UNIVERSIDADE DE COIMBRA
FACULDADE DE CIÊNCIAS E TECNOLOGIA
DEPARTAMENTO DE ENGENHARIA
ELECTROTÉCNICA E DE COMPUTADORES

Odometria Visual usando imagens RGB-D

Luís António Pires Dias

Fevereiro de 2013

Odometria Visual usando imagens RGB-D

Orientador:
Prof. Dr. Hélder de Jesus Araújo

Dissertação submetida para obtenção do grau de Mestre em Engenharia Electrotécnica e de Computadores

Júri:

Prof. Dr. João Pedro de Almeida Barreto (Presidente)
Prof. Dr. Rui Paulo Pinto da Rocha

Luís António Pires Dias

Fevereiro de 2013

"The important thing is not to stop questioning; curiosity has its own reason for existing."

-Albert Einstein.

Agradecimentos

Em primeiro lugar, queria agradecer ao meu orientador Professor Doutor Hélder de Jesus Araújo pela orientação, pela grande disponiblidade, pelos conhecimentos transmitidos e pelas correções que foi sugerindo ao longo do trabalho.

Em segundo lugar, queria agradecer a todos os meus colegas e amigos, que ajudaram-me sempre nos bons e maus momentos, permitindo que esta dissertação fosse possível. Todos vão ficar na minha memória para sempre.

Agradeço ao aluno de doutoramento Luís Carlos Artur da Silva Garrote pela ajuda prestada nos testes práticos indoor.

Agradeço também à minha família, em especial aos meus pais e irmão, que sempre me apoiaram ao longo do curso.

Por fim, agradeço ainda à instituição Universidade de Coimbra que permitiu a minha formação, e disponibilizou as suas instalações para a realização desta dissertação.

Resumo

Neste trabalho, um dos problema a resolver é a localização de um robô móvel num espaço desconhecido e estruturado. Este problema pode ser resolvido por um sistema que realize a Odometria Visual.

A Odometria Visual é uma área da visão por computador que pode ser aplicada à robótica móvel, e tem como principal objectivo estimar a pose de um agente (por exemplo um robô móvel) através do uso de câmaras. Além disso, surge como uma alternativa à odometria usual (por encoders) e não depende das características das rodas e irregularidades do piso.

Nesta dissertação é apresentado um algoritmo de Odometria Visual e efetuada uma avaliação do seu desempenho.

O algoritmo em questão denomina-se RGB-D SLAM, está implementado no sistema ROS e requer o uso de câmaras RGB-D.

Uma plataforma experimental constituída por um robô móvel e um sensor Kinect é utilizada para executar diferentes percursos experimentais em ambientes interiores, com a finalidade de avaliar o desempenho do algoritmo.

O algoritmo é avaliado de acordo com o erro da trajectória, obtido entre a trajectória realmente percorrida pela plataforma (*ground truth*) e a trajectória estimada. Para analisar este erro mais detalhadamente foram criadas métricas de avaliação de erro em Matlab.

As conclusões do trabalho são retiradas com base nos resultados dos percursos, desta forma é possível saber quais os percursos onde o algoritmo apresenta o melhor desempenho.

Além disto, são descritos os componentes da plataforma e também todo o processo da Odometria Visual.

Palavras-Chave: Odometria Visual, percurso, trajectória, RGB-D SLAM, robótica móvel, câmaras RGB-D, detecção de *features*, métricas de avaliação de erro, estimação da pose

Abstract

In this work, one of the problems to solve is the localization of the mobile robot in an unknown and structured space. This problem can be solve by a system with realizes the Visual Odometry.

The Visual Odometry is an area of computer vision which can be applied to mobile robotics, and it has got as main objective to estimate the pose of an agent (for example a mobile robot) through the use of cameras. Besides that, it emerges as an alternative to conventional odometry (by encoders) and that doesn't depend from the characteristics of the wheels and irregularities of the floor.

In this thesis it is presents an alghorithm of Visual Odometry. At the same time it is realized an evaluation of its performance.

The alghorithm, in evidence, is called RGB-D SLAM. This alghoritm is implemented in ROS and requires the use of RGB-D cameras.

An experimental platform consists of a mobile robot and a Kinect sensor which is used to perform different experimental paths in indoor environments, with the purpose of evaluating the performance of the algorithm.

The algorithm is evaluated according to the error of trajectory, obtained between the trajectory actually traveled by the platform (ground truth) and the estimated trajectory. To analyze this error with more details, were created error metrics for evaluation in Matlab.

The conclusions of this work are reported based on the results of paths, so it is possible to know which paths the algorithm presents the best performance.

Moreover, are described the platform components and also all the process of Visual Odometry.

Keywords: Visual Odometry, path, trajectory, RGB-D SLAM, mobile robotics, RGB-D cameras, feature detection, error metrics for evaluation, pose estimation

Conteúdo

Lista de Figuras

Lista de Tabelas

Lista de Símbolos e Abreviaturas

ϕ	Ângulo de rotação em torno do eixo X da câmara
ψ	Ângulo de rotação em torno do eixo Z da câmara
θ	Ângulo de rotação em torno do eixo Y da câmara
T_x	Translação segundo o eixo X da câmara
T_y	Translação segundo o eixo Y da câmara
T_z	Translação segundo o eixo Z da câmara
2D	Bidimensional
3D	Tridimensional
A-h	Ampere-hora
Aria	Advanced Robotics Interface for Applications
BSD	Berkeley Software Distribution
cm	centímetros
D	Profundidade (Depth)
DC	Corrente Contínua (Direct Current)
DOF	Graus de Liberdade (Degrees of Freedom)
DOG	Diferença das Gaussianas (Difference of Gaussian)
EKF	filtro de Kalman estendido (extended Kalman filter)
FPS	Frames per Second
GPS	Global Positioning System
GUI	Interface gráfico do utilizador (Graphical User Interface)
ICP	Iterative Closest Point
IMU	Inertial Measurement Unit
ISR	Instituto de Sistemas e Robótica
IV	Infravermelha
kg	quilograma
km	quilómetro
LIDAR	Light Detection And Ranging

LMA	Levenberg–Marquardt algorithm
LTS	Long Term Support
m	metros
mm	milímetros
ms	milisegundos
mse	erro quadrático médio (mean square error)
OpenCV	Open Source Computer Vision
OV	Odometria Visual (Visual Odometry)
PCL	Point Clouds Library
rad	radianos
RANSAC	Random Sample Consensus
RGB	Vermelho-Verde-Azul (Red-Green-Blue)
rms	valor quadrático médio (root mean square)
rmse	raiz quadrada do erro quadrático médio (root mean square error)
ROS	Robot Operating System
SDK	Software Development Kit
seg	segundos
SFM	Structure From Motion
SIFT	Scale Invariant Feature Transform
SLAM	Localização e mapeamento simultâneos (Simultaneous Localization and Mapping)
SURF	Speeded Up Robust Features
SVD	Decomposição em valores singulares (Singular Value Decomposition)
SVN	Subversion
UC	Universidade de Coimbra
USB	Universal Serial Bus
UXGA	Ultra Extended Graphics Array
V	Volt
VGA	Video Graphics Array
W	Watt

1 Introdução

Actualmente, a Robótica Móvel é uma área em grande evolução, sendo um dos tópicos com maior destaque na investigação por parte de diversas universidades e instituições. Esta área tem dado preferencialmente ênfase ao desenvolvimento e investigação de metodologias nos domínios da localização e navegação de robôs móveis autónomos, destacando os resultados obtidos em [61] e [63].

Geralmente sistemas robóticos capazes de se movimentar autonomamente num ambiente desconhecido precisam de sensores tais como: laser, encoders, câmaras, sonares, GPS (*Global Positioning System*), sensores inerciais IMU (*lnertial Measurement Unit*) e radares. Neste trabalho o único sensor usado é uma câmara para a captura de imagens do ambiente. Ela vai permitir estimar a localização do robô, ou seja, determinar a sua posição e orientação.

O interesse pela localização do robô num espaço 3D é importante, devido ao facto de permitir otimizar o percurso efetuado, evitar a colisão com obstáculos e também melhorar os algoritmos de navegação.

Um dos métodos usados para estimação da localização é a odometria usual, isto é, uso de encoders incorporados nos eixos de rotação das rodas. O processo básico deste método é a integração consecutiva da informação do movimento em um dado período de tempo, o que leva a uma inevitável acumulação de erros na estimação da localização. Este problema é originado devido às características das rodas e às irregularidades do piso, tais como: poeira, areia, lama e outras condições adversas. Basta uma ligeira elevação no caminho para introduzir uma perturbação no sistema. De modo que, ele pensa que andou mais em (x,y,teta) do que na realidade andou. Estas perturbações acabam por ser acumuladas no processo de integração.

Outro dos métodos é a Odometria Visual (OV), que surge como alternativa à odometria usual. Este método usa a visão por computador para observar o meio envolvente, através do uso de câmaras montadas rigidamente ao robô, com o objectivo de medir de forma precisa a localização. Portanto é possível estimar a localização mesmo quando acontece derrapagens ou rotações bruscas do robô, visto que este método captura dados que não são sensíveis a este tipo de adversidades.

Em sistemas com OV, o ambiente ao redor do robô é a preocupação principal. Este precisa de ter iluminação suficiente e permanecer estático ou com uma dinâmica muito lenta em relação ao robô. Além disso, tem de ser estruturado, ou seja, os objectos devem ter posições definidas. Deste modo, é possível capturar imagens com um grande número de *features* visuais sem erros significativos.

Em ambientes dinâmicos é impensável construir sistemas com OV, porque os objectos que fazem parte do local podem mudar inesperadamente a sua posição e orientação independentemente do robô. Assim, as imagens capturadas não são confiáveis e o processo de extracção de *features* irá conter erros significativos.

A estimação da localização do robô requer o uso de um algoritmo que execute todo o processo da OV. Em geral, estes algoritmos são implementados na tentativa de serem robustos, rápidos

e eficazes. Porém, na realidade não o são. Os algoritmos da odometria usuais demonstram ter melhor desempenho nestas particularidades.

1.1 Objectivo do trabalho

O objectivo deste trabalho é apresentar um sistema de OV que possa estimar a pose (posição e orientação) de um robô móvel através do uso de um sistema de visão acoplado a ele. Neste trabalho foi proposto o uso do dispositivo Microsoft® Kinect™, porém podia ser usado outro tipo de câmara que tivesse visão estéreo. O sensor Kinect é usado para capturar imagens RGB-D em tempo real e também para executar um algoritmo de OV, com o objectivo de poder estimar a pose da câmara ao longo do tempo. Atendendo a que a câmara está ligada rigidamente ao robô (colocada no topo deste), então posso salientar que a pose obtida da câmara é aproximada à pose do robô, pois, a diferença entre ambos resume-se numa translação e rotação.

