superior.



Figura 3.1: Estrutura mecânica do *RobVigil*

## Capítulo 4

# Robô Vigilante

Neste capítulo será apresentada a estruturação e os componentes principais do Robô Vigilante tanto a nível de *Hardware* como de *Software*.

### 4.1 Hardware

A montagem do robô foi baseada na configuração do projeto original, mas utilizando uma base diferente.

Na sua base inferior estão montados dois motores juntamente com *encoders* diferenciais. Estes são a fonte de movimento do robô, estando ligados a duas rodas. Do lado oposto, existe uma roda louca que estabiliza o robô numa montagem diferencial. Entre as duas rodas está situada uma bateria de 12V.

No bloco central está situado o computador (ASUS VivoMini UN65H), bem como um router que permite ligação de outros elementos por fio e gerar uma rede wireless a partir da qual o computador pode comunicar para o exterior do robô.

Na sua parte superior estão ligados dois lasers (Fig. 4.4), um para localização (*lase\_top*) e outro para deteção de obstáculos (*laser\_front*). O primeiro está montado no topo do robô e o segundo na frente, com uma inclinação de cerca  $-79^\circ$  com o plano vertical (Fig. 4.5). Está, ainda, ligada a câmara numa configuração catadióptrica vertical. Estes três componentes estão interiormente ligados ao computador, seja por cabo ethernet por intermédio do router ou diretamente por USB.

Para o controlo dos motores, existem dois Arduinos, um MEGA 2560 e um Nano. O primeiro controla o movimento dos motores por intermédio de pontes IBT-2 (H) e está constantemente em comunicação com o computador por porta série. Já o Arduino Nano tem o único propósito de receber a informação dos *encoders* das rodas e transmiti-la para o outro micro controlador (topologia dos Drivers da equipa de futebol robótico do INESC).

Em termos de segurança, foi montado no topo do robô um botão de emergência que corta a alimentação dos motores, bem como um disjuntor entre a bateria e o resto do circuito elétrico do robô funcionando como interruptor "On/Off". Este último encontra-se na parte lateral do robô.

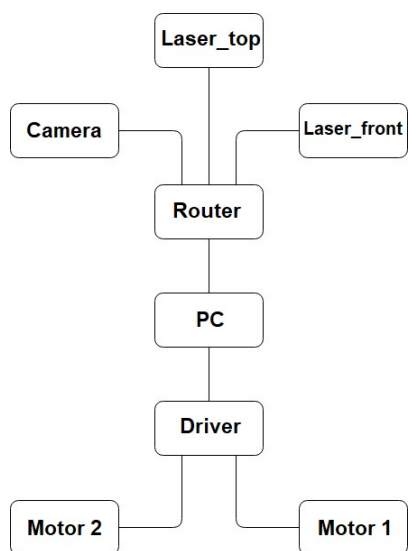


Figura 4.1: Diagrama do hardware

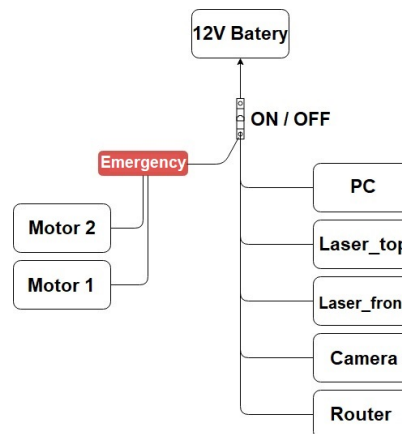


Figura 4.2: Diagrama das funções de segurança



Figura 4.3: Vista frontal do robô

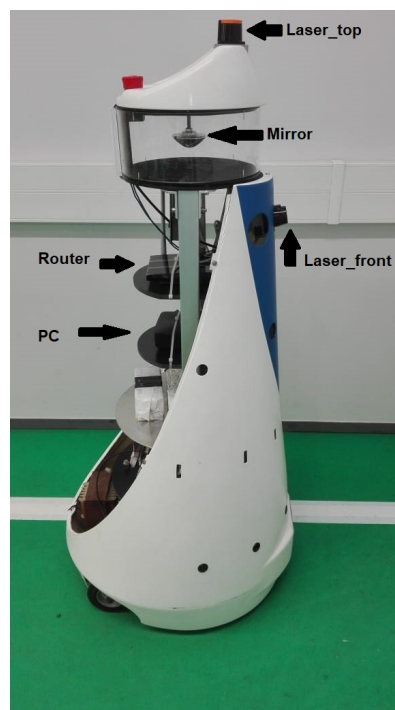


Figura 4.4: Vista lateral do robô (aberto)

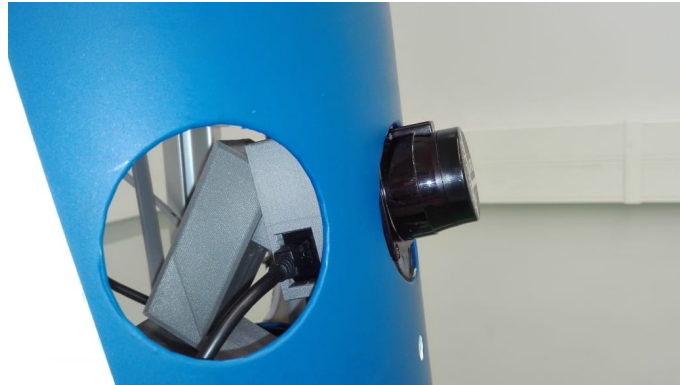


Figura 4.5: Vista aproximada do *laser\_front*



Figura 4.6: Vista interior do *laser\_front*



Figura 4.7: Câmara montada verticalmente



Figura 4.8: espelho concavo diretamente acima da câmara

## 4.2 Software

Foi utilizado como framework de base no robô a versão mais recente do ROS - (Melodic Morenia). Para tal, o computador do robô tem também instalado o Ubuntu 18.04 (Bionic Beaver) dado que, para cada distribuição de ROS, uma versão específica de Ubuntu é necessária (versões mais antigas ou seguintes não tem compatibilidade com o Melodic, salvo algumas exceções).

### 4.2.1 Driver

Os objetivos do Driver<sup>1</sup>, desenvolvido em ROS, são mover o robô e receber informação dos *encoders*. Para tal, foi desenvolvido um *package* baseado no já existente *rosserial\_server* que escreve e lê informações pela porta série diretamente para o Arduino MEGA.

Ao executar o Driver, inicializam-se dois nós, *igus\_driver\_node* e *joystick\_subscriber*. O primeiro, trata da comunicação e tratamento de dados, lendo mensagens escritas no tópico *joint\_state\_goal* para depois as enviar devidamente tratadas para o arduino e lê informação do arduino publicando-as no tópico *joint\_state*. Já o segundo, lê informações de velocidades do tópico *cmd\_vel* (sendo compatível com o teleop do ROS) e, após criação dos devidos *packets*, escreve informação no tópico *joint\_state\_goal*.

### 4.2.2 Mapeamento

Para gerar os mapas foi utilizado o *package hector\_slam*. Este é extremamente versátil, pois apenas necessita de leituras provenientes de um LIDAR para a construção do mapa e consegue fazê-lo com precisão suficiente para ser usado em vários cenários reais. De forma a obter os dados do laser, foi utilizado o *package urg\_node* que permite publicar a informação que recebe do laser no tópico *\scan*. Estando a informação proveniente do laser a ser recebida, falta apenas definir uma relação entre os referenciais do laser e robô ( a relação entre o mapa e o robô é gerada pelo próprio). Neste caso prático, o laser encontra-se, partindo do centro do eixo das rodas do robô, deslocado de 30 cm na direção *x*, 125 cm na direção *z* e com uma rotação de cerca de  $-3.5^\circ$  à volta do eixo *y* (*pitch*). Esta relação pode ser representada em ROS utilizando o *package tf* que será exposto em 4.2.4.

Assim sendo, utilizando o driver desenvolvido (4.2.1), conduz-se o robô pela área que se deseja mapear, enquanto o *hector\_slam* está em funcionamento. Este estabelece uma relação entre as medidas do laser e o quanto o robô se desloca do seu ponto inicial para gerar um mapa que pode ser observado, à medida que este é criado, no *rviz* (ferramenta de visualização 3D do ROS).

---

<sup>1</sup>[https://drive.google.com/open?id=1T9eRkNzhe-MHAK\\_T74qxLilIwFXwliLq](https://drive.google.com/open?id=1T9eRkNzhe-MHAK_T74qxLilIwFXwliLq)

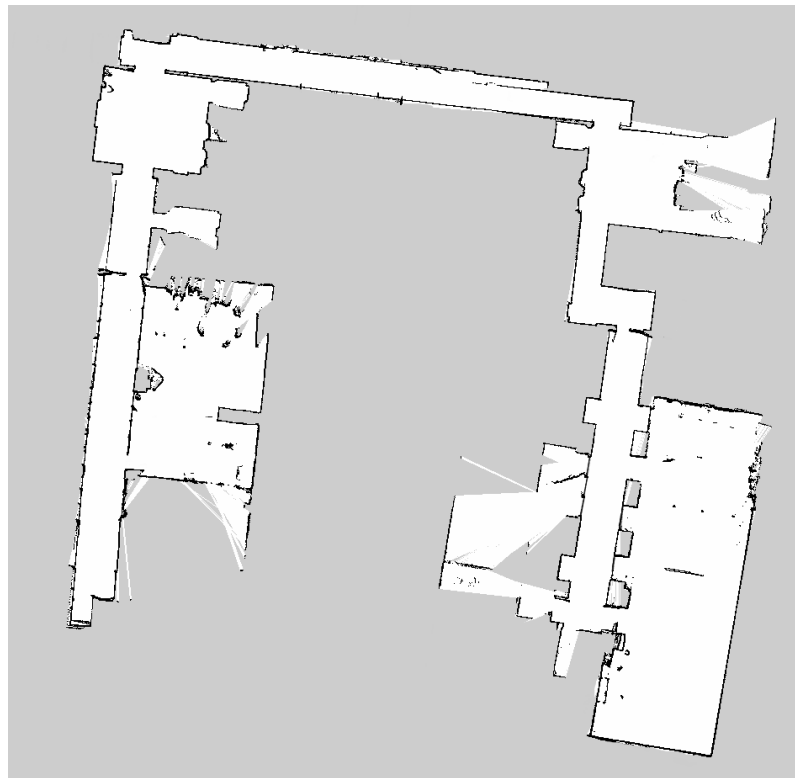


Figura 4.9: Mapa 2D obtido pelo *hector\_slam*

Seguidamente, de forma a ser possível mapear possíveis obstáculos que não sejam detetados pelo *laser\_top* (ou seja, que tenham uma altura inferior a 125 cm), foi gerado um mapa 3D a partir das leituras do *laser\_front*. A sua inclinação permite, da mesma forma que o mapa 2D foi gerado, obter medidas que permitem gerar um mapa 3D do ambiente de trabalho. Para tal, foi utilizado o *package octomap*[37] que permite a criação de uma grelha de ocupação 3D que, utilizando outros recursos desenvolvidos pelo autores (*octomap\_server* e *octomap\_mapping*), é utilizada para representar e guardar um mapa 3D. Para gerar o mapa, é necessário existir uma relação entre o mapa e as leituras do laser e que o robô esteja localizado no mesmo. Assim sendo, a abordagem preferencial consiste na projeção do mapa 2D, na qual é necessário que o robô se localize enquanto o mapa 3D é gerado. Simultaneamente, o *octomap* permite gerar um mapa bidimensional baseado no mapa 3D, segundo alguns critérios. No caso do robô Vigilante, o objetivo é mapear os obstáculos que não sejam visíveis pelo *laser\_top*, sendo pertinente fazer uma compressão do mapa tridimensional de todos os objetos detetados com altura inferior a 125 cm. Na Fig. 4.10 pode observar-se o mapa 3D resultante.

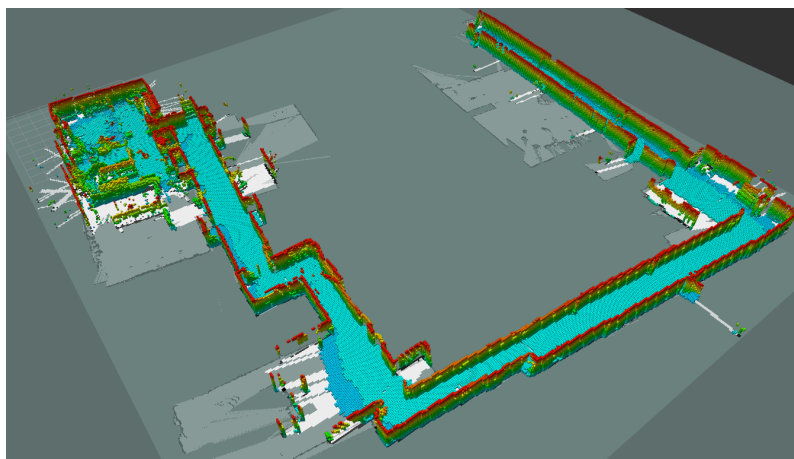


Figura 4.10: Mapa 3D obtido pelo *octomap*

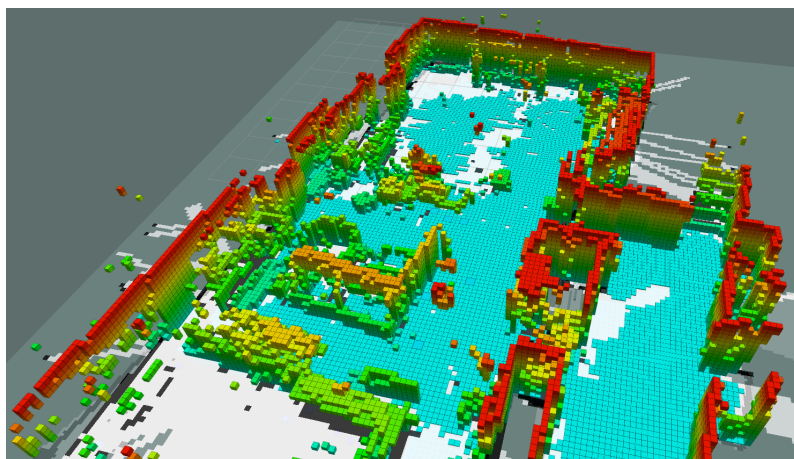
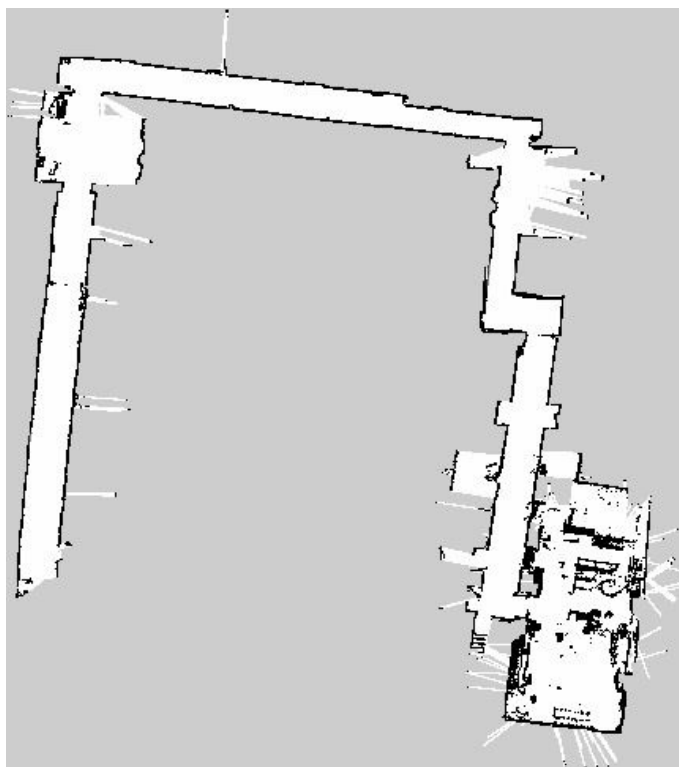


Figura 4.11: Mapa 3D obtido pelo *octomap* da sala inicial

O resultado da compressão do mapa 3D em 2D pelo patamar de altura definido pode ser visualizado na Fig. 4.12. Comparando com o mapa da Fig. 4.9, observa-se uma grande diferença no mapeamento da sala inicial, pois esta tinha diversos objetos espalhados com menos de um metro de altura. O mapa 3D prova, então, ser uma mais valia nestas condições, permitindo visualizar realmente todos os obstáculos e calcular trajetórias que os evitem.

Possuindo estes dois mapas, o robô é capaz de navegar autonomamente no ambiente sendo o primeiro mapa, obtido pelo *hector\_slam*, utilizado para fins de localização e o segundo, obtido pelo *octomap*, para o planeamento de trajetórias que será exposto em detalhe em 5.2.

Figura 4.12: Mapa 2D obtido pelo *octomap*

### 4.2.3 Câmara e transmissão de vídeo

Para obter um feed contínuo da câmara, foi utilizado o pacote *pylon\_camera*<sup>2</sup>. Este funciona como um driver para câmaras Basler, desenvolvido pelos próprios. Mesmo sendo possível a transformação de imagens omnidirecionais em imagens planares em ROS, como estas têm de ser transmitidas, não é aconselhável fazê-lo, uma vez que a imagem original contém exatamente a mesma informação e é transmitida com uma largura de banda muito inferior (imagens panorâmicas têm dimensões bastante grandes).

A imagem obtida a partir do driver da câmara utilizado pode vir codificada de duas formas, monocromático 8 bits (mono8) ou *blue-green-red* 8 bits (bgr8), podendo alternar entre imagem *grayscale* ou a cores. Esta é, então, publicada num tópico (por defeito: *pylon\_camera\_node/image\_raw*). Estando a imagem a ser publicada, foi utilizado o *package web\_video\_server* para transmitir a imagem por Wi-Fi. Após iniciar o *web\_video\_server* basta aceder ao URL: <http://192.168.0.144:8080/>, onde serão apresentados todos os tópicos de ROS que estão a receber informações de imagens. Selecionando o tópico correspondente, é gerado um URL final onde é transmitido diretamente o vídeo.

---

<sup>2</sup>[http://wiki.ros.org/pylon\\_camera](http://wiki.ros.org/pylon_camera)

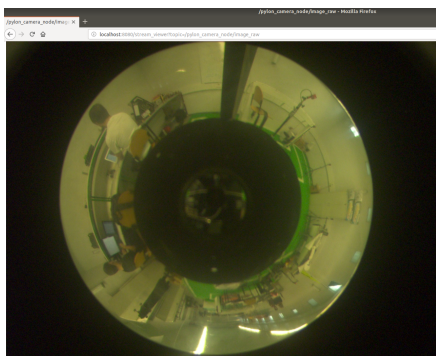


Figura 4.13: Transmissão do vídeo da câmera a cores

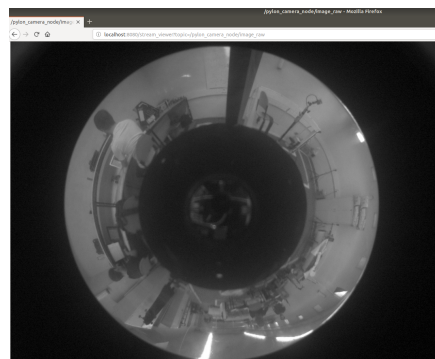


Figura 4.14: Transmissão do vídeo da câmera em *grayscale*

#### 4.2.4 Referenciais

As relações e interações entre os referenciais dos diferentes componentes de um robô são fundamentais para compreender e controlar o seu movimento. O *package tf*<sup>3</sup>, que permite uma simples definição de referenciais, das suas relações com outros e atualiza essas relações durante o funcionamento do robô, é considerado essencial no desenvolvimento de qualquer robô para funcionar com ROS. A configuração do robô Vigilante é bastante simples e tem 5 referenciais:

- **map** — Este é o referencial do mapa no qual o robô se vai localizar; é o referencial global ou que corresponde ao mundo e é a raiz de todos os outros referenciais pois nunca se altera;
- **odom** — O referencial *odom* é parecido com o *map*. Tem duas grandes diferenças. A sua origem coincide com a pose inicial do robô e não é fixo. Partindo deste referencial, o movimento do robô é calculado (odometria), sendo espectável, então, que fosse fixo, porém, na prática, existem vários fatores que podem gerar distúrbios na odometria o que leva a uma correção da posição do referencial;
- **base\_link** — O referencial do robô propriamente dito. A relação entre este referencial e *odom* é correspondente ao seu movimento. Tipicamente, a origem deste referencial é o centro do eixo das rodas do robô;
- **laser** — Este referencial corresponde à posição do laser montado no topo do robô; é o laser usado para a sua localização. A relação entre este referencial e o *base\_link* nunca se altera e já foi exposta em 4.2.2;
- **laser\_front** — Este referencial corresponde ao referencial do laser colocado na frente do robô apontado para baixo com uma ligeira inclinação. Foi definida a relação entre este referencial e o *laser* como uma translação no eixo *x* de 11 cm, no eixo *z* de -35 cm e uma rotação em torno do eixo *y* (*pitch*) de -79°. Tal como o referencial *laser*, esta relação não se altera com o tempo.

<sup>3</sup><http://wiki.ros.org/tf>

As relações dos lasers, como não se alteram, são criadas com o *static\_transform\_publisher*, presente no *tf*. Já as relações entre o *map* - *odom* e *odom* - *base\_link* são dinâmicas e têm de ser calculadas e publicadas de acordo com a informação recebida. A primeira é tipicamente fornecida pelo método de localização do robô e a segunda é normalmente calculada pelo próprio robô.

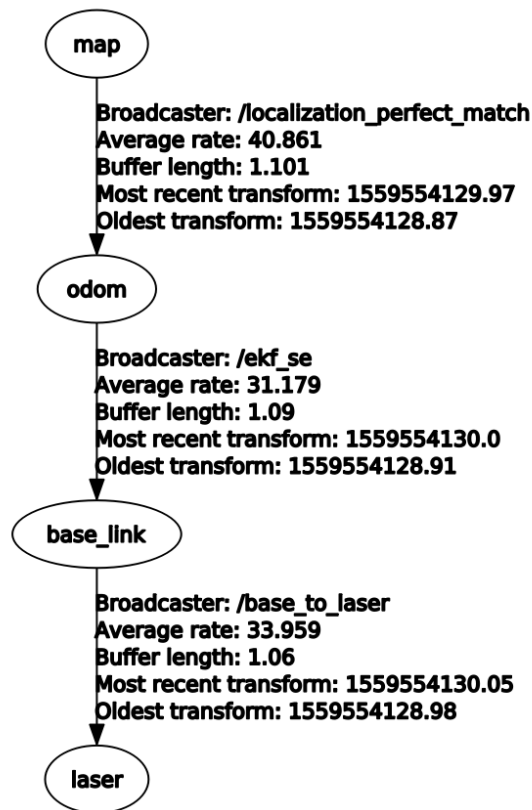


Figura 4.15: Exemplo de relações existentes entre os referenciais (*tf tree*)

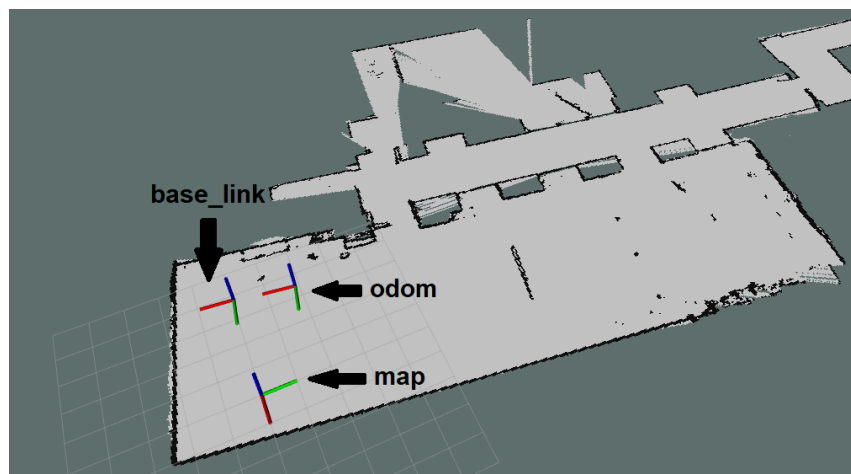


Figura 4.16: Referenciais representados no mapa

### 4.2.5 Odometria

A odometria, *i.e.* a estimativa da posição do robô num dado referencial, é naturalmente uma componente fundamental para a localização do robô. Para o robô Vigilante, foram implementadas três abordagens diferentes: computação direta do seu movimento a partir dos *encoders* das rodas, uma estimativa do deslocamento a partir de leituras sucessivas de um LIDAR e uma estimativa da odometria através da fusão dos dados obtidos pelas duas anteriores.

Para o cálculo da odometria das rodas, foi desenvolvido um nó de ROS (*wheel\_odometry*), que se subscrive ao tópico *joint\_state*, onde é publicada a informação relativa aos *encoders*. Este recebe informação quanto ao número de impulsos contados pelos *encoders* de cada roda a cada dez milissegundos ( $t_c$  - tempo de ciclo do arduino) e aplica a fórmula  $v_x = \frac{enc_x}{I \cdot t_c}$  em que  $v_x$  é a velocidade na roda  $x$ ,  $enc_x$  é o número de impulsos lidos pelo *encoder* da roda  $x$  e  $I$  é o número de impulsos por metro, obtido através de experimentação. Após calcular a velocidade de cada roda, é possível encontrar a velocidade linear  $v$  e angular  $w$  com as equações de cinemática de um robô diferencial[1] :

$$v = \frac{v_1 + v_2}{2} \quad (4.1)$$

$$w = \frac{v_1 - v_2}{b} \quad (4.2)$$

em que  $b$  é a distância entre rodas. Finalmente, para estimar a pose do robô ( $x, y$  e  $\theta$ ), dado que se trata de um robô diferencial, a velocidade no eixo  $y$  do referencial do robô (*base\_link*) em relação ao referencial da odometria (*odom*) é sempre zero, pois este andarás apenas na direção  $x$  (ao rodar, o referencial acompanha a rotação), levando às seguintes equações:

$$\Delta x = v \cos(\theta) t_c \quad (4.3) \quad x = x + \Delta x \quad (4.6)$$

$$\Delta y = v \sin(\theta) t_c \quad (4.4) \quad y = y + \Delta y \quad (4.7)$$

$$\Delta \theta = w t_c \quad (4.5) \quad \theta = \theta + \Delta \theta \quad (4.8)$$

A segunda abordagem, provém da utilização do *package laser\_scan\_matcher*<sup>4</sup>. Este subscrive a um tópico que recebe informações de um LIDAR. Para a configuração do robô Vigilante, o *laser\_top* é ideal para ser utilizado com este *package*, pois obtém medidas de distância às paredes maioritariamente não-obstruídas. O seu princípio de funcionamento baseia-se no *Canonical Scan Matcher*[38] desenvolvido por Andrea Censi. Dadas duas mensagens que contenham a informação do LIDAR num dado intervalo de tempo ( *i.e.* várias medidas correspondentes a diversos ângulos de 0° a 270°), compara as medições e estima o deslocamento do robô para computar a sua odometria. Torna-se possível utilizar o *package* para estimar a odometria do robô sem a necessidade de usar os *encoders* do robô, porém, caso essa informação esteja disponível, pode ser

<sup>4</sup>[http://wiki.ros.org/laser\\_scan\\_matcher](http://wiki.ros.org/laser_scan_matcher)

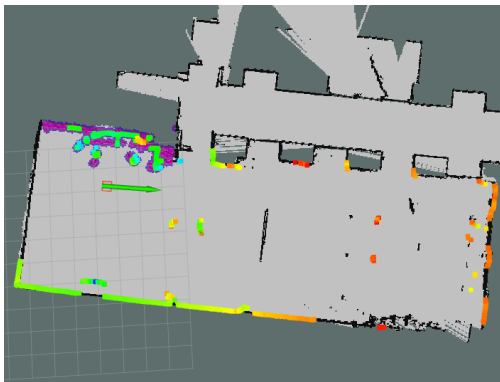


Figura 4.17: Exemplo do Robô localizado

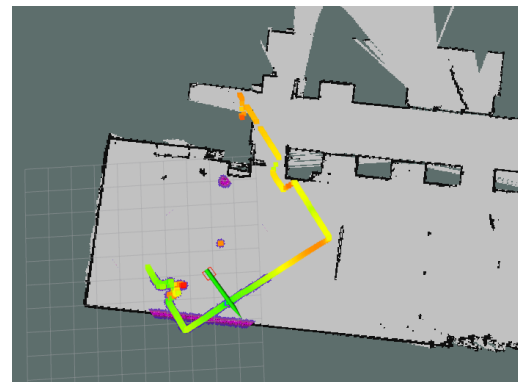


Figura 4.18: Exemplo do robô mal localizado (perdido)

introduzida para uma previsão da odometria estimada, acelerando e melhorando o seu funcionamento.

Contudo, estas duas abordagens não são perfeitas. A odometria convencional está sempre sujeita a muitos erros de origem mecânica - pavimento irregular, derrapamento das rodas, mau funcionamento dos *encoders* - o que leva à diminuição da qualidade da estimativa quanto maior for o trajeto percorrido pelo robô. Uma estimativa da odometria calculada a partir de medidas sucessivas de um LIDAR está sujeita a erros muito grandes em dois casos: obstrução do laser e localização em corredores muito longos. No primeiro caso, ao obstruir o laser, as medidas detetam um salto muito grande mas como, passado alguns ciclos de funcionamento, a estimativa de deslocamento é feita com base em medidas já oriundas da obstrução, esta fica permanentemente alterada para valores errados e imprevisíveis. O segundo, é um caso comum na prática, estando o robô num corredor muito longo, as medidas às paredes mais próximas (laterais) não se alteram e as medidas às paredes mais distantes (frontais) podem ser mal detetadas. Com esta informação, o robô apenas calcula que robô não se movimentou para os lados e pode não conseguir calcular o quanto se deslocou para a frente. A má detecção deve-se à distância da parede ao robô, quanto mais distante, menos precisa é a medida do laser, podendo mesmo não haver medida do todo.