Outro dos objectivos propostos é a avaliação do desempenho do algoritmo para diferentes percursos experimentais.

1.2 Estado da arte

Nesta secção é feita uma apresentação de vários trabalhos propostos para realizar a OV, com as mais diversas abordagens. Deu-se ênfase a abordagens que implementam a OV com base em visão monocular e visão estéreo, duas temáticas diferentes. Além disso, são apresentados trabalhos de investigação inicial sobre OV.

O problema de recuperar as poses relativas da câmara e a sua estrutura 3D a partir de uma um conjunto de imagens capturadas é conhecido na comunidade de Visão por Computador como *structure from motion*[1] (SFM)[19]. A OV é um caso particular do SFM. Este algoritmo é mais geral e aborda o problema da reconstrução 3D, tanto da estrutura como das poses da câmara com base em um conjunto de imagens sequenciais ordenadas ou não. No trabalho [24] a estrutura final e as poses da câmara são aperfeiçoadas com um algoritmo de otimização *offline* - *bundle adjustment* [62], cujo tempo de computação cresce com o número de imagens. De modo recíproco, a OV concentra-se em estimar em tempo real o movimento 3D da câmara sequencialmente à medida que novas *frames* são capturadas. O algoritmo *bundle adjustment* pode ser usado para corrigir a trajectória local estimada.

O problema de estimar a pose de um agente a partir somente de câmaras começou em 1980 e foi descrito por Moravec em [45]. A maior parte da investigação inicial sobre OV pode ser encontrada em [45],[42],[43],[36] e [49]. Estes trabalhos foram efetuados com *Mars rovers*[2], e foram motivados pelo programa de exploração a Marte pela NASA, na tentativa de permitir que estes adquiram a capacidade de estimar os seus 6 graus de liberdade (DOF, do inglês *Degrees of Freedom*) do movimento em terrenos irregulares. O trabalho de Moravec destaca-se por apresentar o primeiro diagrama de blocos para estimação do movimento, e também por descrever um dos primeiros detectores de cantos (depois do primeiro proposto por Hannah em 1974 [27]) que hoje em dia é conhecido por detector de cantos de Moravec [28], um antecessor do proposto por Forstner [23] e Harris e Stephens [30], [29].

[1] *structure from motion* é um algoritmo para estimar a estrutura tridimensional de um objeto através da análise do movimento local ao longo do tempo.

[2] *Mars rover* é um veículo motorizado automatizado capaz de navegar sobre a superfície do planeta Marte, após a sua aterragem.

As câmaras usadas neste tipo de sistemas possuem visão estéreo ou visão monocular e ao longo dos anos estas duas temáticas quase progrediram como duas linhas independentes de pesquisa. No entanto a maior parte da investigação em OV foi realizada com câmaras estéreo. De seguida, são apresentados alguns trabalhos relevantes propostos neste domínio.

- [42] Matthies e [43] Shafter basearam-se no trabalho de Moravec. Eles usaram um sistema binocular e aproveitaram o procedimento do Moravec para detectar e seguir cantos. Em vez de usarem uma representação escalar da incerteza como fez Movarek, eles aproveitaram o erro da matriz das covariâncias das *features* trianguladas e incorporaram-no na etapa de estimação do movimento. Em comparação ao trabalho de Movarek, eles demonstraram resultados melhores na recuperação da trajectória para um robô planetário, com 2% de erro relativo num percurso de 5.5 m.
- [49], [50] Olson *et al.* mais tarde expandiram o trabalho anterior com a introdução de um sensor de orientação absoluto (por exemplo uma câmara omnidirecional) e usaram o detector de cantos Forstner, que tem um desempenho computacional superior em relação ao operador de Moravec. Quando um sensor de orientação é incorporado, o crescimento do erro pode ser reduzido a uma função linear da distância percorrida. Este facto levou a um erro da posição relativa de 1.2% num percurso de 20 m.
- [36] Lacroix *et al.* implementaram outra abordagem para os robôs planetários. Uma proposta para um sistema de OV com visão estéreo semelhante aos trabalhos explicados anteriormente. A diferença encontra-se na seleção dos pontos chave. Em vez de usarem o detector Forstner, eles usaram uma visão estéreo compacta e depois selecionam um candidato a ponto chave por análise da função de correção em torno dos seus picos - uma abordagem explorada em [44] e [32] e em outros trabalhos.
- [44] Milella e Siegwart propuseram uma abordagem para estimar o movimento e remover os *outliers* para todos os rover todo o terreno. Eles usaram a abordagem de Shi-Tomasi [56] para detecção de cantos, e basearam-se no trabalho de Lacroix, mantendo os pontos que tinham grande confiança no mapa de disparidade estéreo. A estimação do movimento foi resolvido usando o método dos mínimos quadrados, e depois usaram o *iterative closest point*[3] (ICP)[55]. Para maior robustez, uma etapa para remover os *outliers* foi incorporada no ICP.
- Em 2004 [48] Nister *et al.* fizeram uma análise completamente diferente dos outros trabalhos. O seu trabalho ficou conhecido não só por introduzir o termo OV, mas também por propor a primeira implementação em tempo real e a longo prazo com um robusto esquema de rejeição de *outliers*. Eles melhoraram as implementações primitivas em várias âmbitos e incorporaram o RANSAC (do inglês *Random Sample Consensus*) [22] na etapa de estimação de movimento para remover os *outliers*.
- *[17]* Comport *et al.* introduziram um método diferente de estimação de movimento. Em vez de usarem ténicas 3D-para-3D no registo de pontos ou 3D-para-2D na estimação da pose da câmara, eles basearam-se no tensor quadrifocal, o que permite o cálculo do movimento a partir de correspondências entre imagens 2D-para-2D sem ter que triangular pontos 3D em qualquer um dos pares estéreo.

Na última década foram obtidos resultados satisfatórios com a utilização de uma única câmara ao longo de grandes distâncias (até vários quilómetros) utilizando tanto câmaras de perspectiva como omnidirecionais. Alguns exemplos destes trabalhos estão em: [18], [38], [26] e [60]. Trabalhos relacionados podem ser divididos em três categorias: métodos *feature-based*,

[3] *iterative closest point* é um algoritmo muito popular para correção da pose.

métodos *appearance-based*, e os métodos híbridos. Os métodos *feature-based* são baseados em *features* salientes e repetíveis que são seguidas ao longo das *frames;* os métodos *appearance-based* usam a informação da intensidade de todos os pixels na imagem ou sub-região da mesma*;* e os métodos híbridos usam a combinação dos dois anteriores. Na primeira categoria estão os trabalhos dos autores: [48], [18], [38], [60], [46], [54] e [51].

De seguida, são apresentados alguns trabalhos relevantes no domínio da visão monocular em sistemas de OV:

- [18] Corke *et al.* providenciaram uma proposta baseada em imagens omnidirectionais a partir de uma câmara catadióptrica e do fluxo óptico.
- [38] Lhuillier e [46] Mouragnon *et al.* apresentaram uma abordagem baseada no algoritmo local *windowed-bundle adjustment* para recuperar tanto o movimento como o mapa 3D. Eles usaram o *five-point* RANSAC [47] para remover os *outliers.* Este algoritmo tornou-se muito popular em OV e foi usado em diversos trabalhos neste domínio.
- [60] Tardif *et al.* apresentaram uma abordagem sem o *bundle ajustment*. Este trabalho foi realizado em um carro durante um percurso longo de 2.5 km. Contrariamente aos trabalhos anteriores, eles dissociaram a estimação da rotação e da translação. A rotação foi estimada usando pontos no infinito e a translação com base na recuperação do mapa 3D. Correspondências falsas foram removidas com o *five-point* RANSAC.

Em sistemas de OV, os métodos da visão monocular são interessantes, porque um sistema de OV com visão estéreo degenera para o caso monocular quando a distância até à cena é muito maior do que a linha de base estéreo (isto é, a distância entre as duas câmaras). Neste caso, a visão estéreo torna-se ineficiente e os métodos monoculares devem ser usados.

Uma vantagem de um sistema estéreo comparado com o monocular, é que a correspondência de *features* precisa ser calculada somente entre dois cenários em vez de três cenários como no caso monocular. Outra vantagem é que as features em 3D são calculadas directamente em escala absoluta.

1.3 Estrutura da dissertação

Esta dissertação está organizada da seguinte forma:

- **Capítulo 2**: É efetuada uma descrição da plataforma experimental e do framework ROS.
- **Capítulo 3**: Explicação do algoritmo de Odometria Visual, analisando os diversos passos do processo.
- **Capítulo 4**: Explicação do algoritmo RGB-D SLAM.
- **Capítulo 5**: Apresentação de resultados experimentais a partir de vários percursos efetuados.
- **Capítulo 6**: Conclusões do trabalho e apresentação de algumas sugestões para trabalho futuro.

2 Plataforma experimental

Neste capítulo é abordada a plataforma experimental utilizada nos testes práticos, bem como o sistema de software instalado que permitiu ser possível a navegação. A plataforma que foi utilizada é constituída por um robô Pioneer 3DX e pelo dispositivo Microsoft Kinect (ver Figura 2.1). Estes dois elementos pertencem ao Instituto de Sistemas e Robótica (ISR) da Universidade de Coimbra (UC) e serão caracterizados neste capítulo.

Figura 2.1 – Vista frontal da plataforma experimental.

2.1 Pioneer 3DX: características

Os testes realizados na prática só foram possíveis com o uso deste robô, chamado de Pioneer 3DX ou apenas P3DX (Figura 2.2). Em [1] está disponível o site oficial, onde são apresentadas várias aplicações envolvendo este tipo de robôs tais como: mapeamento, teleoperação, localização, monitorização, reconhecimento, entre outras. Além disso, também apresentam várias características úteis tais como, versatilidade, segurança e longevidade. Este robô foi escolhido devido a estas características e por ser considerado um dos mais populares do mundo para estudo científico e educação nas universidades e institutos.

Figura 2.2 – Robô P3DX - Uma base para muitas aplicações.

O P3DX é um robô móvel inteligente de condução diferencial adequado para ambientes de laboratório e outros ambientes interiores (*indoor*). Ele tem um peso reduzido (apenas 9 kg) e apresenta um corpo robusto de dimensões reduzidas em alumínio. Estas duas características em conjunto permitem a realização de trajectórias com elevada mobilidade. Em pisos planos, o P3DX pode mover-se a uma velocidade máxima de 1.6 m/seg. Para velocidades lentas, ele pode levar no seu topo (base) até 23 kg de peso adicional. A base permite a junção de vários tipos de acessórios (ou dispositivos) tais como: lasers, manipuladores robóticos, câmaras e outros.