A terceira abordagem visa eliminar estes problemas. Utilizando a técnica de fusão de dados, baseada num Filtro de Kalman Estendido (EKF), a estimativa filtrada da odometria tem em consideração ambas as medidas bem como a probabilidade de estas estarem certas. Assim sendo, para qualquer instante de tempo ou para qualquer situação, a odometria tem a maior probabilidade de estar correta, permitindo a recuperação da pose do robô no caso de uma das fontes se "perder". Esta abordagem é posta em prática através do *package robot\_localization* que permite a fusão de vários tipos de dados de um número arbitrário de fontes. No caso do robô Vigilante existem duas fontes que fornecem o mesmo tipo de dados, a pose do robô ( $x, y$  e  $\theta$ ), expressa no referencial *odom*: uma com o formato típico de mensagens de odometria em ROS (*nav\_msgs::Odometry*) e outra com o formato de uma informação de pose (*geometry\_msgs::PoseWithCovarianceStamped*). Ambos os formatos, entre outros, são suportados pelo *package* e necessitam que venham preenchidos com

valores de covariância associados a cada medida - quanto mais pequena, mais confiança se tem de que a medida está certa. Adicionalmente, a estimativa da pose a partir dos *encoders* das rodas é tratada como diferencial, *i.e.* os dados são derivados em função do tempo ficando-se então a tratar de velocidades, visto que não é adequado ter duas fontes a fornecer o mesmo tipo de dados diretamente e a odometria das rodas é *a priori* menos precisa que a do *laser\_scan\_matcher*.

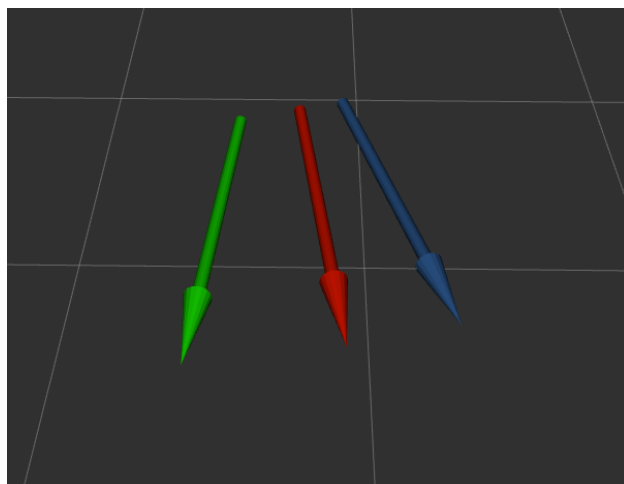


Figura 4.19: Pose dada por odometria (verde), pelo *laser\_scan\_matcher* (azul) e o resultado da fusão (vermelho)

#### 4.2.6 Detecção de obstáculos fatais

A configuração do robô, tendo em conta apenas o *laser\_top*, ainda que funcional, está sujeita a um problema: obstáculos com altura inferior ao laser. Para combater este problema, foi gerado o mapa 3D com o *laser\_front* para fundir essa informação com o mapa já gerado, de forma a ter em conta todos os obstáculos existentes no momento do planeamento de trajetórias. Porém, esta solução não tem em consideração obstáculos desconhecidos ou dinâmicos que possam impossibilitar o bom funcionamento do robô Vigilante. Assim sendo, foi desenvolvido um nó de ROS que recebe continuamente informação das leituras do *laser\_front* que implementa uma "zona proibida". Esta zona é uma caixa retangular ligeiramente em frente ao robô (Fig. 4.20) que, caso o laser detete algo seja um obstáculo estático desconhecido, obstáculo dinâmico ou até mesmo uma pessoa que inesperadamente se coloque em frente ao robô, bloqueando o seu movimento, envia um sinal de paragem ao robô por motivos de segurança.

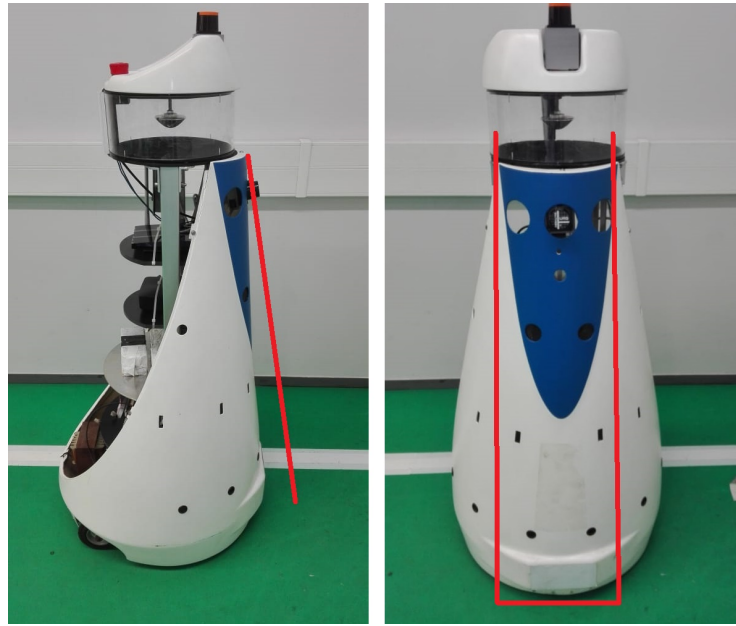
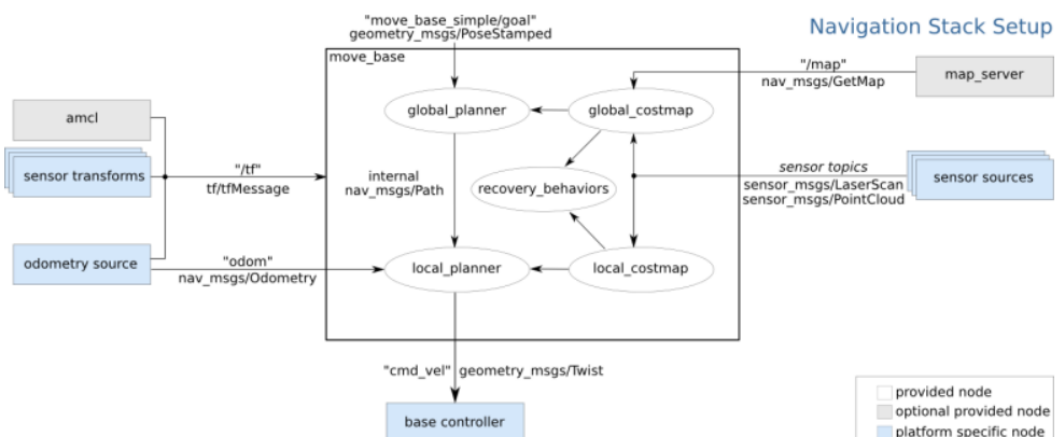


Figura 4.20: Zona proibida (vermelho)

### 4.2.7 Navegação

A implementação da *stack* de navegação do ROS tem quatro pré-requisitos para o seu funcionamento (Fig. 4.21, evidenciados a azul). Estes já foram expostos: a definição dos referenciais usando o *tf* em 4.2.4, um driver que movimenta o robô com comandos recebidos no tópico *cmd\_vel* em 4.2.1, tratamento da informação recebida pelos sensores (lasers) em 4.2.2 e a publicação da odometria do robô, tanto diretamente através de um tópico como através do *tf* em 4.2.5.

Figura 4.21: Configuração necessária da stack de navegação do ROS<sup>5</sup>

<sup>5</sup><http://wiki.ros.org/navigation/Tutorials/RobotSetup>

Estando cumpridos os quatro pré-requisitos, já existe uma plataforma base para a implementação da *stack* de navegação. Esta permite a escolha entre vários métodos de planeamento de trajetórias diferentes. Todos estes métodos são utilizados em conjunto com os *packages* *move\_base*<sup>6</sup> e *costmap\_2d*<sup>7</sup> para a implementação das trajetórias e a criação de mapas de grelha, a partir dos mapas gerados previamente. A boa definição dos parâmetros destes dois *packages* é crucial para o funcionamento da *stack* pois pequenas alterações podem fazer grandes diferenças. Estes serão explorados em mais detalhe no capítulo seguinte.

---

<sup>6</sup>[http://wiki.ros.org/move\\_base](http://wiki.ros.org/move_base)

<sup>7</sup>[http://wiki.ros.org/costmap\\_2d](http://wiki.ros.org/costmap_2d)

## Capítulo 5

# Configuração da *stack* de Navegação

### 5.1 Mapa de custos

Os mapas de custos são uma peça essencial para o planeamento de trajetórias. Após a criação de um mapa de grelha, a partir de um dado mapa, são atribuídos diferentes valores às células resultantes que entram em consideração ao computar a trajetória. Tipicamente, essas funções visam minimizar esse tal custo, *i.e.* encontrar uma trajetória que, somando os custos das células em que passa, tem um resultado final inferior a qualquer outra. Regra geral, quanto mais próxima de uma célula ocupada (ou seja, um obstáculo), maior o custo associado.

Podem então expor-se certas questões: a partir de que distância a um obstáculo se atribui custos? Como é que estes custos evoluem de acordo com a proximidade? A partir de que momento um custo se torna demasiado grande? Qual a resolução das células?

De forma a poder controlar todos estes aspetos e muitos mais, existe sempre um ficheiro de parâmetros associado ao mapa criado.

A abordagem mais comum, para efeitos de navegação, é a criação de dois mapas - um global e outro local. O mapa global gere os custos do mapa na sua totalidade, já o mapa local apenas tem em consideração os custos das células até uma certa distância do robô (*e.g.* uma janela quadrada de 5m à volta do robô). Para lidar com os dois mapas, são geralmente criados três ficheiros de parâmetros: um para os parâmetros do mapa local, outro para os do mapa global e um terceiro para os parâmetros comuns aos dois.

### 5.2 Algoritmos de planeamento de trajetórias

Tal como foi referido anteriormente, a *stack* de Navegação fornece vários algoritmos de planeamento de trajetórias. Estes podem ser divididos em dois grupos: local e global, tal como os mapas de custos. O planeador global computa a trajetória, caso ela seja possível, desde o ponto de origem até ao objetivo. Já o planeador local trata de movimentar o robô, o máximo possível de acordo com a trajetória fornecida pelo planeador global, evitando obstáculos que se encontrem a uma determinada distância.

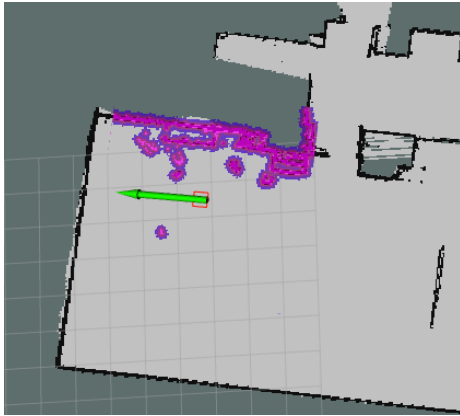


Figura 5.1: Mapa de custos local (camada roxa)

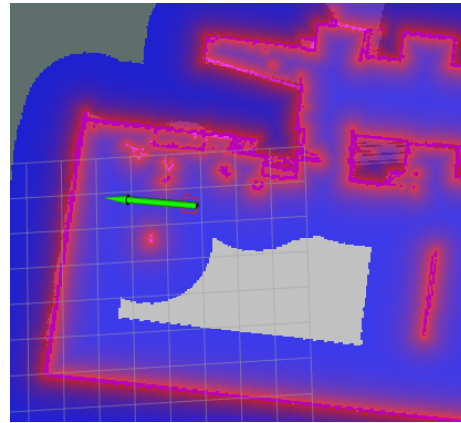


Figura 5.2: Mapa de custos global (camada azul)

São fornecidos dois *packages* para planeamento de trajetórias locais pela *stack* de Navegação: *base\_local\_planner* e *DWA\_local\_planner*. Já para planeamento de trajetórias globais, são fornecidos três: *carrot\_planner*, *navfn* e *global\_planner*.

### 5.2.1 Planeadores locais

O *base\_local\_planner* é o *package* pré-definido para planeamento de trajetórias da *stack* de navegação. Este implementa duas abordagens: *Trajectory Rollout* (TR) e *Dynamic Window Approach* (DWA). O funcionamento base de ambas é o seguinte:

- 1 — Amostrar discretamente o espaço de controlo do robô ( $\dot{x}, \dot{y}, \dot{\theta}$ );
- 2 — Para cada velocidade amostrada, executar uma simulação a partir do estado atual do robô para prever o que aconteceria se as velocidades amostradas fossem aplicadas (ou seja, seguir a trajetória correspondente) durante um certo período de tempo, geralmente pequeno;
- 3 — Avaliar (atribuir uma pontuação) cada trajetória resultante a partir das simulações usando uma função que incorpora características como: proximidade a obstáculos, proximidade ao objetivo, proximidade à trajetória global e velocidade. Descartar trajetórias inválidas (que colidem com obstáculos);
- 4 — Escolher a trajetória com a maior pontuação e enviar as velocidades associadas ao controlador da base móvel;
- 5 — Repetir até chegar ao objetivo;

A diferença entre as duas abordagens é apenas a maneira como amostram o espaço de controlo (primeiro passo). A TR amostra as velocidades possíveis para todo o período de simulação dados os limites de aceleração do robô. Já a DWA faz o mesmo mas apenas durante um passo de simulação. Sendo assim, a DWA é a abordagem mais eficiente mas pode vir a produzir uma trajetória menos indicada do que uma calculada a partir da TR em situações onde os limites de

aceleração do robô são pequenos, por exemplo. Na prática, em casos gerais, é recomendado utilizar apenas a DWA, uma vez que a diferença de desempenho é quase impercetível e é a abordagem mais eficiente.

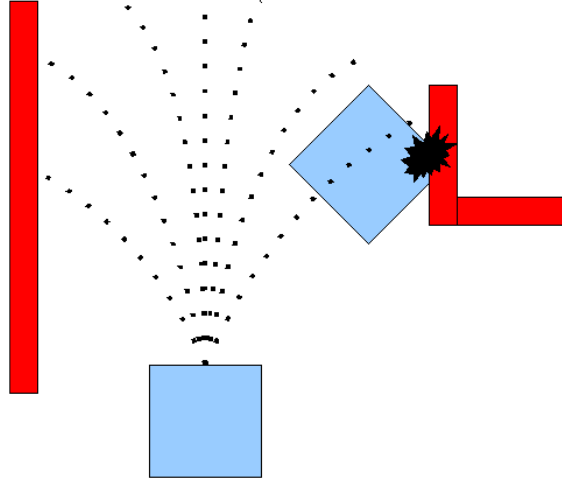


Figura 5.3: Exemplo ilustrativo das simulações dadas certas velocidades amostradas<sup>1</sup>

Por fim, o *dwa\_local\_planner* implementa apenas a abordagem DWA. O algoritmo é o mesmo mas com pequenas diferenças de implementação que visam melhorar ainda mais o seu desempenho, mantendo a eficiência. Este é o algoritmo de planeamento de trajetórias locais mais utilizado.