Na tabela 2.1 são apresentadas algumas especificações relevantes do P3DX.

Características	Descrição
Dimensões	455mm comprimento x 381mm largura x 237mm altura
Capacidade de carga	até 23kg
Velocidade rotação	300^o/seg
Máximo ângulo subida	25^o
Protecção eléctrica	IP 20
Temperatura de Operação	0^oC−35^oC
Microcontrolador E/S	32 E. digitais, 8 S. digitais, 8 E. analógicas, 3 portas série de extensão

Notas: E=entrada; S=saída ; IP 20 = O equipamento eléctrico está protegido contra objectos sólidos com diâmetro acima de 12mm e não está protegido contra a água.

Tabela 2.1 – Especificações técnicas do P3DX.

Todos os robôs Pionner 3 são constituídos pelos seguintes elementos essenciais ([53]):

- Mesa - Corresponde ao topo do robô, composto por uma superfície plana para a montagem de outros componentes.
- Botão de Paragem do Motor - Efectua a paragem da alimentação dos motores.
- Painel de Controlo do Utilizador - Efectua o acesso ao microcontrolador. Contém botões de controlo, indicadores e uma porta série RS-232.
- Corpo, Nariz, e Painel de Acessórios - O corpo do Pioneer contém baterias, motores de tração, electrónica e outros componentes, incluindo o anel de sonares. O Nariz é onde está localizado o PC interno, o acesso é feito removendo alguns parafusos e abertura do nariz. O Painel de Acessórios é um painel removível do lado direito, onde se pode instalar outros acessórios.
- Anel de Sonares - Permitem a detecção de objetos, o reconhecimento de padrões, e também a localização e navegação. A localização dos sonares pode ser frontal , lateral ou traseira.
- Motores, Rodas, e Encoders - O sistema de tração dos robôs Pioneer 3 utiliza motores DC reversíveis de elevada velocidade e elevado torque. Cada motor é equipado com um encoder óptico de alta resolução para definições precisas de posição e velocidade. Os encoders estão situados por trás das rodas dianteiras.
- Baterias e Alimentação - Contém até três baterias - hot swaping[1], 7A-h, 12V DC, fechadas e acessíveis na porta traseira.

Na Figura 2.3 são mostrados alguns elementos essenciais do P3DX.

[1] hot swaping descreve a substituição de componentes sem interrupção significativa do sistema.

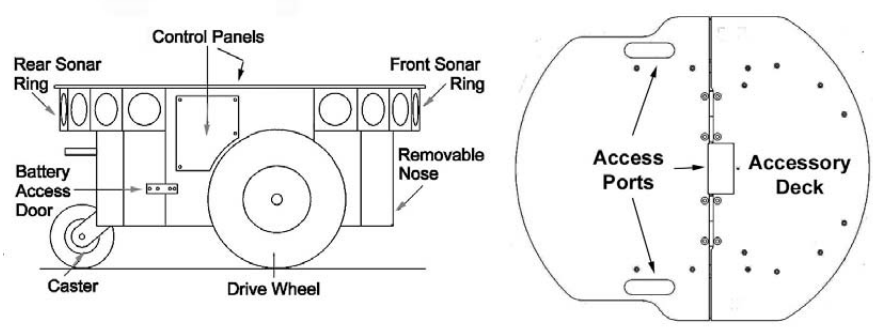

Figure 2.3 – Arquitetura do P3DX.

2.1.1 Controlo em modo de teleoperação

Os robôs Pioneer trazem um *Software Development Kit* (SDK), que é um conjunto de bibliotecas e aplicações que ajudam no desenvolvimento de projetos na área da robótica. Este software está disponível para Linux e Windows. O SDK é constituído pelo seguinte software: Aria, MobileSim, MobileEyes, Mapper 3-Basic e SONARNL. Para o controlo em modo de teleoperação apenas é necessário instalar o Aria e o MobileEyes.

Na lista seguinte são descritos os dois softwares usados:

- O software *Advanced Robotics Interface for Applications* (Aria) está incluído em todas as plataformas da empresa MobileRobots. O Aria é um ambiente de desenvolvimento em código-livre baseado em C++ que fornece uma interface robusta para uma variedade de sistemas inteligentes robóticos, incluindo o microcontrolador do robô. No interior do Aria faz parte o ARNetworking, que é usado para facilitar a comunicação em rede com a plataforma, isto é, fornece uma camada de abstração para transferência de informação baseada em TCP/IP.
- O MobileEyes é um interface gráfico com o utilizador (Figura 2.4), permitindo visualizar o movimento do robô dentro de um mapa, visualizar um fluxo de imagens retiradas de câmaras e também permite enviar comandos para controlo do robô (modo teleoperação). Além disso tem ferramentas para alterar os parâmetros de configuração e controlar o software de navegação e localização. Ele interage com o software servidor através do envio de comandos personalizados.

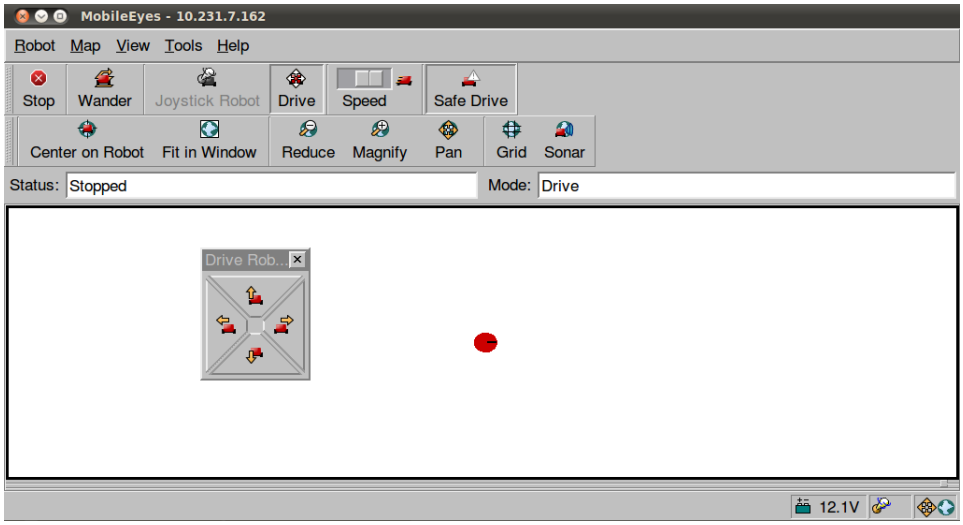

Figura 2.4 – Interface Gráfico - MobileEyes.

A realização do modo teleoperação requer dois computadores portáteis (PCs) com placa de rede sem fios (*Wireless*). Este modo utiliza o paradigma cliente-servidor. Portanto, um dos computadores corresponde ao cliente (PC *offboard*) e o outro corresponde ao servidor (PC *onboard*). O cliente para funcionar precisa da aplicação MobileEyes instalada e o servidor da aplicação Aria. O servidor é ligado directamente ao P3DX via cabo série e o cliente liga-se ao servidor através de uma ligação sem fios.

Para estabelecer a ligação cliente-servidor, em primeiro lugar é recomendado que o robô esteja imobilizado para evitar possíveis movimentos indesejados. Depois ligar o robô no botão "MAIN POWER" e escrever o seguinte comando na consola do servidor:

```
sudo /usr/local/Aria/ArNetworking/examples/serverDemo -rp /dev/ttyUSB0
```

Quando a ligação entre o servidor e o robô estiver estabelecida então o cliente pode iniciar a aplicação MobileEyes através do comando:

```
sudo /usr/local/MobileEyes/bin/MobileEyes
```

O MobileEyes apresenta uma caixa de diálogo com três campos, "User Name", "Password" e "Robot Servers" (ver Figura 2.5).

Figura 2.5 – Imagem de apresentação do MobileEyes.

O campo Robot Servers é de prenchimento obrigatório. Neste campo, escreve-se o nome do host da rede ou o endereço IP do computador onde o servidor está a ser executado. Os campos User Name e Password normalmente não são necessários e podem ficar em branco.

A ligação cliente-servidor é estabelecida se os dados introduzidos estiverem correctos. Após este passo, o utilizador através do PC cliente poderá enviar comandos ao robô. Desta maneira é possível navegar com o robô em modo manual sem ser necessário sensores para evitar os obstáculos.

2.2 Microsoft Kinect

Existem diferentes tipos de abordagens para controlar uma máquina, como um computador ou um uma consola de jogos, os clássicos são o teclado, rato ou o comando da consola. O dispositivo Kinect cria uma nova categoria e foi originalmente desenvolvida para jogar jogos na XBOX 360 sem haver necessidade de segurar um dispositivo como um comando de jogos. Os investigadores escreveram um controlador *open source* chamado *libfreenect* para ligar e usar o Kinect com o computador em vez do uso habitual com a XBOX 360 [34]. A ligação entre o PC e Kinect é efetuada directamente via *Universal Serial Bus* (USB).

Neste trabalho o único sensor utilizado é o Kinect, que se encontra localizado no topo do robô P3DX. Este dispositivo está a receber muita atenção graças à rapidez do sistema em estimar a pose humana. Este sensor é considerado uma boa escolha por ser de baixo custo em relação a outros sensores e a informação capturada tem uma boa resolução. Ele é considerado um dos principais dispositivos 3D para medição em ambientes interiores robóticos, construção de cenas em 3D e reconhecimento de objetos, devido aos atributos fiabilidade e velocidade de medicão [57].

2.2.1 Características

Figura 2.6 – Arquitetura do Kinect.

O Kinect contém um sistema sofisticado óptico que consiste em três componentes: um projetor de luz infravermelha (IV), uma câmara infravermelha (IV) e uma câmara RGB[2]. O projetor de luz IV e a câmara IV funcionam como um sensor de profundidade e são usados para triangular pontos no espaço. A câmara RGB pode ser usada para identificar conteúdo numa imagem e a textura de pontos 3D. Entre a câmara RGB e o projetor de luz IV existe um led de estado com a cor verde que não desempenha nenhuma função no sistema óptico actual. Uma característica específica do Kinect é a combinação do projetor e da câmara, em vez das habituais duas câmaras para visão estéreo, a fim de permitir uma redução do tempo computacional.

Em relação ao áudio, existem dois conjuntos de microfones constituídos por quatro sensores e são capazes de separar o som de diferentes direcções. O principal foco deste trabalho é a parte visual, portanto o som não vai ser considerado.

Na Figura 2.6 é visível o projetor de luz IV situado à esquerda, e as outras câmaras localizadas no meio muito próximas uma da outra.

Na tabela 2.2 são apresentadas as especificações do Kinect.