### 5.2.2 Planeadores globais

O *carrot\_planner* implementa um algoritmo muito simples, apenas criando uma trajetória em linha reta em direção ao objetivo e definindo como novo objetivo um ponto nessa reta que seja legal, isto é, que seja possível de alcançar. Este algoritmo é apenas usado como prova de conceito pois não é dinâmico e não permite o desvio de obstáculos. Por estas razões não foi testado.

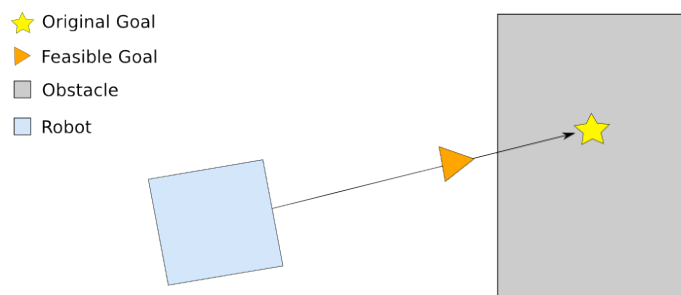


Figura 5.4: Exemplo de uma trajetória gerada pelo *carrot\_planner*<sup>2</sup>

O *navfn* é o *package* pré-definido para planeamento global da *stack* de Navegação. Este implementa uma função de navegação que permite a criação de planeamento de trajetórias para uma

<sup>1</sup>[http://wiki.ros.org/global\\_planner?distro=melodic](http://wiki.ros.org/global_planner?distro=melodic)

base móvel, com recurso a mapas de custo e assumindo um robô circular (garantido pelo uso do *costmap\_2d*). O algoritmo usado para o planeamento em si é o algoritmo de *Dijkstra*. Na Fig. 5.5, note-se que, embora o seu funcionamento geral seja satisfatório, o ponto inicial não corresponde realmente à origem do robô, visto que as coordenadas tem uma imprecisão de cerca de meia célula.

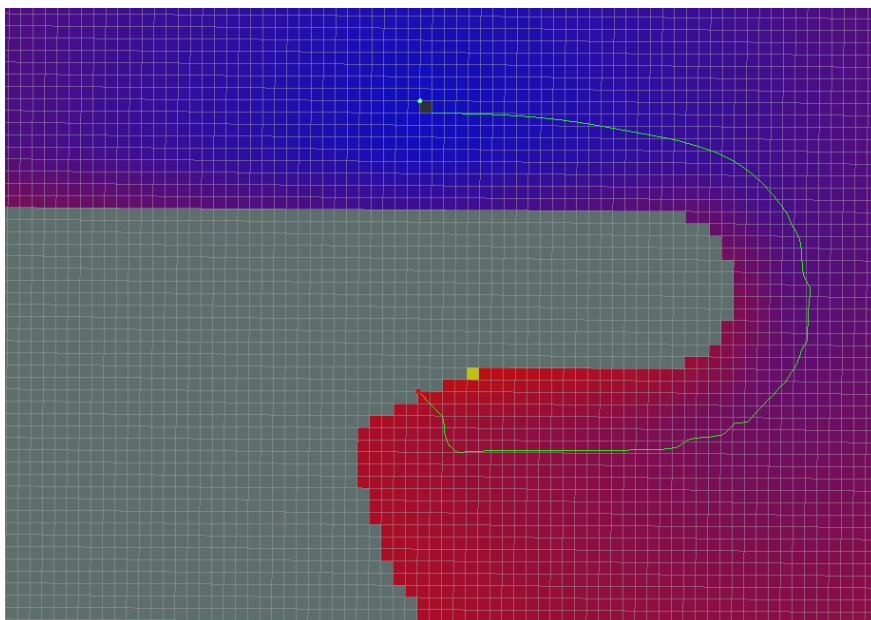


Figura 5.5: Exemplo de uma trajetória global criada pelo *navfn* [6]

Finalmente, o *global\_planner* é um *package* que foi desenvolvido com intuito de ser um substituto mais flexível do *navfn*. Este permite, a quem o utilizar, uma ampla escolha de algoritmos para o planeamento de trajetórias: Dijkstra, A\* ou até mesmo o algoritmo do próprio *navfn*. Além disso, tem a opção de seguir um percurso por células inteiras (*Grid Path*) e de utilizar uma aproximação quadrática para o cálculo do potencial.

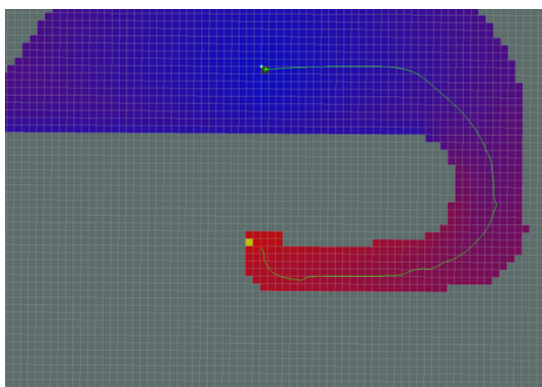


Figura 5.6: Trajetória com A\*[6]

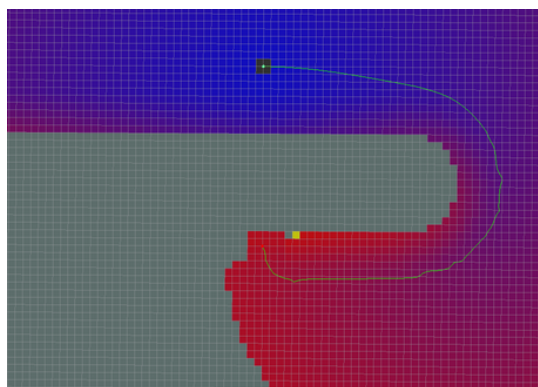


Figura 5.7: Trajetória com Dijkstra[6]

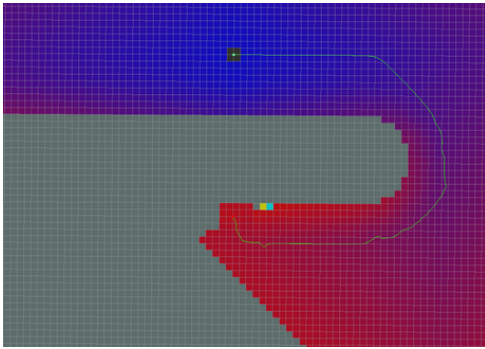


Figura 5.8: Trajetória (Dijkstra) sem aproximação quadrática do Potencial[6]

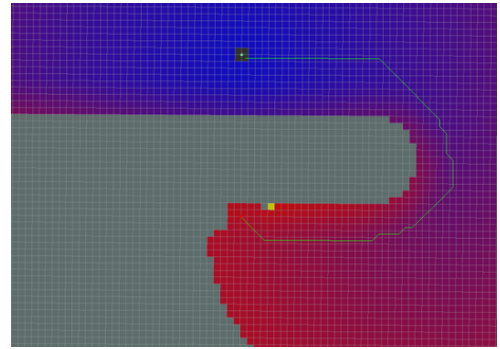


Figura 5.9: Trajetória (Dijkstra) com Grid Path[6]

### 5.3 *move\_base*

O *move\_base* é o *package* central da *stack* de Navegação. Este permite que, dado um objetivo ( ponto de destino) no mapa, o robô se desloque até ao mesmo. Para tal, utiliza as ferramentas já expostas: mapas de custo e planeadores de trajetórias locais e globais, bem como os pré-requisitos enumerados em 4.2.6.

Adicionalmente, são implementados comportamentos de recuperação, *i.e.* como o robô se comporta quando algo corre mal (ficar "perdido", não conseguir encontrar uma trajetória válida, etc...). O comportamento de recuperação de um robô é fulcral para o desenvolvimento de um robô autónomo, na prática, pois existem vários fatores imprevisíveis que podem prejudicar o bom funcionamento dos vários processos que estão a ser executados. Estes são principalmente do tipo *reset* ou *rotation* (Fig. 5.10). A primeira, consiste na tentativa de recomeçar a navegação com a esperança de conseguir que o robô volte a um estado mais estável em que consiga prosseguir com a trajetória. Já a segunda, tenta fazer uma atualização da localização e do mapa de custos local na área circundante ao robô rodando sobre si próprio.

*move\_base* Default Recovery Behaviors

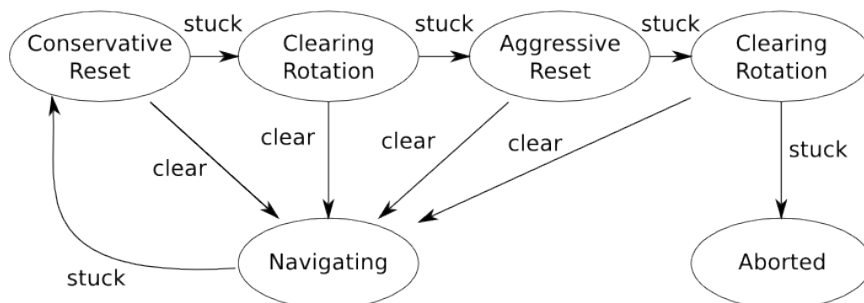


Figura 5.10: Diagrama de comportamentos de recuperação do *move\_base*

<sup>2</sup>[http://wiki.ros.org/carrot\\_planner?distro=melodic](http://wiki.ros.org/carrot_planner?distro=melodic)

## 5.4 Parâmetros

Os parâmetros pré-definidos dos diferentes *packages* mencionados neste capítulo são suficientes para um funcionamento básico da *stack* de Navegação. Porém, em ambiente real, o funcionamento básico apresenta resultados muito insuficientes. Assim sendo, a boa definição dos diversos parâmetros disponíveis, documentados ou não (caso sejam necessários) é fundamental para atingir resultados admissíveis e devem ser revistos e/ou alterados tendo em conta o robô que se quer utilizar. Durante o desenvolvimento do robô Vigilante, estes parâmetros foram alterados várias vezes de acordo com as indicações presentes em [7] e [39].

Dentro dos vários parâmetros disponíveis, os que durante o desenvolvimento do robô tiveram o maior impacto (nomeadamente para o problema de movimentar o robô através de passagens estreitas) foram os parâmetros: *sim\_time*, *vx/vtheta samples*, *robot\_footprint* e *inflation\_layer*. Os primeiros dois estão associados aos planeadores de trajetórias e os outros dois aos mapas de custo.

### 5.4.1 *sim\_time* e *vx/vtheta samples*

O parâmetro *sim\_time* define o tempo de simulação na computação da trajetória local (Ponto 2 do funcionamento da DWA em 5.2.1). Um tempo de simulação maior faz com que as trajetórias avaliadas sejam maiores, aumentando, também, o tempo de processamento, embora permita uma definição muito melhor da sua trajetória, tendo maior influência nas trajetórias curvilíneas (Fig. 5.11). Já os parâmetros *vx samples* e *vtheta samples* definem o número de velocidades lineares e angulares que se vão simular. Na Fig. 5.3, cada linha a tracejado representa uma trajetória diferente que foi explorada. O número de velocidades a simular aumenta o número de trajetórias diferentes, pois a simulação computa todos os pares diferentes possíveis de velocidades. Consequentemente, o aumento destes parâmetros também aumenta o tempo de processamento. Porém, também aumenta a probabilidade do robô encontrar a melhor trajetória possível e, caso o aumento não seja uma impossibilidade devido ao CPU, na maior parte dos casos, é aconselhável. Para o robô Vigilante foram utilizados os seguintes valores:

$$sim\_time = 4.0$$

$$vx = 40$$

$$vtheta = 20$$

### 5.4.2 *robot\_footprint* e *inflation\_layer*

O parâmetro *robot\_footprint* define o formato com que o robô é representado nos mapas. Este pode ser definido como um polígono, através de uma série de pontos que serão ligados, ou através de um círculo, sendo necessário fornecer o seu raio. O centro do robô, ponto a partir do qual o formato será desenhado, é a origem do referencial *base\_link*. O parâmetro *inflation\_layer* define a camada de inflação, isto é, a partir de que distância a uma célula ocupada (obstáculo ou parede) é que se atribuem custos às células circundantes. Quanto maior for esta camada, mais definição se tem no cálculo de trajetórias. Regra geral, um bom valor para uso em ambiente interior é um pouco menos da distância entre paredes. Porém, é preciso ter em atenção que na definição do

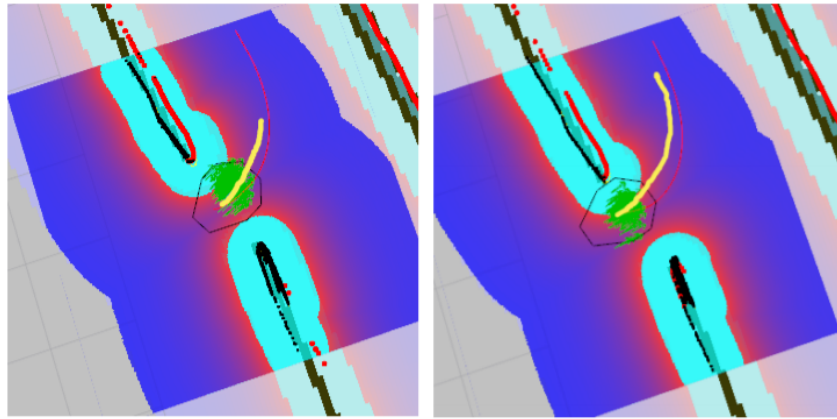


Figura 5.11: Trajetória computada (amarelo) com *sim\_time* pré-definido (esquerda) e com um valor superior (direita)[7]

mapa de custos local, a camada de inflação deve ser relativamente pequena, pois nunca permite a entrada em posições dentro da mesma.

A representação do robô no mapa também é sujeita a uma pequena camada de inflação de forma a tentar assegurar que este não embate em nenhum obstáculo, caso a trajetória designada o movimento relativamente perto destes. Caso o robô seja representado por um polígono, esta camada é equivalente ao círculo circunscrito do próprio e pode observar-se a dispersão dos custos da camada na Fig. 5.12. A camada de inflação é representada a azul na Fig. 5.2.