[2]RGB é a abreviatura do sistema de cores aditivas formado por Vermelho (Red), Verde (Green) e Azul (Blue).

Campo de Visão (Horizontal, Vertical, Diagonal)	58^{o}Hor, 40^{o}Ver, 70^{o}Dia
Tamanho imagem profundidade	VGA (640x480)
Resolução espacial x/y	3mm para 2m de distância
Resolução profundidade Z	1cm para 2m de distância
Frame rate	60 FPS
Resolução VGA	40ms
Alcance de Operação	0.8 - 4m
Tamanho da imagem RGB	UXGA (1600x1200)
Áudio (microfones embutidos)	2
Áudio (entradas digitais)	4
Interface de dados	USB 2.0
Fonte de Alimentação	USB 2.0
Consumo de energia	2.25W
Dimensões (largura x altura x espessura)	28x7x7 (em cm)
Ambiente de Operação	Interior
Temperatura de Operação	0^o - 40^{o}C

Tabela 2.2 – Especificações do Kinect.

2.2.2 Imagem RGB-D

Neste trabalho a captura de imagens é feita pelo Kinect. Este sensor é um dispositivo RGB-D, ou seja, permite representar as cores (RGB) e a profundidade de uma cena (D, do inglês *Depth*). Portanto, uma cena capturada pelo Kinect é usualmente representada por um par de imagens de resolução 640x480, com uma imagem em formato colorido RGB (24bits/pixel) e a outra imagem representando a profundidade de cada pixel (D). A profundidade representa a distância dos pixels em relação ao sensor, formando um mapa LxCxP (Linha x Coluna x Profundidade). A imagem de profundidade é construída a partir do sensor de profundidade. Inicialmente o projetor emite um padrão irregular de pontos de luz IV com intensidades diferentes. Depois a câmara IV reconstrói a imagem de profundidade reconhecendo a distorção do padrão através do processo de triangulação. Assim é possível determinar a que distância o objecto se encontra.

A Figura 2.7 mostra um exemplo de uma imagem captura pelo Kinect, com a imagem RGB Fig.2.7(b) e a imagem do mapa de profundidade Fig.2.7(a). Na imagem de profundidade são aplicadas falsas cores (imagem apresentada com cores não convencionais), com as cores frias (tendendo ao azul) representando os pixels mais distantes, e as cores quentes (tendendo ao vermelho) representando os pixels mais próximos do sensor.

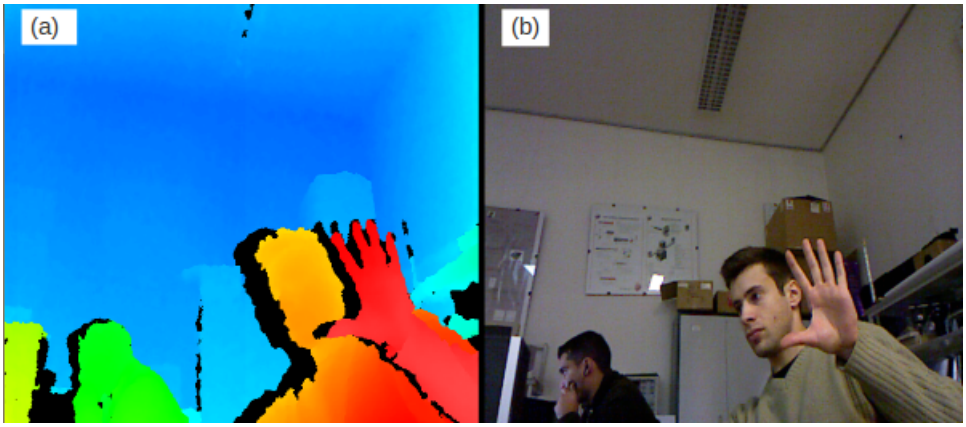

Figure 2.7 – Imagem captura pelo Kinect. (a) Imagem de profundidade em falsas cores; (b) Imagem RGB; (Lab. de Visão por Computador no piso 0 - ISR UC)

2.2.3 Limitações

O Kinect é um dispositivo com limitações no alcance, no campo de visão horizontal e no tipo de ambiente. Geralmente é comparado com os lasers por terem características semelhantes.

Em 2.2.1 foi visto que o Kinect tem um campo de visão horizontal de 58^o enquanto que os lasers Hokuyo têm entre 240° e 270°. Existem sensores com um campo completo de 360^o, como por exemplo o LIDAR (do inglês *Light Detection and Ranging*) Neato XV-11. Em presença de aplicações de Localização e mapeamento simultâneos[3] (SLAM, do inglês *Simultaneous Localization and Mapping*) o maior campo de visão permite visualizar mais características e também permite ao robô construir um mapa sem falhas de forma eficiente. Um robô com um estreito campo de visão vai ter a necessidade de fazer vários percursos no mesmo ambiente para construir um mapa completo.

Quanto ao alcance, o Kinect pode ir até aos 4m enquanto que existem lasers actuais que podem chegar até aos 60m, como por exemplo, o UTM-30LX. Portanto, em aplicações *SLAM*, um robô equipado com o Kinect pode ter dificuldades em navegar em corredores muito longos.

O ambiente é outro factor que pode influenciar o funcionamento. Visto que, o Kinect foi apenas projectado para ambientes interiores enquanto que os lasers já foram utilizados com sucesso em ambientes exteriores e interiores.

Uma solução para estas limitações é usar o Kinect em conjunto com um laser para obter melhores resultados.

2.3 Robot Operating System

Neste trabalho foi escolhida a framework *Robot Operating System* (ROS) [52], que é um conjunto de bibliotecas e ferramentas para facilitar a pesquisa e o desenvolvimento de sistemas robóticos. Após uma breve análise, este sistema demonstrou ser o mais adequado a aplicar na plataforma móvel, tanto em uma perspectiva de implementação quer numa perspectiva de futuro.

[3] A técnica SLAM é utilizada por robôs e veículos autónomos para o mapeamento de um ambiente ao mesmo tempo que é efetuada a localização.

2.3.1 Características

O ROS é uma plataforma distribuída, desenvolvida no Stanford Artificial Intelligence Lab e posteriormente pela Willow Garage. Está muito bem documentada, é de código aberto e de fácil aprendizagem. Possui suporte para os principais robôs usados actualmente, tanto academicamente como comercialmente, além de simuladores robóticos.

Todo o sistema está bem organizado e as bibliotecas de software seguem padrões de programação, tornando a leitura do código fonte um trabalho simples.

2.3.2 Arquitetura de Software

O sistema ROS é organizado por pacotes (directório contendo ficheiros de código, executáveis, *makefiles*[4] e ficheiros de configuração, de uma forma estruturada) e dentro de cada pacote podem existir nodos[5], mensagens e serviços. É possível também organizar um conjunto de pacotes em *stacks*[6].

A divisão em partes distintas do sistema é atingida com a separação dos diferentes nodos em execução e utilizando o interface de comunicação (mensagens e serviços) mantendo assim uma baixo nível de inter-dependência entre os diversos nodos. Para que um novo processo tenha conhecimento de outros processos e possam existir canais de comunicação entre eles, este deverá estar ligado a um programa central denominado roscore (Figura 2.8), este programa é o núcleo de todo o sistema ROS e funciona como um servidor de nodos, de uma forma semelhante a uma rede peer-to-peer de processos e deverá ser o primeiro a ser lançado.

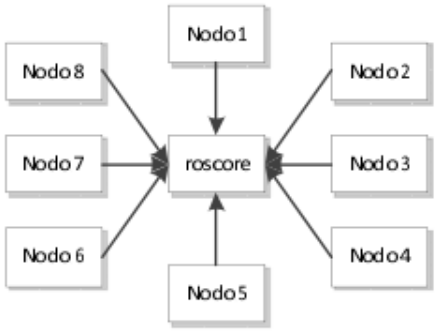

Figure 2.8 – Organização modular do sistema de execução

O sistema de mensagens e o sistema de serviços são baseados em mensagens predefinidas. As mensagens podem ser definidas por vários campos de dados de acordo com a informação que se pretende comunicar. Os tipos de dados disponíveis são os tipos de dados primitivos (inteiros, vírgula flutuante, booleanos entre outros) da linguagem C++ (ou Python). É possível também criar mensagens que têm como campos outras mensagens, ou até vectores de mensagens. O

[4]Um *makefile* é um ficheiro para configurar a compilação. A ideia é simplificar a compilação de programas.

[5]Um nodo pode ser definido como um processo que utiliza a plataforma ROS.

[6]Uma *stack* é uma forma de organizar um conjunto de pacotes que têm como objectivo no seu conjunto realizar uma tarefa.

sistema de mensagens permite que um nodo publique[7] informação num tópico[8] tornando essa informação disponível para os nodos que subscreveram essa informação (Figura 2.9 a)). As mensagens têm uma estrutura bem definida, pelo que se um nodo tentar comunicar com uma estrutura de mensagem diferente não irá conseguir.

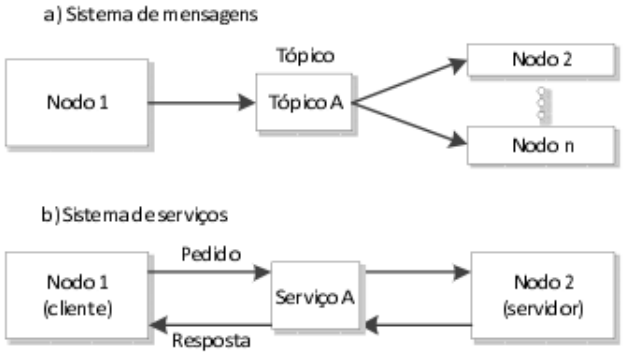

Figure 2.9 – Interface de Comunicação. a) Descrição do sistema de mensagens
b) Descrição do sistema de serviços

O sistema de serviços é um mecanismo que permite a um determinado nodo efectuar um pedido a outro nodo que anuncia[9] que tem o serviço desejado disponível (Figura 2.9 b)). A comunicação é terminada quando o nodo que recebe o pedido responde.

São suportadas oficialmente pelo ROS as linguagens de programação Python, C++, Octave, e LISP, existindo também algum suporte numa fase inicial para Java e Matlab.

Grande parte da interação com o sistema é efetuada utilizando linha de comandos, alguns dos comandos mais relevantes são:

- Para criar um pacote:
  rocreate-pkg <nome> <dependência 1> ... <dependência n>
- Para compilar um pacote :
  rosmake <pacote>
- Para lançar um nodo:
  rosrun <pacote> <nodo> <argumentos>
- Para ver as mensagem enviadas para um tópico:
  rostopic echo <nome do tópico>
- Para enviar uma mensagem para um tópico:
  rostopic pub <nome do tópico> <tipo de mensagem> <dados>

2.3.3 Instalação

A escolha do sistema operativo é sempre importante em projectos que requerem navegação de robôs e captação de imagens com câmaras. Para a elaboração deste trabalho foi instalada nos

[7] Publicar significa enviar informação de um nodo, e encapsulada num formato predefinido (mensagem).

[8] Semelhante a um endereço electrónico, é valido para todo o sistema ROS o que significa que todos os nodos do sistema podem ler e escrever em um tópico.

[9] Anunciar significa no contexto do sistema ROS indicar aos restantes nodos do sistema que uma mensagem ou um serviço tão disponíveis.

dois PCs a distribuição GNU/Linux Ubuntu - versão Lucid Lynx (10.04), e também a framework ROS apresentada na secção *2.3*. A versão Lucid Lynx (10.04) foi escolhida por ser uma versão *Long Term Support* (LTS) e por ser muito usada em aplicações robóticas. A versão do Kernel usada pelo PC *onboard* é a 2.6.32-41-generic x86_64 e a do *offboard* é a 2.6.32-38-generic i686.

Existem várias distribuições do ROS que foram analisadas tais como, ROS Box Turtle, ROS C Turtle e ROS Diamondback. A distribuição escolhida foi o ROS Diamondback (configuração Desktop-Full) porque contém a primeira versão do *OpenNI Kinect* (disponibliza *stacks* de apoio e drivers ao dispositivo Kinect) e uma biblioteca estável em código aberto - *Point Clouds Library*[10](PCL). O principal objectivo desta distribuição é fazer o ROS mais pequeno e mais leve. Esta distribuição foi instalada no PC *onboard*, porque este já tem instalado os drivers para o Kinect e apresenta um desempenho melhor que o *offboard* (Tabela 2.3).

Características	PC *onboard*	PC *offboard*
Processador	Intel Core Duo T9400 @ 2.53GHz	Intel Atom N280 @ 1.66GHz
Memória	4GB	1GB
Placa gráfica	nVidia GeForce 9600M GS	Intel Mobile 945GM

Tabela 2.3 – Características relevantes dos computadores usados nos testes práticos.

A instalação da distribuição ROS Diamondback é efetuada com os os seguintes passos [9]:

1. **Configurar os repositórios do Ubuntu**
   Antes de iniciar a instalação deve configurar os repositórios do Ubuntu para permitir transferências a partir da internet. Escolher os repositórios "restricted", "universe", e "multiverse". Pode obter mais informações sobre este passo aqui [6].
2. **Configurar o ficheiro sources.list para aceitar software do site ROS.org**
   Para a versão Ubuntu 10.04 (Lucid) escrever o seguinte comando na consola:

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu lucid main" > /etc/apt/sources.list.d/ros-latest.list'
   ```

   Para a versão Ubuntu 10.10 (Maverick):

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu maverick main" > /etc/apt/sources.list.d/ros-latest.list'
   ```

   Para a versão Ubuntu 11.04 (Natty):

   ```
   sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu natty main" > /etc/apt/sources.list.d/ros-latest.list'
   ```

[10]A biblioteca PCL é utilizada para manipular nuvens de pontos

3. **Configurar as senhas**

```
wget http://packages.ros.org/ros.key -O - | sudo apt-key add -
```

4. **Instalação**

   Actualiza a lista de repositórios locais do servidor ROS.org:

```
sudo apt-get update
```

   Instalação da configuração Desktop-full:

```
sudo apt-get install ros-diamondback-desktop-full
```

5. **Configurar o ambiente**

```
echo "source /opt/ros/diamondback/setup.bash" >> ~/.bashrc . ~/.bashrc
```

3 Odometria Visual

Hoje em dia é importante conhecer a posição e orientação de um agente (por exemplo: veículo, humano e robô) dentro de um ambiente estruturado, usando apenas uma ou várias câmaras para esse efeito. A este método de estimar a pose (posição e orientação) de um agente com base em câmaras (imagens capturadas) denomina-se Odometria Visual (OV). Um sistema que utiliza OV pode utilizar câmaras com visão monocular, intitulando-se de OV monocular, ou então, câmaras com visão estéreo, denominando-se de OV estéreo.

A OV tem muitas aplicações tais como, electrónica de miniaturização, realidade aumentada e robótica móvel (aplicações de *SLAM).*

O termo OV foi escolhido por ser semelhante ao método de odometria, que estima o movimento do robô por integração consecutiva da informação do deslocamento das rodas ao longo do tempo. Do mesmo modo, a OV estima a pose do veículo de forma incremental, através da análise das mudanças do movimento induzidas nas imagens das câmaras a bordo. A principal vantagem da OV em relação à das rodas consiste no facto de não ser afectada pelo deslizamento das rodas em terrenos irregulares ou outras condições adversas. Além disso, foi demonstrado que a OV proporciona mais precisão na estimação de trajectórias, com um erro de posição relativo variando entre 0.1 e 2%. Esta qualidade torna a OV um interessante complemento à odometria das rodas e, adicionalmente, para outros sistemas de navegação.

Num sistema de OV a estimação da pose é realizada com base em um algoritmo. Para que o algoritmo produza os melhores resultados possíveis, as seguintes condições têm que ser verificadas: suficiente iluminação no ambiente, e a cena tem de permanecer estática e conter textura suficiente para permitir a extracção do movimento aparente. Além disso, deve ser garantido que as *frames* capturadas de modo contínuo contenham suficiente sobreposição da cena.

Uma vez que a OV calcula a trajectória da câmara de forma incremental, pose após pose, então os erros introduzidos por cada novo movimento, *frame* após *frame*, vão-se acumulando ao longo do tempo. Isto gera um desvio da trajectória estimada em relação à trajectória real. Para algumas aplicações é de extrema importância manter o desvio o mais pequeno possível, o que pode ser feito através da otimização local das últimas m poses da câmara. Esta acção denomina-se *windowed bundle adjustment* e tem sido usada em muitos trabalhos tais como, [25] e [59]. Mais tarde vai ser descrita neste capítulo.

Outros problemas em OV são os *outliers*[1] e correspondências falsas que podem levar a poses erradas, e por conseguinte as estimativas seguintes irão conter este erro. Assim, a trajectória global estimada também irá conter este erro. Para que o movimento da câmara seja estimado com precisão é importante que os *outliers* sejam removidos de uma forma robusta, para esse efeito pode-se usar o algoritmo RANSAC (do inglês *Random Sample Consensus*) [22] que vai ser abordado mais tarde neste capítulo.

[1] *outliers*: pontos de dados que não se ajustam ao modelo correspondente desejado, estão fora de uma certa tolerância de erro.

3.1 Descrição do método

Nesta secção é feito um breve resumo ao núcleo da computação em sistemas de OV, no qual é descrito com detalhe como é calculado as transformações relativas entre as poses.

Um determinado agente ligado rigidamente a um sistema de câmaras move-se num determinado ambiente e captura imagens em instantes de tempo discreto k. Se o sistema tiver visão monocular, o conjunto de imagens capturadas em intervalos k é representado por $I_{0:n} = \{I_0, \ldots, I_n\}$. No caso de ter visão estéreo, então existem duas imagens (esquerda e direita) em cada instante de tempo, representadas por $I_{e,0:n} = \{I_{e,0}, \ldots, I_{e,n}\}$ *e* $I_{d,0:n} = \{I_{d,0}, \ldots, I_{d,n}\}$. Na Figura *3.1* é apresentado uma ilustração deste cenário.

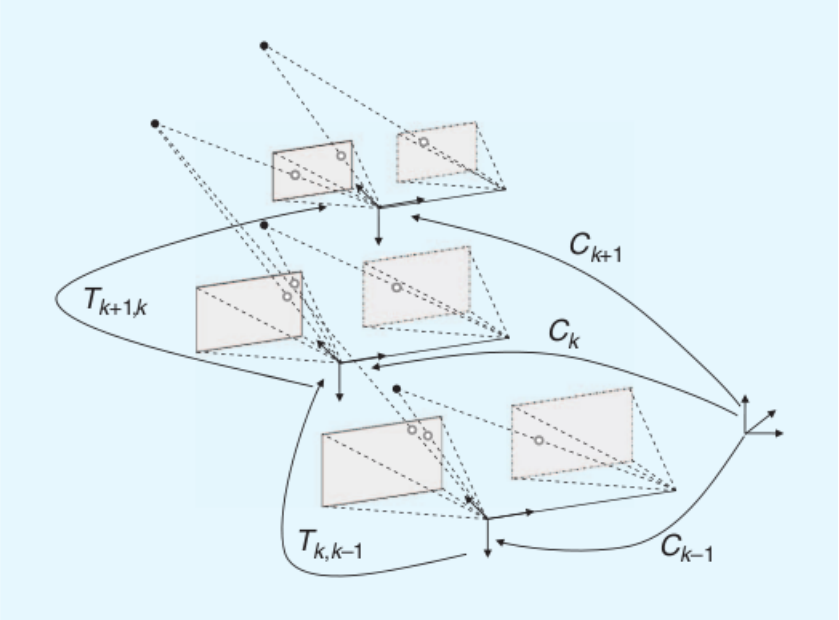

Figura 3.1 – Ilustração do problema de OV para visão estéreo. As poses relativas $T_{k,k\text{-}1}$ de adjacentes posições da câmara são calculadas a partir de características visuais e concatenadas para obter poses absolutas C_k em relação à coordenada da *frame* inicial no instante $k = 0$.

Em cada *frame* é assumido por simplicidade que o sistema de coordenadas da câmara coincide com o sistema de coordenadas do agente. No caso do sistema tiver visão estéreo, então considera-se que o sistema de coordenadas da câmara esquerda pode ser utilizado como origem sem perda de aspectos gerais.

As posições de uma câmara em instantes de tempo adjacentes $k-1$ e k estão relacionadas por uma transformação de corpo rígido $T_{k,k-1} \in \mathbb{R}^{4\times4}$ da seguinte forma:

$$T_{k,k-1} = \begin{bmatrix} R_{k,k\text{-}1} & t_{k,k\text{-}1} \\ 0 & 1 \end{bmatrix}, \tag{3.1}$$

onde $R_{k,k-1} \in \mathbb{R}^{3\times3}$ é a matriz de rotação e $t_{k,k-1} \in \mathbb{R}^{3\times1}$ é o vector de translação. O conjunto $T_{1:n} = \{T_{1,0}, \ldots, T_{n,n-1}\}$ contém todos os movimentos sequenciais da câmara. Para simplificar

a notação, a partir de agora, T_k será usado em vez de $T_{k,k-1}$. Finalmente, o conjunto das poses da câmara $C_{0:n} = \{C_0, \ldots, C_n\}$ contém as transformações da câmara em relação à coordenada da *frame* inicial no instante $k = 0$. A pose actual C_n pode ser calculada juntando todas as transformações T_k ($k = 1 \ldots n$), e assim, $C_n = C_{n-1}T_n$, com C_0 sendo a pose da câmara no instante $k = 0$, que pode ser definida arbitrariamente pelo utilizador.