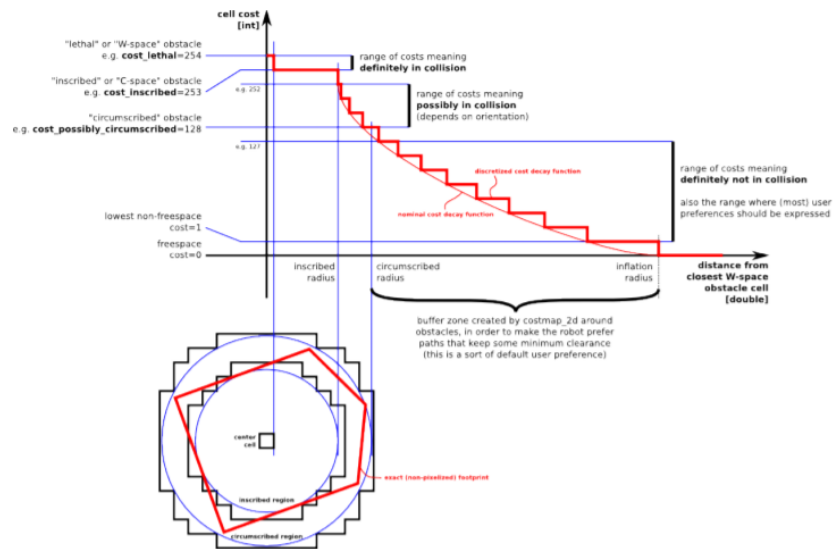


Figura 5.12: Definição do *footprint* do robô[7]



## Capítulo 6

# Análise dos resultados obtidos

### 6.1 Objetivos

O objetivo principal para testar o funcionamento do robô consiste em verificar que este, definindo um ponto num dado mapa, se desloca até ao objetivo autonomamente, desviando-se de obstáculos (não fatais) durante o percurso. Foi escolhido como ponto de partida a sala I-108 (interior) e como objetivo final o laboratório do CRIIS.

Dado este objetivo final, definiram-se sub-objetivos que, não sendo cumpridos, impossibilitam a satisfação do objetivo principal. Estes são: sair da primeira sala (1), passar a primeira porta dupla (2), passar a zona intermédia (3) e atravessar o corredor sem se perder (4). Na Fig. 6.1 estão exemplificados os percursos em questão.

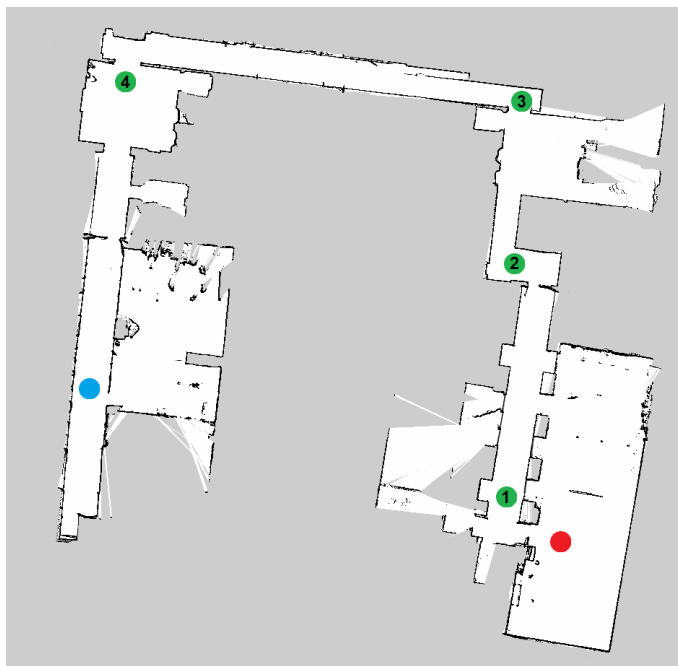


Figura 6.1: Ponto de início (vermelho), objetivo final (Azul) e sub-objetivos (Verde 1-4) no mapa

## 6.2 Primeira solução

A primeira solução implementada foi caracterizada por dois fatores: Localização através do *amcl* (Adaptive Monte Carlo Localization) e odometria através do *laser\_scan\_matcher*. O primeiro é o *package* pré-definido de localização para a *stack de navegação* e é responsável pela relação entre os referenciais *map* e *odom*. Este localiza-se num dado mapa através de um Filtro de Partículas, juntamente com dados provenientes da odometria. Por sua vez, a odometria, bem como a relação entre os referenciais *odom* e *base\_link*, provém do *laser\_scan\_matcher* (exposto em 4.2.5).

Numa primeira fase, foi utilizado o planeador de trajetórias local *base\_local\_planner* visto ser o pré-definido. Após alguns testes, foi também testado o *dwa\_local\_planner*. Os resultados foram semelhantes.

Observa-se na Fig. 6.2, o início do funcionamento do robô. Este encontra-se na primeira sala, em posição de partida, devidamente localizado. A localização inicial depende de um ponto, fornecido pelo utilizador, através do *rviz*. A localização inicial é tão precisa quanto a definição deste ponto, podendo haver uma ligeira diferença entre o fornecido e a realidade. O estado da localização pode ser visualizado analisando quão bem as leituras do *laser\_top* se sobrepõem ao mapa. As diversas leituras são representadas por pequenos quadrados coloridos, a cor varia segundo a intensidade do feixe do laser.

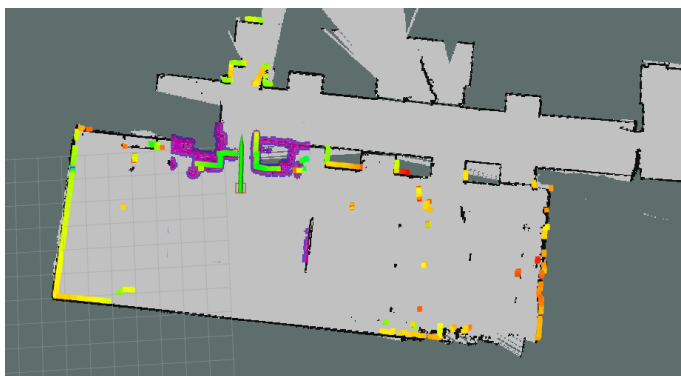


Figura 6.2: Ponto de início - primeira solução

A primeira solução provou ser capaz de cumprir os objetivos 1 e 2 ( Fig. 6.3 e Fig. 6.4 respetivamente). O robô mantém-se devidamente localizado no decorrer do percurso e não demonstra dificuldade em navegar por espaços estreitos caso esteja relativamente enquadrado. Isto deve-se particularmente à afinação dos parâmetros do mapa de custos.

Porém, após o objetivo 2, o robô entra na zona intermédia. Pode observar-se imediatamente uma discrepância na sobreposição do laser ao mapa nas paredes situadas em frente ao robô (Fig. 6.5). À medida que este continua o percurso, o erro propaga-se devido às propriedades do *laser\_scan\_matcher* (Fig. 6.6 e Fig. 6.7), não havendo possibilidade de recuperar - o erro está permanentemente presente na odometria. A localização a partir do *amcl* também depende

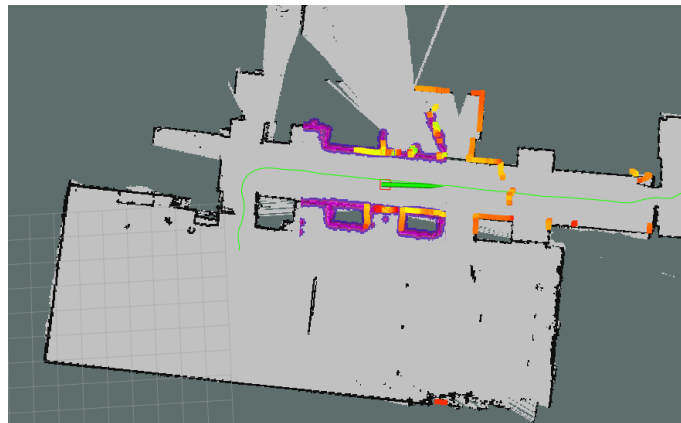


Figura 6.3: Corredor após primeira passagem estreita (porta)



Figura 6.4: Chegada ao objetivo 2

bastante da odometria, nomeadamente para estimar o seu comportamento para poder ajustar os referenciais para corrigir pequenas falhas - o erro observado provou ser demasiado grande para haver recuperação.

Eventualmente, ao continuar o seu movimento, o robô pára o seu funcionamento pois conclui que não existe uma trajetória viável para chegar ao destino. Isto acontece visto que a trajetória global permanece inalterada mas o planeador local indica que, se continuar a percorrer a trajetória indicada, ou qualquer outra calculada durante o período de simulação, irá embater contra um obstáculo. Esta situação pode ser visualizada na Fig. 6.8: as leituras do laser, detetando a parede que se encontra em frente ao robô, estão a bloquear a entrada no corredor. Do ponto de vista do robô traduz-se num obstáculo situado mesmo à entrada do corredor que bloqueia toda e qualquer passagem.

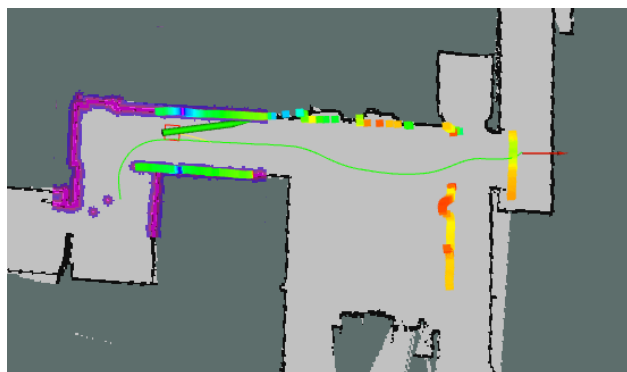


Figura 6.5: Entrada na zona intermédia - falha na localização

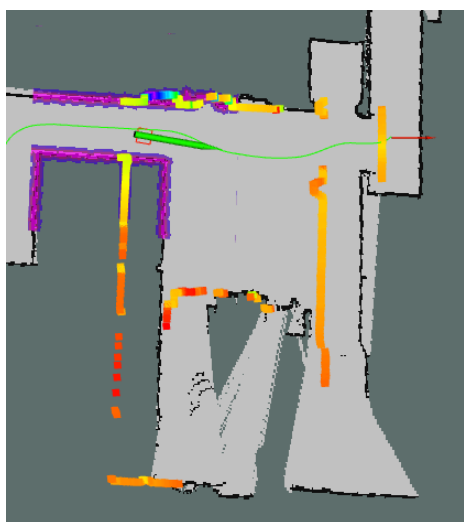


Figura 6.6: Zona intermédia - falha na localização mais evidente

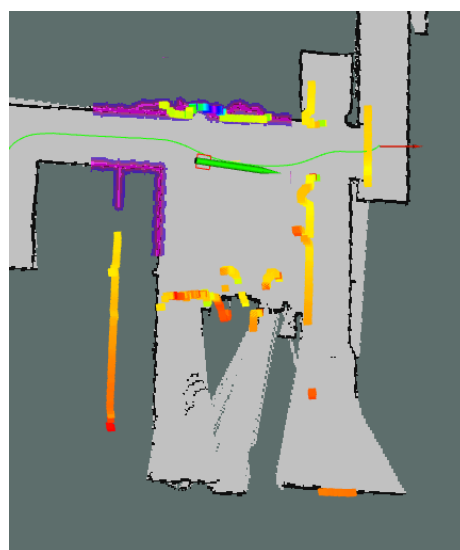


Figura 6.7: Zona intermédia - falha na localização permanente



Figura 6.8: Paragem de funcionamento da primeira solução

### 6.3 Segunda solução - Adaptação do *Localization\_Perfect\_Match*

No seguimento da primeira solução, foi adaptada a *stack Localization\_Perfect\_Match* desenvolvida por Sobreira *et al.* procurando encontrar melhores resultados. Esta *stack* já teria sido desenvolvida para o projeto original, mas não dispôs de oportunidades para ser testada no próprio. Contudo, foi testado noutras plataformas e alvo de estudo de outras dissertações tendo sido concluído que oferece vários melhoramentos à solução original.

Sendo assim, a localização era obtida através desta *stack*. Já a obtenção da odometria foi testada de duas formas diferentes, novamente com o *laser\_scan\_matcher* e com um nó de ROS desenvolvido para calcular a odometria a partir dos *encoders* das rodas (exposto em 4.2.5).

Pode observar-se o início de funcionamento do robô na Fig. 6.9. Tal como na solução anterior, este encontra-se devidamente localizado. Porém, a localização inicial na solução 2, que integra o *Localization\_Perfect\_Match*, é mais robusta e precisa, pois efetua uma correção da pose inicial, de forma a que o laser se sobreponha perfeitamente ao mapa. O robô cumpre os objetivos 1 e 2 sem dificuldades (Fig. 6.10 e Fig. 6.11) e com um movimento mais constante. Isto deve-se à qualidade da localização uma vez que, quando se trata de passagens estreitas, o robô tem um comportamento mais cauteloso e quer ter a certeza que se consegue deslocar sem embater.

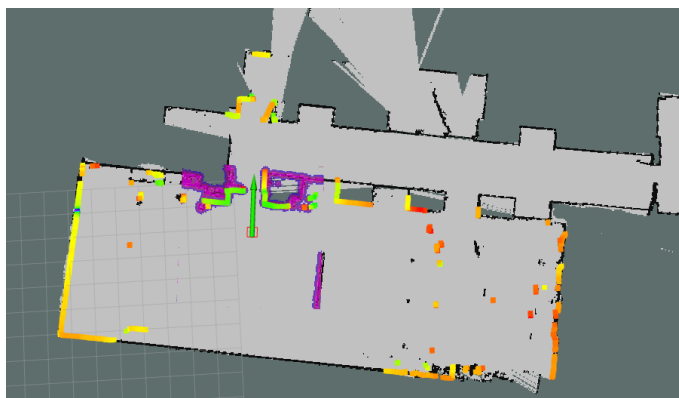


Figura 6.9: Ponto de início - solução 2

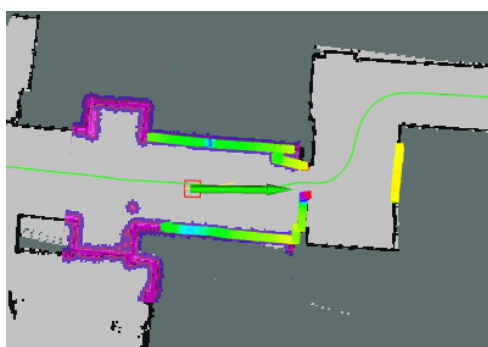


Figura 6.10: Após objetivo 1 e antes do objetivo 2

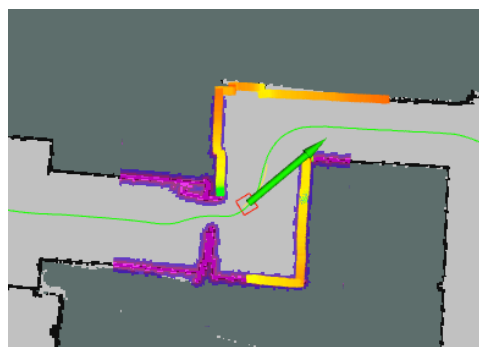


Figura 6.11: Objetivo 2 cumprido

De seguida, o robô começa o deslocamento pela zona intermédia (Fig. 6.12), desta vez, mantendo-se localizado. A discrepância na odometria existe; contudo, o *Localization\_Perfect\_Match* é capaz de aplicar as correções necessárias de forma a manter o robô localizado (Fig. 6.13 e Fig. 6.14). Aproximando-se da entrada do corredor, o robô consegue prosseguir o seu funcionamento (Fig. 6.15) e cumprir o objetivo 3 (Fig. 6.16).



Figura 6.12: Entrada na zona intermédia



Figura 6.13: Zona intermédia - robô localizado (1)

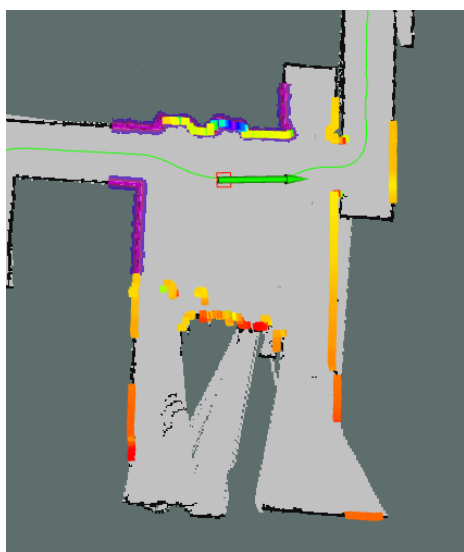


Figura 6.14: Zona intermédia - robô localizado (2)

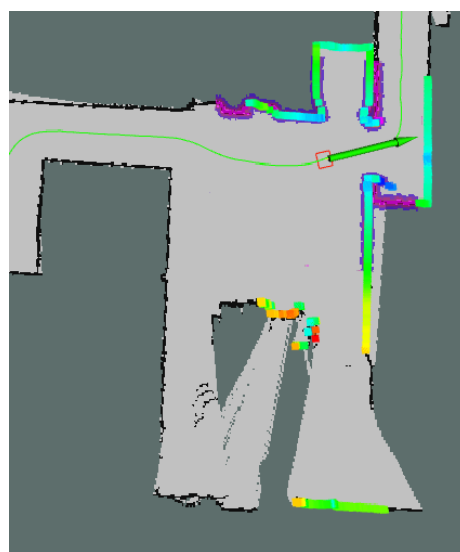


Figura 6.15: Início de entrada no corredor

Ao iniciar o percurso do corredor, pode observar-se uma possível fonte de problemas (Fig. 6.17). O laser está apenas a detetar leituras nas paredes, diretamente à esquerda e à direita do robô. A parede, que agora se encontra na retaguarda, nunca seria detetada visto que o laser tem apenas uma deteção de  $270^\circ$ . Já a parede na frente do robô encontra-se demasiado distante, levando a que as medidas obtidas para essa orientação sejam ou inexistentes ou classificadas como erradas. Este acontecimento é comum na localização de robôs através de lasers em corredores compridos

e prejudica a localização do robô, pois consegue calcular bem a sua posição entre as paredes laterais mas não tem informação quanto à sua posição no corredor, concluindo que não se está a movimentar. Este comportamento pode ser observado na Fig. 6.18, na Fig. 6.19 e na Fig. 6.20. As três figuras mostram o estado do robô, de acordo com a localização, separadas por cerca de cinco segundos entre cada uma respetivamente. A posição do robô não se alterou significativamente ainda que, na realidade, tenho quase percorrido o corredor até ao fim. Na Fig. 6.20 observa-se nitidamente uma grande diferença entre a deteção da parede em frente ao robô relativamente ao mapa - o robô está mal localizado, com um erro demasiado grande para ser retificado.

Finalmente, o robô chega muito perto do fim do corredor mas o planeador global infere que este deve continuar a movimentar-se em frente. Porém, o planeador local, recebendo a informação do laser de que a parede em frente ao robô está muito próxima (Fig. 6.21) , tenta calcular uma trajetória que lhe permita continuar o plano global. Chegando à conclusão que não é possível, tal como na primeira solução, pára o seu funcionamento. Este comportamento é comum tanto na utilização do *laser\_scan\_matcher* como do *wheel\_odometry*.

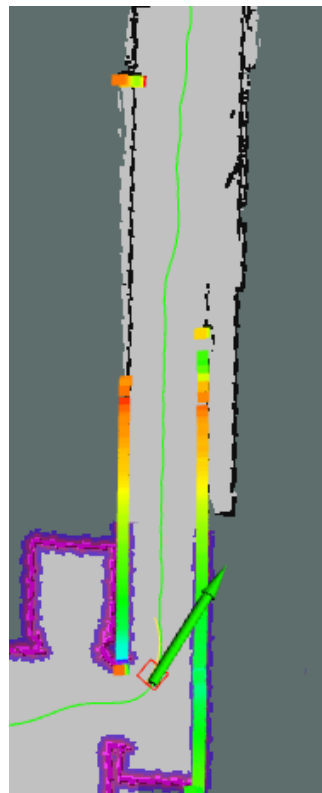


Figura 6.16: Entrada no corredor: objetivo 3 cumprido

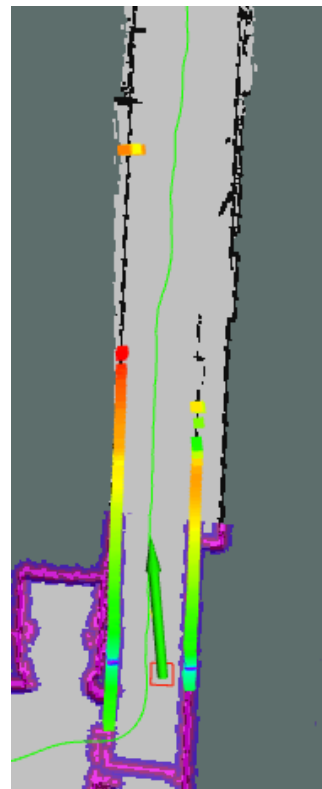


Figura 6.17: Início do corredor



Figura 6.18: Corredor - robô ligeiramente mal localizado

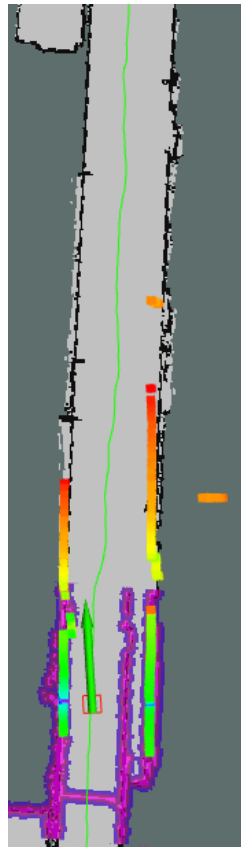


Figura 6.19: Corredor - robô mal localizado



Figura 6.20: Corredor - robô efetivamente perdido



Figura 6.21: Paragem de funcionamento da solução 2

## 6.4 Terceira solução - Implementação da fusão

Como uma abordagem final, após os bons resultados de localização do *Localization\_Perfect\_Match*, procurou-se melhorar a odometria do robô em busca do cumprimento consistente do objetivo principal. Assim sendo, foi implementada a fusão da odometria proveniente do *wheel\_odometry*

e do *laser\_scan\_matcher* através do *package robot\_localization* (exposto em 4.2.5). O principal motivo para a implementação da fusão surge do comportamento do robô no corredor. Prevê-se que, podendo contar com a odometria das rodas quando o laser está em condições de funcionamento sub-ótimas, o robô será capaz de navegar no túnel sem se perder definitivamente.

De novo, vê-se na Fig. 6.22 a posição inicial do robô. Este funciona exatamente da mesma maneira que a segunda solução, alcançando os objetivos 1,2 e 3 (Fig. 6.23, Fig. 6.24) sem qualquer problema.

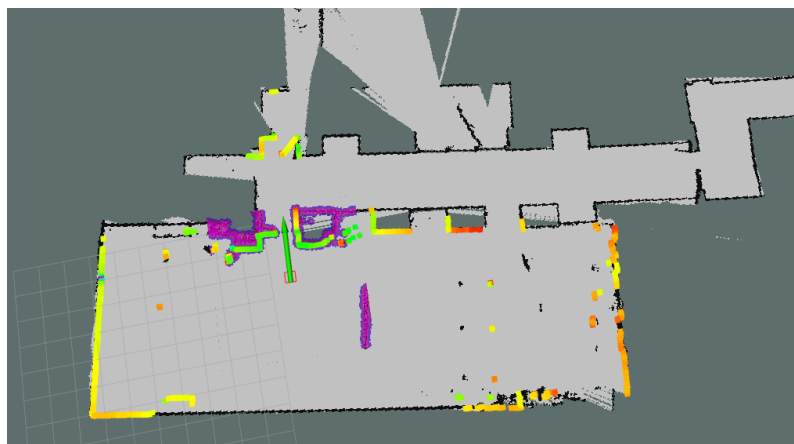


Figura 6.22: Início da terceira solução

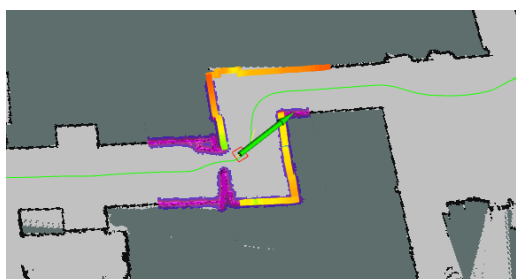


Figura 6.23: Terceira solução: objetivo 2 cumprido

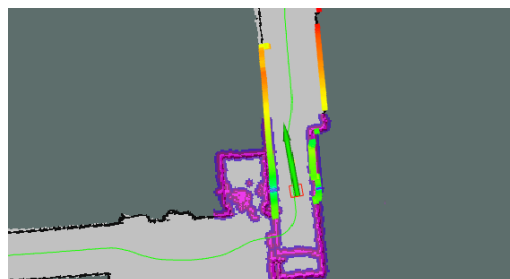


Figura 6.24: Terceira solução: objetivo 3 cumprido

O comportamento difere após o objetivo 3, *i.e.* entrar no corredor. Assim que o robô começa a percorrer o corredor, observa-se que a localização é afetada originando pequenas discrepâncias na sobreposição das leituras do laser ao mapa, bem como no mapa de custos. O estado da reflexão dos feixes do laser diretamente em frente ao robô é o elemento desencadeador - quando estes não estão a fazer uma deteção completa, a covariância associada à localização obtida pelo laser aumenta proporcionalmente ao número de feixes do laser que não detetam nenhum obstáculo. Assim sendo, o resultado da fusão tende a aproximar-se mais da odometria das rodas levando a que a localização do robô não fique presa na mesma posição, como acontecia na solução anterior. Porém, a odometria das rodas acaba por não ser perfeita e o robô, movimentando-se um pouco para um dos lados, aumenta o número de feixes que obtém medidas, diminuindo a covariância,

levando a que o robô não fique perfeitamente localizado durante todo o percurso. Comparando a localização do robô no túnel com a segunda solução (Fig. 6.18, Fig. 6.19, Fig. 6.20) à obtida pela terceira solução (Fig. 6.25) observa-se que, na terceira solução, a localização do robô acompanha o seu movimento, ainda que não perfeitamente.

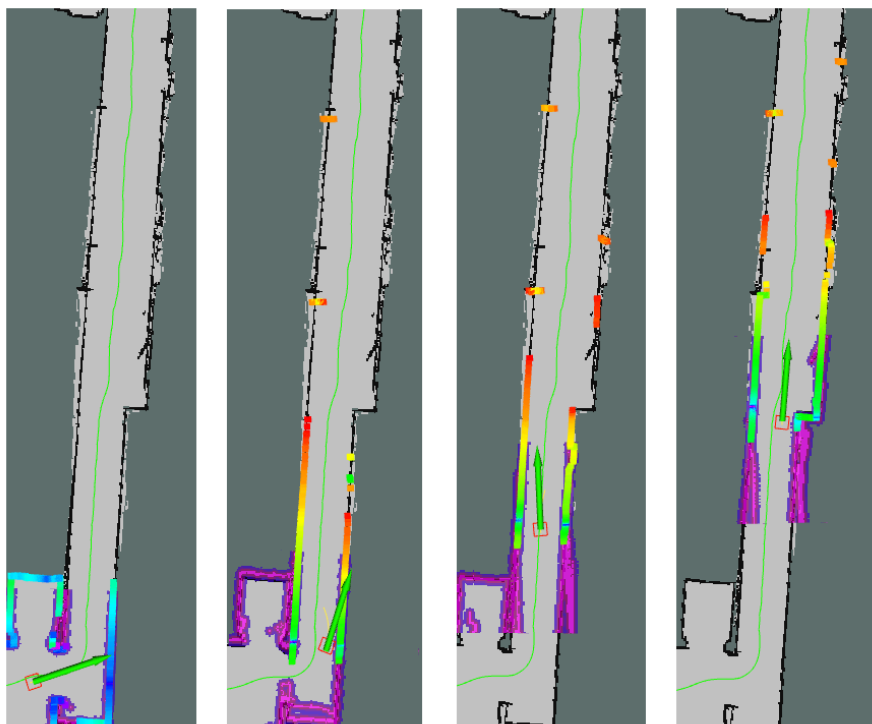


Figura 6.25: Localização do robô no corredor

O robô mantém-se então em movimento, seguindo o percurso, evitando obstáculos e mantendo-se em linha reta na maior parte do tempo, ou seja, o robô nunca pára (como se observa na solução anterior na Fig. 6.21). Durante o percurso, a localização do robô nunca fica completamente parada, salvo durante muito pequenos intervalos de tempo, levando a que este, de certa forma, se aproxime da localização correta. Após um certo patamar, o laser consegue detetar completamente todos os obstáculos e paredes no seu raio de ação, sendo capaz de fornecer uma boa localização, mas já se encontra com uma discrepância considerável. Observa-se, então, um comportamento diferente em relação à solução anterior: tentativas de recuperação.

Pode observar-se detalhadamente o processo de recuperação na Fig. 6.26. O robô chega ao ponto em que pretende virar, seguindo a trajetória do planeador global. Porém, ao detetar a parede e concluir que vai embater nesta se prosseguir com a trajetória, decide parar. Neste momento, poderia simplesmente parar o seu funcionamento, tal como anteriormente, mas como consegue detetar uma discrepância na sua localização e tem uma ligeira margem de manobra (em relação às paredes), inicia comportamentos de recuperação (Fig. 5.10). Desde a imagem 1 à imagem 10 da Fig. 6.26, o robô praticamente não se movimentou, salvo alguns solavancos da ação designada

"Aggressive Reset", porém, pode observar-se uma grande diferença na posição do robô no mapa, em relação às leituras do laser. A recuperação, essencialmente, é representada pela diferença entre estas duas imagens, sendo que na imagem 10 as leituras do laser estão perfeitamente sobrepostas ao mapa - indicando uma boa localização do robô. Pode ainda observar-se que, a partir da imagem 1, que demonstra o início da recuperação, a trajetória global é diferente, mantendo o objetivo final, embora o ponto de partida seja agora a localização atual do robô - característico das recuperações do tipo "Reset".