A principal tarefa em sistemas de OV é calcular as transformações relativas T_k a partir das imagens I_k e I_{k-1}, e depois juntar as transformações para recuperar a trajectória completa $C_{0:n}$ da câmara. Isto significa que o sistema consegue recuperar de forma incremental a trajectória, pose após pose.

Existem duas principais abordagens para calcular o T_k: os métodos *appearance-based* (ou métodos globais) e os métodos *feature-based.*

Os métodos *appearance-based* usam a informação da intensidade de todos os pixels nas duas imagens de entrada e os métodos *feature-based* extraem *features*[2] repetíveis e salientes ao longo das imagens. A maioria dos sistemas de OV implementados utiliza os métodos *feature-based*, porque são mais rápidos, mais precisos e mais baratos do que os métodos *appearance-based*.

3.2 Principais tarefas de um sistema de OV

A fim de efectuar um sistema de OV é necessário realizar várias etapas. Um diagrama de blocos para melhor ilustrar esta metodologia é apresentado na Figura 3.2.

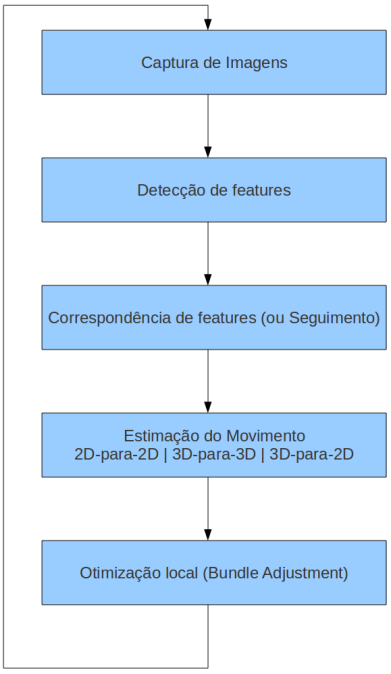

Figura 3.2 – Etapas de um sistema de Odometria Visual.

[2] *feature* é uma região na imagem que difere da sua vizinhança em termos de intensidade, cor e textura.

O diagrama mostra os passos fundamentais para obtenção da pose da câmara (ou do agente) em relação a um referencial inercial.

A primeira etapa representa a captura de uma nova imagem I_k, ou um par de imagens no caso da câmara ter visão estéreo.

A segunda etapa descreve o processo de detecção de *features*, que consiste na procura de características visuais salientes que possam corresponder noutras imagens.

A terceira etapa apresenta dois métodos diferentes para encontrar *features* e suas correspondências. Os métodos são: correspondência de *features* e seguimento de *features*. Muitas vezes pode acontecer que as correspondências estejam erradas, isto é, o par da correspondência não é o correcto, assim, o sistema deve ser capaz de identificar e remover estes erros. Uma solução para o problema é através do algoritmo RANSAC.

A quarta etapa consiste no cálculo do movimento relativo T_k entre os instantes de tempo $k-1$ e k. Após a obtenção do T_k, então é possível calcular a pose da câmara C_k por junção do T_k com a pose anterior C_{k-1}.

Finalmente, a última etapa descreve a aplicação do algoritmo *Bundle Adjustment* ao longo das últimas m *frames* com o objectivo de obter uma estimativa mais precisa da trajectória local.

3.3 Detecção de *features*

A detecção de *features* (etapa 2) é o processo responsável por determinar *features* numa imagem. Estas *features* devem ser escolhidas de forma a serem possíveis de encontrar e recuperar futuramente. Relativamente a sistemas de OV, foi descoberto que a distribuição de f*eatures* numa imagem afecta de uma forma considerável os resultados obtidos. Em particular, quanto mais *features* encontradas mais estáveis são os resultados da estimação do movimento (etapa 4) e vice-versa. As *features* são procuradas numa imagem através de um detector de *features*. Para ser um bom detector de *features* tem de obedecer às seguintes características:

- Precisão na localização, tanto em posição como em escala
- Repetibilidade, ou seja, um grande número de *features* deve ser detectado de novo nas próximas imagens
- Eficiência computacional
- Robustez ao ruído e desfocagem
- Distinção (de modo que as *features* possam ser correspondidas precisamente entre imagens diferentes)
- Invariância à iluminação e mudanças geométricas

Na área da OV os detectores de *features*, tais como *cantos*[3] ou *blobs*[4], são muito importantes porque é possível saber com precisão a sua posição numa imagem. A Figura 3.3 mostra a aplicação destes dois detectores de *features*.

[3] *canto* é definido como um ponto de intersecção de duas ou mais arestas.

[4] *blob* são regiões mais brilhantes (ou mais escuras) do que o meio circundante.

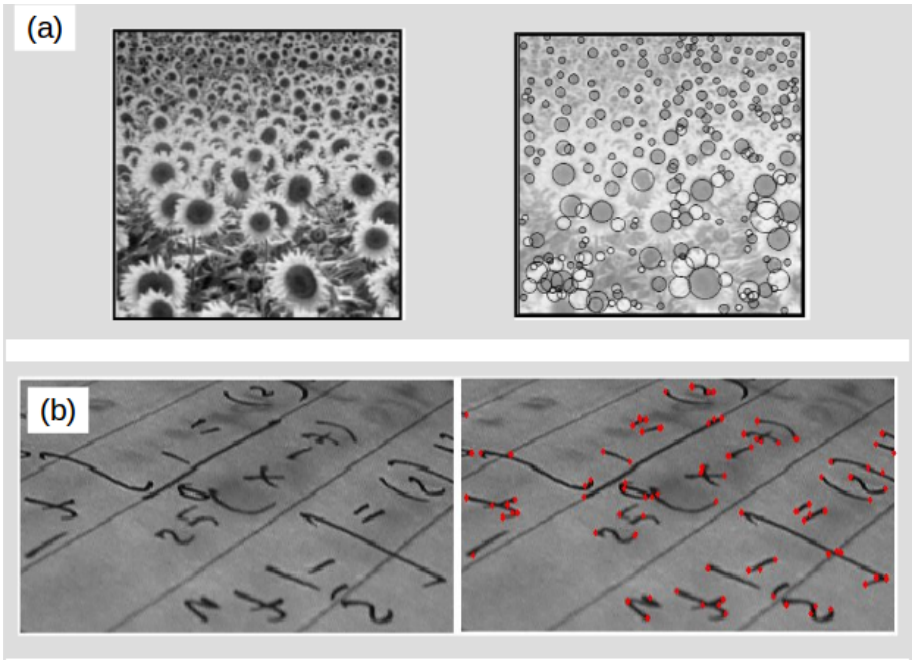

Figura 3.3 – Exemplos de métodos de extracção de *features*. (a) Aplicação do detector de *features blob* à imagem da esquerda (original); (b) Aplicação do detector de *features canto* à imagem da esquerda (original).
As imagens (a) e (b) foram retiradas respectivamente dos sites [2] e [3].

Neste trabalho, para a determinação de *features* nas imagens foram escolhidos os modernos algoritmos detectores de *features* SIFT (do inglês *Scale Invariant Feature Transform*)[40] e SURF (do inglês *Speeded Up Robust Features*)[14]. Estes algoritmos fazem parte do conjunto detectores de *blobs* e costumam ser invariantes a diversas características presentes em imagens, como iluminação, rotação, escala, translação, entre outras. Por estas razões são algoritmos adequados à tarefa de determinação de f*eatures* em OV, considerando que podemos ter robôs móveis com dinâmicas rápidas, tendo assim variações significativas no número de *frames*. A seguir é feita uma breve descrição de cada algoritmo.

3.3.1 SIFT

O algoritmo SIFT é um método robusto para extrair e descrever *features* de uma imagem. Este algoritmo é dividido em quatro etapas:

1. **Detecção de extremos do espaço de escala:** Esta etapa consiste na identificação de pontos de interesse invariáveis à escala e rotação. A procura é feita através de uma função de escala chamada Gaussiana. A repetida convolução da imagem original em tons de cinza com a função Gaussiana gera um conjunto de Gaussianas. A subsequente subtração dos resultados das Gaussianas resulta na chamada Diferença das Gaussianas (DOG, do inglês *Difference Of Gaussian*). Em seguida, as imagens Gaussianas são quantizadas pela metade e o processo se repete por algumas iterações. Este processo garante invariância à escala.
2. **Localização de *features*:** Para cada localização em que foi detectado um extremo, um modelo detalhado é ajustado a fim de determinar a localização exacta e a escala. As *features* são selecionadas com base em medidas de estabilidade. Nesta etapa são definidos os melhores pontos para o sistema de mapeamento através de medidas do gradiente.

3. **Atribuição da orientação:** A orientação é definida para cada *feature* através dos gradientes locais da imagem. Depois toda a operação será feita em relação a dados transformados da imagem relativamente à orientação e escala de cada *feature*. Desta maneira, obtém-se invariância a estas transformações.
4. **Descritor das *features*:** O gradiente local de cada *feature* é obtido utilizando a vizinhança do ponto. Depois é transformado para uma representação que permite tolerância a níveis significativos de distorção e mudança de iluminação.

O algoritmo SIFT procura pontos na imagem que sejam invariantes a mudanças de escala, possibilitando a detecção de pontos em diferentes *viewpoints*[5] do mesmo objeto, com altas taxas de repetibilidade, através de uma localização de pontos em um espaço de escala. Foi descoberto que ele proporciona excelentes resultados em sistemas de OV.

3.3.2 SURF

O método SURF é uma alternativa baseada no método SIFT. É um método robusto e com baixo custo computacional para extrair e descrever *features* de uma imagem. O método SURF possui um desempenho que se aproxima de métodos similares como o SIFT e as seus *features* podem ser detectadas e comparadas mais rapidamente, mesmo em alguns casos sacrificando robustez a algumas variações. Comparações entre SIFT e SURF revelam que o algoritmo SURF pode ser mais rápido que o SIFT, e pode também ser mais resistente a alterações de iluminação e imagem desfocada ou com ruído. Isso permite que o SURF seja uma alternativa viável para determinadas aplicações. O algoritmo é dividido em 3 etapas:

- **Cálculo do integral da Imagem**
  O integral da imagem ($I_{\sum}$) numa localização $X = (x, y)$ representa a soma de todos os pixels na imagem de entrada I dentro de uma região rectangular formada pela origem e X. A equação que define o integral da imagem tem a seguinte forma:

$$I_{\sum}(X) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j) \tag{3.2}$$

- **Determinação das *features***
  Para a determinação das *features* são necessários dois passos. O primeiro passo é determinar a Fast-Hessian para cada escala diferente através de uma aproximação do kernel gaussiano de segunda ordem. O segundo passo é percorrer as imagens resultantes a fim de determinar as *features* para os quais serão calculados os seus descritores.