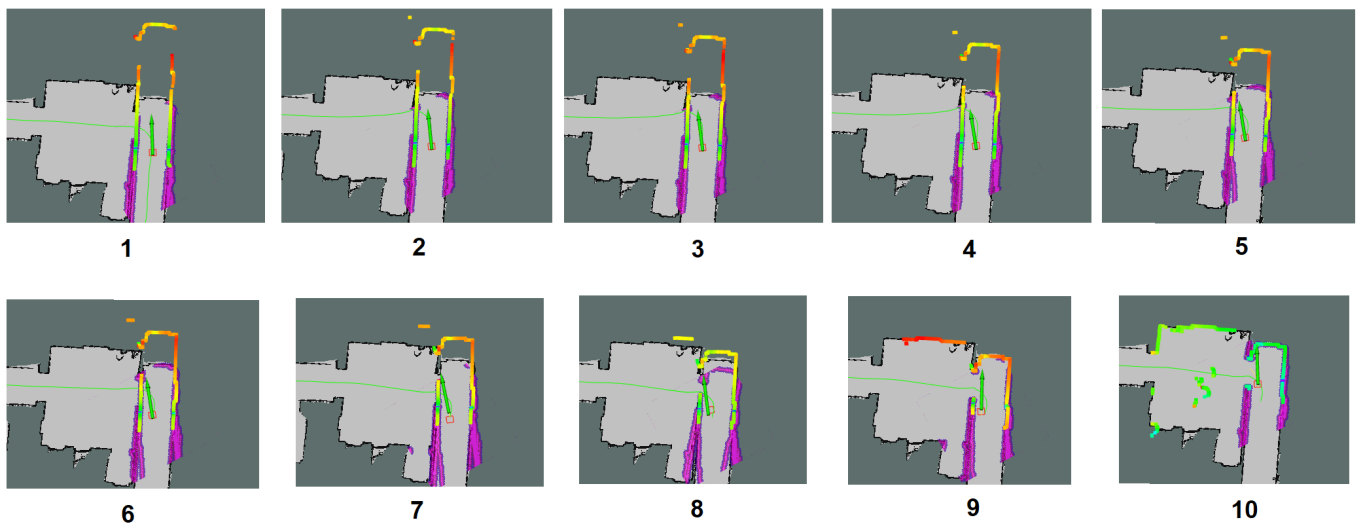


Figura 6.26: Recuperação, objetivo 4 cumprido

Após se encontrar novamente bem localizado, o robô prossegue com o percurso (Fig. 6.27), atravessando uma última passagem estreita e chegando ao destino (Fig. 6.28). Esta solução é, então, capaz de superar os objetivos 1,2,3 e 4 e alcançar o objetivo final.

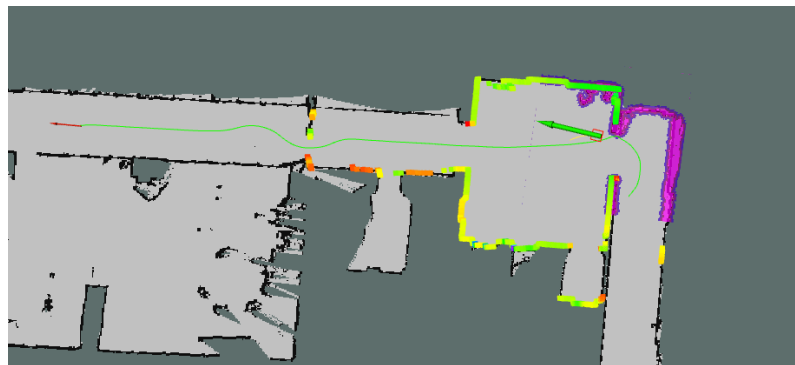


Figura 6.27: Continuação da trajetória após recuperar

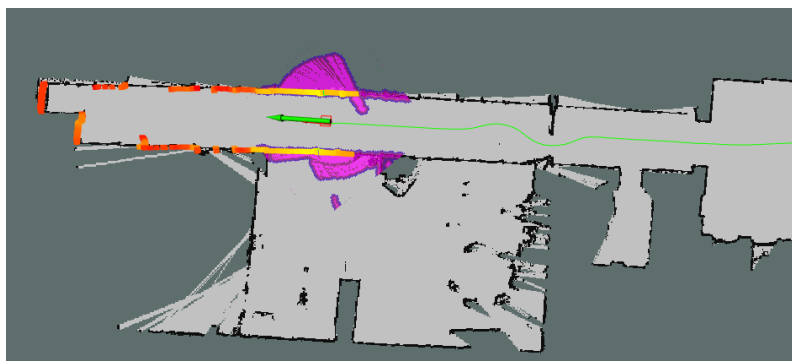


Figura 6.28: Chegada ao objetivo final

#### 6.4.1 Terceira solução - Percurso inverso

Ao atingir todos os objetivos de navegação propostos, torna-se pertinente testar a solução num percurso alternativo com o intuito de avaliar a sua robustez. Assim sendo, tomou-se como ponto de partida o objetivo final anterior (laboratório do CRIIS) e como objetivo final o ponto de partida anterior (I-108). O robô terá então de percorrer o corredor, bem como cumprir os objetivos 1,2 e 3, em sentido inverso.

O robô inicia o seu percurso e desloca-se até ao corredor sem dificuldades, como seria de esperar dada a robustez do *Localization\_Perfect\_Match*. Ao entrar no corredor o comportamento não se altera: a localização começa a acumular pequenas discrepâncias mas, graças à implementação da fusão, não permanece inalterada. Porém, ao percorrer o corredor neste sentido, o tempo total em que o robô se encontra mal localizado diminui pois o algoritmo pode iniciar uma correção da localização detetando um canto no corredor. Este canto funciona, na prática, como um marco; ao ser detetado, o algoritmo constata que nesse instante a localização não corresponde à realidade e inicia uma correção, sem alterar o funcionamento do robô. Este comportamento pode ser observado na Fig. 6.29, o canto em questão encontra-se na parede à direita do robô.

O robô Vigilante consegue prosseguir a trajetória, navegando pela zona intermédia sem se perder e atingir a zona do objetivo 2. Constata-se aqui um comportamento diferente pois, até agora, as passagens estreitas eram abordadas pela robô com uma trajetória quase perfeitamente frontal, mas no seguimento deste percurso esta passagem é atingida com uma trajetória lateral, uma vez que o movimento do robô nesta zona é em zigue-zague. Ao chegar à entrada da passagem o robô pára de se movimentar e avalia a situação, entrando em vigor um misto dos comportamentos de recuperação de localização já mencionados e comportamentos de recuperação de trajetória do *navfn*, procurando uma maneira de se movimentar pela passagem estreita, sem colisões. O robô movimenta-se lentamente e faz uma rotação sobre si mesmo até decidir reorientar-se de forma a abordar a passagem com uma trajetória mais frontal. Após o reposicionamento, prossegue com a trajetória até atingir o novo objetivo final (Fig. 6.30, trajetória global a verde).

Conclui-se, então, que a terceira solução apresenta um bom nível de robustez, mantendo o funcionamento em diversas condições, de forma a superar situações adversas, como o corredor

ou o ângulo de abordagem a passagens estreitas, e aproveitar situações mais favoráveis, como a excelente localização em ambientes fechados não extensos ou a utilização de características do ambiente como marcos para correção da localização.

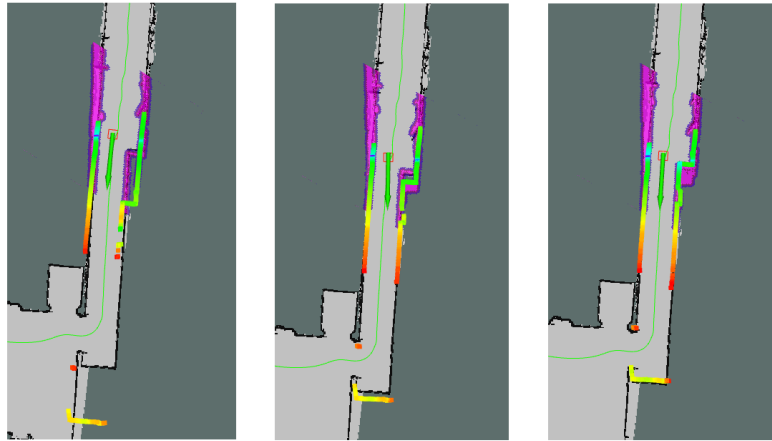


Figura 6.29: Correção no canto do corredor

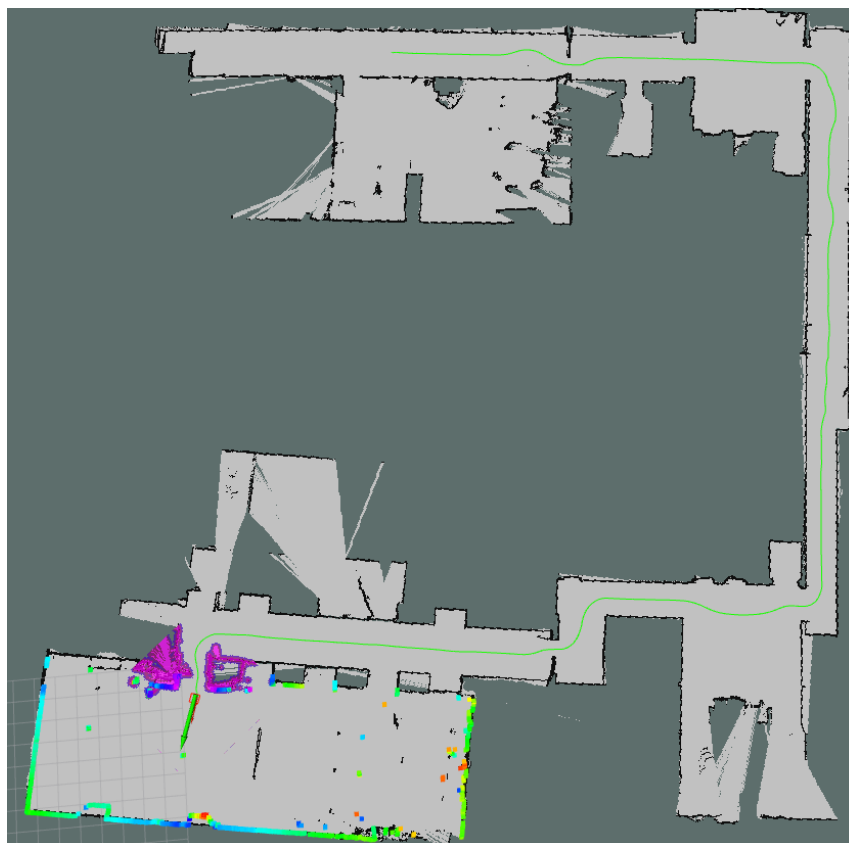


Figura 6.30: Fim do funcionamento - perspectiva global



## Capítulo 7

# Conclusões e Trabalho Futuro

### 7.1 Trabalho realizado e satisfação dos objetivos

A presente dissertação teve como primeira etapa a montagem do robô. Sendo que inicialmente apenas existia a estrutura mecânica base, os dois motores e as rodas, foi analisada a configuração do "RobVigil" bem como os requisitos que esta nova versão necessita para proceder a uma montagem simples, eficiente, bem organizada e flexível à integração de novos componentes. O resultado foi uma configuração de hardware robusta e intuitiva, apenas com os componentes fundamentais e com potencial para outros.

Após a montagem estar completa, iniciou-se o desenvolvimento de software para o robô. Sendo um dos objetivos principais um robô que se localize e navegue autonomamente baseado em ROS, a integração da *stack* de Navegação foi a solução proposta. Com este objetivo em mente, foi desenvolvido um driver que movimenta o robô de acordo com os comandos que seriam obtidos (velocidades no tópico *cmd\_vel*), fazendo a ponte entre o micro controlador e o *output* da *stack*. Ao ser possível movimentar o robô por intermédio de um tópico, foi utilizado o *package hector\_slam* para gerar mapas da área onde se pretendia trabalhar com o robô Vigilante, guiando-o manualmente através das diversas divisões (mapeou-se o piso -1 do DEEC em 2D e 3D permitindo obter toda a informação possível sobre as imediações). Após a criação dos mapas, foi explorado o *package tf* que permite definir as relações entre todos os referenciais existentes, bem como atualizá-las ao longo do tempo. O *tf* é uma peça fundamental para projetos desta natureza pois permite estabelecer as relações entre o robô e o mapa possibilitando a sua localização. Uma outra necessidade para o funcionamento do robô é uma fonte de odometria, ou seja, o quanto o robô se movimentou num dado espaço de tempo em relação a um certo ponto. Foram exploradas três soluções: *encoders* das rodas, odometria através de leituras sucessivas de um LIDAR (*laser\_scan\_matcher*) e a fusão destas duas fontes (*robot\_localization*).

Ficaram, então, cumpridos os pré-requisitos da *stack* de Navegação. Foram implementadas três grandes soluções que diferem no método de localização e/ou na fonte de odometria. A primeira implementação tinha como método de localização o método pré-definido da *stack* implementado pelo *amcl* e como fonte de odometria a estimativa obtida através do *laser\_scan\_matcher*.

Esta solução cumpria objetivos mínimos e demonstrava um bom funcionamento para casos genéricos em espaços amplos sem muitos obstáculos. Porém, em situações em que tinha de passar por espaços estreitos, provou ser inconsistente. A segunda implementação integrou a *stack Localization\_Perfect\_Match* desenvolvida por Sobreira *et al.* para o projeto original. Esta provou fornecer uma localização muito mais robusta, sendo capaz de cumprir todos os objetivos cumpridos pela primeira solução, bem como navegar por passagens estreitas consistentemente. Porém, mesmo sendo testadas duas fontes de odometria distintas - *encoders* das rodas e *laser\_scan\_matcher*, a solução não era capaz de consistentemente localizar o robô num corredor comprido. A terceira solução visou superar esse problema, implementando a técnica de fusão como fonte de odometria. Ajustando as covariâncias das duas fontes, fundiu-se o resultado da pose do robô obtido tanto pela odometria como pelo *laser\_scan\_matcher* de forma a que, quando o laser não se encontra em boas condições de funcionamento, utilize a odometria das rodas. Observou-se nos testes que esta implementação diminui a discrepância entre a posição real e o resultado da localização, tornando possível ao *Localization\_Perfect\_Match*, juntamente com comportamentos de recuperação do *move\_base*, recuperar a localização e prosseguir com a trajetória. Sendo assim, esta solução conseguiu alcançar todos os objetivos pré-definidos.

Foi também desenvolvido um nó de ROS que pára o robô caso detete um obstáculo fatal, isto é, um obstáculo dinâmico que impeça o movimento do robô, por exemplo, uma pessoa que se ponha bruscamente em frente ao robô Vigilante, enquanto este está em funcionamento. O nó subscreeve a um tópico que contém informação sobre as leituras do *laser\_front* e implementa a "zona proibida" em frente ao robô que, caso detete algo nessa zona, envia um sinal de paragem ao robô.

Finalmente, foi implementada uma funcionalidade de videovigilância através da câmara montada no robô. A imagem recebida está no formato omnidirecional e é assim transmitida através de Wi-Fi para qualquer computador que se encontre ligado à rede do robô Vigilante através de um endereço web pré-definido. Esta funcionalidade permite que um utilizador monitorize o ambiente circundante do robô remotamente.

Em suma, o robô Vigilante é capaz de se deslocar autonomamente em todos os tipos de ambientes interiores, com uma localização robusta e capaz de recuperar em caso de erro, evitando obstáculos dinâmicos (com altura superior a 125 cm) ou parando o seu movimento caso seja detetado um obstáculo fatal. A adaptação a outros ambientes é facilitada pela integração dos *packages* geradores de mapas 2D e 3D. O robô transmite ainda vídeo em tempo real, 360° à sua volta, acessível em qualquer plataforma.

## 7.2 Trabalho futuro

Por forma a dar continuidade ao trabalho efetuado, seria pertinente testar o robô Vigilante noutros ambientes para aferir a robustez da arquitetura implementada. Pode ainda ser interessante adaptar o funcionamento do robô de forma a funcionar com SLAM, eliminando assim a necessidade de um mapa conhecido, sendo que a *stack* de navegação já tem suporte para este modo de

funcionamento. Prevê-se, contudo, um aumento na memória utilizada (dependendo do tamanho do ambiente em que for testado) e erros na localização devido a falhas na reflexão dos feixes do laser provenientes de movimentos erráticos (pode ser controlado de forma a que seja uma ocorrência muito rara) ou fatores externos.

Adicionalmente, o *hector\_slam* é reconhecido por fornecer uma localização muito precisa, porém apenas se localiza no mapa gerado (SLAM). Foi posta a hipótese de alterar o código fonte do *package*, tendo sido chegado à conclusão que seria de facto possível (corroborado pelo autor, Stefan Kohlbrecher) utilizando o *package map\_server* para carregar o mapa no qual se pretende obter a localização e introduzi-lo na definição da estrutura de dados dedicada ao mapa. Isto seria idealmente incorporado na função "mapPubContainer" (linha 137 do ficheiro *HectorMappingRos.cpp*).

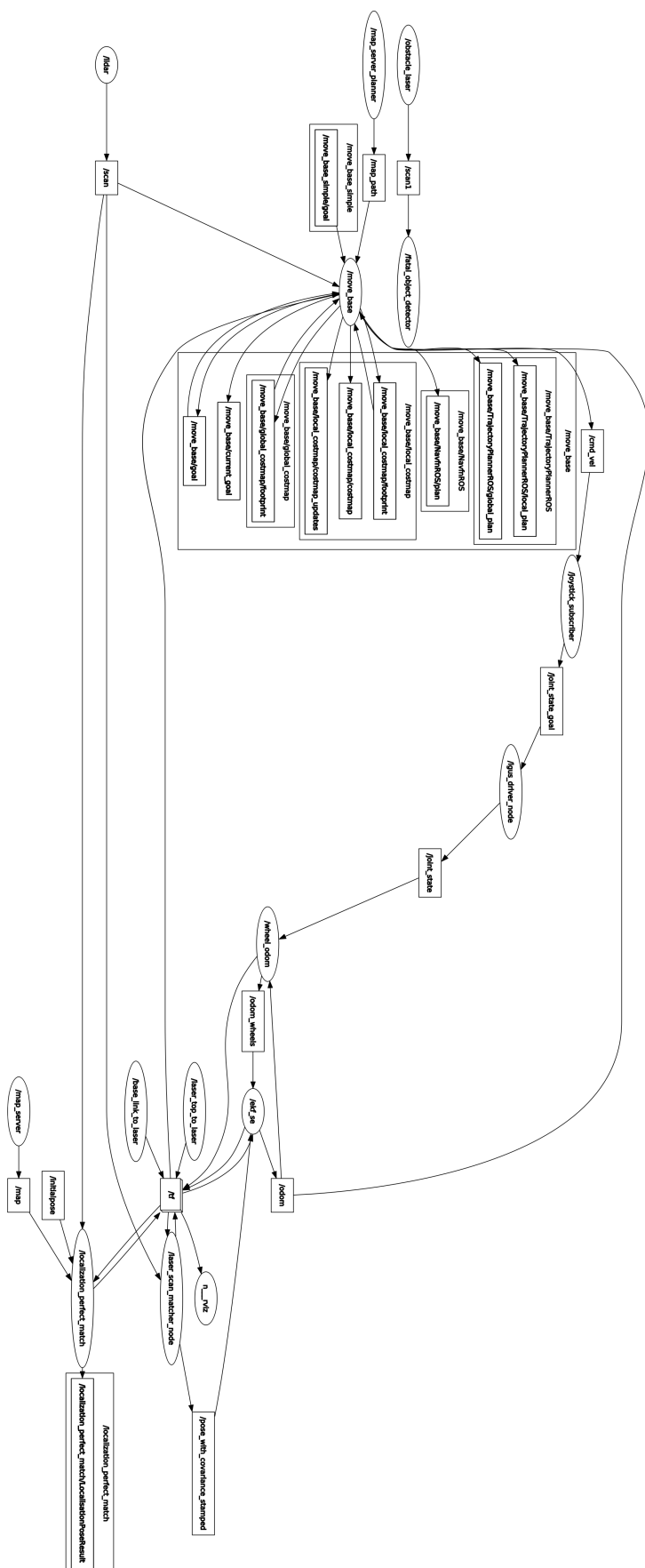
No tópico da vigilância, é possível incorporar diversos sensores no robô, como por exemplo os que contavam no projeto original (detetor de fumo, humidade, monóxido de carbono, entre outros). A configuração do hardware permite uma muito simples integração destes, havendo já espaço na carapaça exterior bem como entradas vazias no driver para a ligação com o arduino MEGA. Após a montagem, basta apenas criar uma rotina que permita ao micro-controlador receber informação do(s) sensor(es) e enviá-la para o PC, onde necessita, também, de ser alterada a leitura de dados para ter em conta a informação adicional. Estas informações poderiam, então, ser transmitidas a um utilizador fazendo com que este consiga monitorizar os respetivos valores.

Finalmente, ainda no mesmo tópico mas tendo em conta a usabilidade, a criação de uma interface gráfica para qualquer plataforma seria uma mais valia. Neste momento, para utilizar o robô Vigilante, é necessário um computador com o ROS. Porém, existem protocolos, nomeadamente o ZeroMQ, que permitem a transmissão de dados no ambiente do ROS para outros sistemas operativos. Assim sendo, seria possível desenvolver uma GUI flexível a qualquer computador que conseguiria receber informação do robô (dados dos sensores, stream da câmara, localização num mapa), bem como controlá-lo. Seria pertinente incorporar na interface a transformação de imagem omnidirecional para panorâmica, pois ao transformar já no destino, diminui-se a latência do vídeo e torna-se possível utilizar qualquer ferramenta a não depender do ROS.



## **Anexo A**

### **Gráfico dos nós e tópicos de ROS durante o funcionamento do robô**



# Referências

- [1] A Paulo Moreira et al. Apontamentos da disciplina SRAU - Sistemas Roboticos Autonomos do Mestrado Integrado em Engenharia Eletrotecnica da FEUP 2017-2018. <https://drive.google.com/drive/folders/1A1rBluuiqeqVxKdk5rTO2iAOYkweGpc4?usp=sharing>, 2017-2018.
- [2] Pedro Luís Cerqueira Gomes da Costa et al. Planeamento cooperativo de tarefas e trajectórias em múltiplos robôs. *none*, 2011.
- [3] Oliver Ploder. *SecUAV - A Unified Testbed for the Evaluation of Secure State Estimators*. PhD thesis, UCLA, 07 2018.
- [4] Carlos Jaramillo. *Single-Image Omnidirectional Vision Systems*. PhD thesis, City College, 2016.
- [5] Vcodex Limited. An Overview of H.264 Advanced Video Coding. <https://www.vcodex.com/an-overview-of-h264-advanced-video-coding/>, 2007-2019.
- [6] Michael Ferguson David V. Lu and Aaron Hoy. Wiki: global\_planner. [http://http://wiki.ros.org/global\\_planner](http://http://wiki.ros.org/global_planner), 2018.
- [7] Kaiyu Zheng. ROS Navigation Tuning Guide. [http://kaiyuzheng.me/documents/papers/ros\\_navguide.pdf](http://kaiyuzheng.me/documents/papers/ros_navguide.pdf), 2016.
- [8] CRIIS. RobVigil. <http://criis.inesctec.pt/index.php/criis-projects/robvigil/>, 2011.
- [9] Johann Borenstein and Liqiang Feng. Measurement and correction of systematic odometry errors in mobile robots. *IEEE Transactions on robotics and automation*, 12(6):869–880, 1996.
- [10] Fantian Kong, Youping Chen, Jingming Xie, Gang Zhang, and Zude Zhou. Mobile robot localization based on extended kalman filter. In *Intelligent Control and Automation, 2006. WCICA 2006. The Sixth World Congress on*, volume 2, pages 9242–9246. IEEE, 2006.
- [11] David Tedaldi, Alberto Pretto, and Emanuele Menegatti. A robust and easy to implement method for imu calibration without external equipments. In *Robotics and Automation (ICRA), 2014 IEEE International Conference on*, pages 3042–3049. IEEE, 2014.
- [12] Jason M O’Kane. Global localization using odometry. In *Robotics and Automation, 2006. ICRA 2006. Proceedings 2006 IEEE International Conference on*, pages 37–42. IEEE, 2006.
- [13] John J Leonard and Hugh F Durrant-Whyte. Mobile robot localization by tracking geometric beacons. *IEEE Transactions on robotics and Automation*, 7(3):376–382, 1991.

- [14] Puneet Goel, Stergios I Roulmeliotis, and Gaurav S Sukhatme. Robust localization using relative and absolute position estimates. In *Intelligent Robots and Systems, 1999. IROS'99. Proceedings. 1999 IEEE/RSJ International Conference on*, volume 2, pages 1134–1140. IEEE, 1999.
- [15] Miguel Pinto, António Paulo Moreira, and Anibal Matos. Localization of mobile robots using an extended kalman filter in a lego nxt. *IEEE Transactions on Education*, 55(1):135–144, 2012.
- [16] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Markov localization for mobile robots in dynamic environments. *Journal of artificial intelligence research*, 11:391–427, 1999.
- [17] Manuel Carlos Monteiro Gouveia et al. Estudo e implementação de um algoritmo de localização baseado na correspondência de mapas. *None*, 2009.
- [18] Martin Lauer, Sascha Lange, and Martin Riedmiller. Calculating the perfect match: an efficient and accurate approach for robot self-localization. In *Robot Soccer World Cup*, pages 142–153. Springer, 2005.
- [19] Tim Bailey, Juan Nieto, Jose Guivant, Michael Stevens, and Eduardo Nebot. Consistency of the ekf-slam algorithm. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, pages 3562–3568. IEEE, 2006.
- [20] Montemerlo Michael, Thrun Sebastian, Koller Daphne, and Wegbreit Ben. Fastslam 2.0: An improved particle filtering algorithm for simultaneous localization and mapping that provably converges. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI)*, volume 2, 2003.
- [21] Dieter Fox, Sebastian Thrun, Wolfram Burgard, and Frank Dellaert. Particle filters for mobile robot localization. In *Sequential Monte Carlo methods in practice*, pages 401–428. Springer, 2001.
- [22] M Cummins. Probabilistic localization and mapping in appearance space. 2009, balliol college, 2009.
- [23] Calyampudi Radhakrishna Rao. Rao-Blackwell Theorem. [http://www.scholarpedia.org/article/Rao-Blackwell\\_theorem](http://www.scholarpedia.org/article/Rao-Blackwell_theorem), 2011.
- [24] Charles W Warren. Global path planning using artificial potential fields. In *Robotics and Automation, 1989. Proceedings., 1989 IEEE International Conference on*, pages 316–321. IEEE, 1989.
- [25] Amit Patel Stanford University. Introduction to A\*. <http://theory.stanford.edu/~amitp/GameProgramming/AStarComparison.html>, 1997-2019.
- [26] Yoram Koren and Johann Borenstein. Potential field methods and their inherent limitations for mobile robot navigation. In *Robotics and Automation, 1991. Proceedings., 1991 IEEE International Conference on*, pages 1398–1404. IEEE, 1991.
- [27] A Paulo Moreira, Paulo Costa, and Pedro Costa. Real-time path planning using a modified a\* algorithm. In *Proceedings of ROBOTICA 2009-9th Conference on Mobile Robots and Competitions*, 2009.

- [28] Tiago Pereira Do Nascimento, Pedro Costa, Paulo G Costa, António Paulo Moreira, and André Gustavo Scolari Conceição. A set of novel modifications to improve algorithms from the a\* family applied in mobile robotics. *Journal of the Brazilian Computer Society*, 19(2): 167–179, 2013.
- [29] Amanda Dattalo. Wiki: ROS/Introduction. <http://wiki.ros.org/ROS/Introduction>, 2018.
- [30] T. Moore and D. Stouch. A generalized extended kalman filter implementation for the robot operating system. In *Proceedings of the 13th International Conference on Intelligent Autonomous Systems (IAS-13)*. Springer, July 2014.
- [31] Matthias Gruhler. Wiki: navigation. <http://wiki.ros.org/navigation>, 2017.
- [32] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*. IEEE, November 2011.
- [33] Giorgio Grisetti, Cyrill Stachniss, and Wolfram Burgard. Improved techniques for grid mapping with rao-blackwellized particle filters. *IEEE transactions on Robotics*, 23(1):34–46, 2007.
- [34] Pedro Lima, Andrea Bonarini, Carlos Machado, Fabio Marchese, Carlos Marques, Fernando Ribeiro, and Domenico Sorrenti. Omni-directional catadioptric vision for soccer robots. *Robotics and Autonomous Systems*, 36(2-3):87–102, 2001.
- [35] Venkata N Peri and Shree K Nayar. Generation of perspective and panoramic video from omnidirectional video. In *Proc. DARPA Image Understanding Workshop*, volume 1, pages 243–245. Citeseer, 1997.
- [36] Gary J Sullivan, Pankaj N Topiwala, and Ajay Luthra. The h. 264/avc advanced video coding standard: Overview and introduction to the fidelity range extensions. In *Applications of Digital Image Processing XXVII*, volume 5558, pages 454–475. International Society for Optics and Photonics, 2004.
- [37] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. doi: 10.1007/s10514-012-9321-0. URL <http://octomap.github.com>. Software available at <http://octomap.github.com>.
- [38] Andrea Censi. An ICP variant using a point-to-line metric. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Pasadena, CA, May 2008. doi: 10.1109/ROBOT.2008.4543181. URL <http://purl.org/censi/2007/plicp>.
- [39] Kaiyu Zheng. Basic Navigation Tuning Guide. <http://wiki.ros.org/navigation/Tutorials/Navigation%20Tuning%20Guide>, 2018.