- **Criação do descritor de cada *feature***
  Para a determinação do descritor é necessário utilizar a soma das respostas do kernel 2D *Haar-wavelet* em diferentes orientações x e y.

3.4 Correspondência de *features* (ou seguimento)

O processo correspondência de *features* (ou seguimento) pertence à etapa 3. É um pré-requisito para muitas aplicações relacionadas com imagens, tais como, recuperação de uma

[5] viewpoint neste contexto significa *frame* de referência ou posição.

imagem, detecção do movimento e reconstrução da forma. As diferentes *features* determinadas pela etapa anterior precisam ser correspondidas, ou seja, é preciso determinar quais *features* são as mesmas em imagens distintas. Geralmente existem dois métodos diferentes para encontrar *features* e suas correspondências: correspondência de *features* e seguimento de *features*. Na lista seguinte são descritos resumidamente os dois métodos.

- O método de correspondência de *features* consiste na detecção de *features* de maneira independente em todas as imagens com um determinado detector de *features*. Idealmente, as *features* detectadas representam a mesma informação nas imagens. Depois são efetuadas as correspondências entre as *features* detectadas usando uma estratégia de comparação baseada em medidas de semelhança. Este método é muito usado em ambientes exteriores de grandes dimensões, em que as imagens são capturadas a partir de *viewpoints* distantes entre si, com a finalidade de limitar os problemas relacionados com desvio do movimento.

- O método seguimento de *features*, como alternativa, detecta *features* em apenas uma imagem com um determinado detector de *features*. Na próxima imagem a *feature* correspondente é procurada na área em torno da localização da *feature* detectada. Este método de "detecção e seguimento" é muito usado em ambientes interiores de pequenas dimensões e é adequado para aplicações de OV. Uma vez que as imagens são capturadas a partir de *viewpoints* próximos e a quantidade de movimento entre *frames* sucessivas geralmente é pequena. Portanto este foi o método utilizado nos testes práticos.

3.5 Estimação do Movimento

A etapa estimação do movimento representa o núcleo da computação em um sistema de OV. Ela efectua o cálculo do movimento da câmara entre a imagem actual e a imagem anterior, ou pares de imagens. A trajectória da câmara e do agente (assumindo que estão ligados rigidamente) pode ser recuperada completamente através da concatenação de todos os movimentos individuais.

Esta secção explica como a transformação relativa T_k pode ser calculada entre duas imagens I_{k-1} e I_k a partir de dois conjuntos de *features* correspondentes f_{k-1}, f_k nos instantes de tempo $k-1$ e k, respectivamente. Existem três métodos diferentes para o cálculo do T_k, dependendo se as correspondências das *features* são especificadas em três ou duas dimensões. Os métodos a que me refiro são:

- 2D-para-2D: Neste caso, ambos f_{k-1} e f_k são especificados pelas coordenadas 2D da imagem.
- 3D-para-3D: Neste caso, ambos f_{k-1} e f_k são especificados pelas coordenadas 3D da imagem. Para realizar esta acção, é necessário a triangulação de pontos 3D em cada instante de tempo, isto pode ser feito a partir de um sistema com câmaras estéreo, por exemplo.
- 3D-para-2D: Neste caso, f_{k-1} são especificados em 3D e f_k são as suas correspondentes reprojeções em 2D na imagem I_k.

O segundo método está disponível para sistemas que usam visão estéreo. Como nos testes práticos é usado uma câmara com visão estéreo então este foi o método usado. De seguida é feita uma descrição deste método.

O movimento da câmara T_k pode ser calculado por determinação da transformação de alinhamento dos dois conjuntos 3D de *features*. A solução geral consiste em encontrar um T_k que minimize a distância L_2 entre os dois conjuntos 3D de *features*

$$arg\min_{T_k} \sum_i \left\| \tilde{X}_k^i - T_k \tilde{X}_{k-1}^i \right\|, \tag{3.3}$$

onde o índice i denota a i-ésima *feature*, e $\tilde{X}_k$, $\tilde{X}_{k-1}$ são as coordenadas homogêneas dos pontos 3D, isto é, $\tilde{X} = [x, y, z, 1]^T$. A solução mínima envolve três correspondências não colineares 3D-para-3D, que pode ser usada para uma estimação robusta na presença de *outliers*. Para o caso de $n \geq 3$ correspondências, uma possível solução (de acordo com Arun et al. [12]) é calcular a parte da translação como uma diferença dos centros de massa dos conjuntos 3D de *features* e a parte da rotação é usar o algoritmo Decomposição em valores singulares (SVD, do inglês *Singular Value Decomposition*)[13].

A translação é dada por

$$t_k = \overline{X_k} - R\overline{X}_{k-1}, \tag{3.4}$$

onde $\bar{.}$ significa valor médio aritmético.

A rotação pode ser calculada de modo eficiente usando o SVD como

$$R_k = VU^T, \tag{3.5}$$

onde $USV^T = svd\left(\left(X_{k-1} - \overline{X}_{k-1}\right)\left(X_k - \overline{X}_k\right)^T\right)$ e X_{k-1}, X_k são conjuntos de pontos 3D correspondentes.

Se as incertezas da medição dos pontos 3D são conhecidas, então podem ser adicionadas como pesos na estimativa (de acordo com Maimone et al. [41]). As transformações calculadas têm escala absoluta, e assim, a sequência da trajectória pode ser determinada directamente juntando as transformações.

O algoritmo de OV com as correspondências 3D-para-3D é resumido em seguida:

1. Capturar dois pares estéreo de imagens $I_{e,k-1}$, $I_{d,k-1}$ e $I_{e,k}$, $I_{d,k}$
2. Extrair e corresponder as *features* entre $I_{e,k-1}$ e $I_{e,k}$
3. Triangular as *features* que foram correspondidas para cada par estéreo
4. Calcular T_k a partir das *features* em 3D X_{k-1} e X_k
5. Juntar todas as transformações e calcular a pose da câmara $C_k = C_{k-1}T_k$
6. Repetir desde o passo 1

3.6 Windowed bundle adjustment

A última etapa de um sistema de OV apresenta aplicação do *Windowed bundle adjustment* [62]. O *bundle adjustment* é um algoritmo criado no campo da fotogrametria em meados da década de *1950* e tornou-se muito utilizado no campo da visão por computador, explicitamente na área de reconstrução 3D.

A função principal é tentar otimizar ao mesmo tempo os parâmetros (intrínsecos e extrínsecos) da câmara, bem como os parâmetros dos pontos 3D de referência. Ele é aplicado

para os casos em que as *features* detectadas numa imagem são procuras em mais do que duas *frames.* Este algoritmo considera uma "janela" de n *frames* da imagem e depois faz uma optimização dos parâmetros das poses da câmara e dos pontos 3D de referência para este conjunto de *frames* da imagem.

No *bundle adjustment*, a função de erro a minimizar é o erro de reprojeção da imagem:

$$arg\min_{X^i,C_k} \sum_{i,k} \left\| p_k^i - g(X^i, C_k) \right\|^2, \tag{3.6}$$

onde p_k^i é o i-ésimo ponto da imagem dos pontos 3D de referência X^i medido na k-ésima imagem e $g(X^i, C_k)$ é a sua imagem de reprojeção de acordo com a pose actual da câmara C_k. O erro de reprojeção é uma função não-linear e a optimização é geralmente efetuada usando o algoritmo LMA (do inglês *Levenberg–Marquardt algorithm*)[39].

Para concluir, o principal objectivo do *bundle adjustment* em sistemas de OV é a redução do desvio entre a trajectória estimada e a real.

3.7 RANSAC

A correspondência de pontos é muita vezes contaminada por *outliers.* As possíveis causas dos *outliers* são: tremores e ruído na imagem e mudanças no ponto de vista e iluminação.

Tal como referido anteriormente, uma das funções do algoritmo RANSAC [22] é a remoção dos *outliers*, que permite a estimação do movimento da câmara de uma forma mais precisa. A etapa de rejeição dos outliers é a mais delicada em sistemas de OV.

De uma forma geral, este algoritmo é utilizado quando se pretende estimar um modelo (por exemplo os parâmetros de rotação e translação de uma câmara) na presença de *outliers.*

A ideia fundamental do RANSAC é calcular candidatas modelo a partir de amostras de conjuntos de pontos de dados. A escolha das amostras é feita aleatoriamente. Posteriormente verifica essas candidatas em outros pontos de dados. A candidata que mostrar maior consenso com os outros dados é selecionada como solução.

O esboço deste algoritmo é apresentado em seguida:

1. Inicialmente: Seja A um conjunto de N correspondências de *features*
2. Repetir
   a) Escolhe aleatoriamente uma amostra de pontos de tamanho s a partir do A
   b) Ajusta um modelo para estes pontos
   c) Calcula a distância de todos os outros pontos para este modelo
   d) Constrói o conjunto de *inliers*[a], isto é, conta o número de pontos cuja distância com base no modelo $< d$
   e) Armazena estes *inliers*
   f) Até que o número máximo de iterações seja alcançado
3. O conjunto com o máximo número de *inliers* é escolhido como uma solução para o problema
4. Estima o modelo usando todos os *inliers*

[a] *inliers*: pontos de dados que se ajustam com um determinado modelo desejado dentro de uma certa tolerância de erro.

O número de iterações N necessário para garantir a solução correcta é calculado pela equação 3.7.

$$N = \frac{log\,(1-p)}{log\,(1-(1-\varepsilon)^s)} \tag{3.7}$$

O valor s representa o número de pontos de dados a partir do qual o modelo pode ser instanciado, ε é a percentagem de *outliers* nos pontos de dados e p representa a probabilidade de sucesso pretendida [22].

Notar que o algoritmo é um método iterativo e não determinístico de tal foma que calcula uma diferente solução em cada execução. Contudo, a solução tende a estabilizar à medida que o número de iterações aumenta.

Em [64] é apresentado um tutorial com vários exemplos que usam a *Toolbox* RANSAC do software Matlab.

4 Software RGB-D SLAM

Neste capítulo é apresentado o software (ou algoritmo) RGB-D SLAM utilizado nos percursos experimentais. Este algoritmo realiza a OV e é apropriado para sistemas que utilizem sensores RGB-D, como é o caso do dispositivo Kinect. O principal objectivo é obter uma estimativa em tempo real das poses do Kinect, *frame* após *frame*, ao longo de uma trajectória e no final obter uma estimativa global e otimizada dessa trajectória. Além disso, ele possui a capacidade para adquirir modelos 3D de cenas em ambientes interiores, representados como nuvens de pontos a cores. De modo que, consegue construir representações de ambientes interiores.

Neste trabalho o meu interesse são as poses estimadas e a trajectória global estimada, e portanto, não vou usar na prática a representação de modelos 3D de cenas.

4.1 Descrição

O software foi desenvolvido em ROS [52] pelos autores Felix Endres, Juergen Hess, Nikolas Engelhard, Juergen Sturm e Wolfram Burgard. Trata-se de um software de código aberto e com uma licença do tipo Berkeley Software Distribution (BSD) que permite a distribuição e alteração do código sem restrições. O código está disponível em [7] por via SVN (do inglês *Subversion*). Uma breve descrição do funcionamento é feita em seguida.

O RGB-D SLAM fornece um *front-end*[1] baseado na extracção de *features* visuais através dos algoritmos SURF[14] ou SIFT[40]. Depois faz a correspondência entre as *features* obtidas e usa o algoritmo RANSAC[22] para uma estimação robusta das transformações em 3D entre elas. Além disso, constrói um grafo em 3D que é depois otimizado com o g2o[2] para reduzir os erros acumulados nas poses da câmara. A acção de otimizar o grafo faz parte do *back-end*[3] do software.

Para produzir bons resultados é necessário verificar as seguintes condições:

- O movimento do agente deve ser feito a baixas velocidades para extrair o maior número de *features,* uma vez que estas são sensíveis a transformações grandes e movimentos rápidos.
- As condições de luminosidade devem ser boas.
- Manter sempre uma distância aceitável entre o Kinect e a cena, caso contrário o sensor não consegue obter informação suficiente de profundidade.

4.2 Instalação

O software RGB-D SLAM foi desenvolvido e testado apenas na distribuição GNU/Linux Ubuntu. A instalação requer várias dependências instaladas. A lista seguinte contém as de-

[1] *front-end* é um termo generalizado para referir a etapa inicial do software. Ele tem como função interagir directamente com o utilizador e processar os dados de entrada.

[2] g2o é um framework para otimização de grafos baseado em funções de erro não lineares.

[3] *back-end* é um termo generalizado para referir a etapa final do software. Ele tem como função receber os dados tratados pelo *front-end*.

pendências necessárias:

- ***O*pen Source Computer Vision (OpenCV) [16]:** É um conjunto de bibliotecas em C/C++ distribuídas gratuitamente para aplicações na área de visão. Estas bibliotecas são usadas nos algoritmos SIFT e SURF do RGB-D SLAM.
- ***Qt4* [15]:** É um sistema para o desenvolvimento de programas de interface gráfica. Ele é necessário para o funcionamento do Interface gráfico do utilizador (GUI, do inglês *Graphical User Interface)* do RGB-D SLAM.

O seguinte conjunto de instruções instala o software e assume que tem instalado o ROS Diamondback (configuração Desktop-Full). No caso de não ter instalado então deve seguir as instruções da subsecção 2.3.3.

1. **Transferir g2o**
   Escrever na consola o seguinte comando dentro da sua pasta de trabalho (*ros_workspace*):
   ```
   svn co https://code.ros.org/svn/ros-pkg/stacks/vslam/trunk/g2o
   ```
2. **Compilar o g2o e suas dependências**
   ```
   rosmake --rosdep-install
   ```
3. **Transferir o código fonte do RGB-D SLAM**
   Escrever o seguinte comando dentro da sua pasta de trabalho:
   ```
   svn co https://svn.openslam.org/data/svn/rgbdslam/trunk rgbdslam
   ```
4. **Compilar o RGB-D SLAM e suas dependências**
   ```
   rosmake --rosdep-install rgbdslam
   ```

4.3 Iniciar o software

Antes de proceder à execução do software, verifique se o Kinect está ligado ao computador (por via cabo USB 2.0) e se os drivers respectivos estão instalados. Depois de verificar estas condições então poderá iniciar o software.

O software pode ser usado com o GUI para facilitar a interação entre o utilizador e o programa, ou então, a partir de comandos introduzidos na consola (sem o GUI). Foram analisadas as duas possibilidades e a escolhida foi a primeira.

Para iniciar o ficheiro de lançamento *rgbdslam* basta escrever o seguinte comando na consola:

```
roslaunch rgbdslam kinect+rgbdslam.launch
```

Alternativamente, pode usar os nodos *openni* e *rgbdslam* separadamente, isto é:

```
roslaunch openni_camera openni_node.launch
rosrun rgbdslam rgbdslam
```

4.4 Arquitetura do software

Nesta secção é abordada uma descrição detalhada do algoritmo RGB-D SLAM. O esquema geral do mesmo é apresentado, bem como o funcionamento com o GUI.

Na secção 4.1 deste capítulo foram referidas as duas partes do software: *front-end* e *back-end.* Estas duas partes são responsáveis pelo processo da estimação da trajectória e vão ser descritas em seguida.

No *front-end*, primeiro são extraídas *features* visuais usando o detector de *features* SIFT a partir das imagens RGB capturadas pelo Kinect. Notar que podia ter sido usado o SURF em vez do SIFT. A biblioteca OpenCV é a responsável pelo funcionamento do SIFT ou outros detectores que possam ser usados. As *features* obtidas são correspondidas com as *features* extraídas das imagens RGB anteriores. As imagens de profundidade capturadas pelo Kinect são analisadas nos locais das *features* extraídas, e assim obtém-se um conjunto 3D de correspondências pontuais entre quaisquer duas *frames*. Com base na análise destas correspondências, pode-se estimar a transformação relativa da pose da câmara entre *frames* com o uso do algoritmo RANSAC. Este algoritmo é necessário porque consegue lidar com dados ruidosos e *outliers*. Uma vez que as *features* visuais não fornecem confiabilidade perfeita em relação a dados repetidos e correspondências falsas.

Os pares das transformações estimadas entre *frames* (poses do sensor) formam as arestas do grafo das poses. Porém devido a erros de estimação as arestas criam uma trajectória que na totalidade não é estável.

No *back-end* o principal objectivo é corrigir os erros de estimação do *front-end*, ou seja, tentar obter uma trajectória globalmente estável. Para esse objectivo é necessário executar rotinas de otimização baseadas em grafos, nomeadamente o uso do framework g2o. Este framework realiza uma minimização de uma função de erro não linear que pode ser representada como um grafo, como por exemplo o que é criado no *front-end*.

Geralmente a função de erro a minimizar tem a seguinte forma:

$$F(x) = \sum_{\langle i,j \rangle \epsilon \mathcal{C}} e(x_i, x_j, z_{ij})^T \Omega_{ij} e(x_i, x_j, z_{ij}) \tag{4.1}$$

$$x^* = \underset{x}{argmin} F(x) \tag{4.2}$$

Onde $x = \left(x_1^T, ..., x_n^T\right)^T$ é um vector de parâmetros, em que cada x_i representa um genérico bloco de parâmetros. z_{ij} e Ω_{ij} representam, respectivamente, a média e uma matriz com informação de uma restrição relativa aos parâmetros x_j e x_i. Finalmente, $e(x_i, x_j, z_{ij})$ é um função de cálculo do vector de erro que mede se os blocos de parâmetros x_i e x_j satisfazem a restrição z_{ij}. A função de erro é 0 quando x_i e x_j correspondem perfeitamente à restrição.

No contexto deste trabalho, cada bloco de parâmetros x_i representa uma pose da câmara, contendo o vector translação t_i e a componente de rotação α_i. A translação t_i toma forma num espaço Euclidiano e a componente de rotação α_i é representada por uma matriz 3×3 ou um vector quaternião 1×4. O trabalho [35] apresenta com maior detalhe a minimização da função de erro e também do framework g2o.

O grafo global das poses otimizado pode ser visualizado no GUI do RGB-D SLAM (Figura 4.4). A otimização de grafos de poses com dimensões reduzidas é rápida de realizar em tempo real, isto é, para cada *frame*. Se tivermos sequências muito longas (grafos de poses com dimensões grandes) o tempo de otimização aumenta significativamente, como era de esperar. No entanto, se as estimativas do movimento (método realizado no *front-end*) são confiáveis, então a otimização global não é necessária para todas as *frames*.

O esquema geral desta abordagem é apresentado na Figura 4.1. O *front-end* e o *back-end* são visíveis na figura.

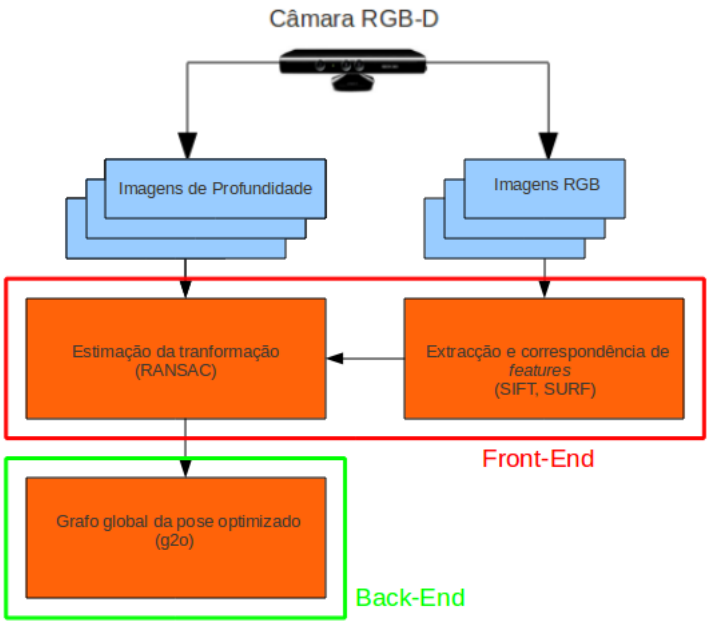

Figura 4.1 – Esquema da visão geral do algoritmo RGB-D SLAM.

Na secção 4.3 foram apresentados os passos para iniciar o programa. Agora é descrito o funcionamento do programa com o GUI.

O programa inicialmente começa com o GUI no modo "Pausa", perceptível na Figura 4.2. Este modo mostra as imagens RGB-D visualizadas pelo Kinect Fig.4.2b)c) e uma lista de comandos para o rato Fig.4.2d) para ajudar o utilizador a manusear o visualizador 3D Fig.4.2a).

O utilizador se quiser começar a gravar uma *stream*[4] contínua deve carregar no botão "space", ou se quer apenas capturar uma única *frame* então deve usar o botão "enter". No visualizador 3D está representado o sistema de coordenas do Kinect.

[4] *stream* é um fluxo de dados.

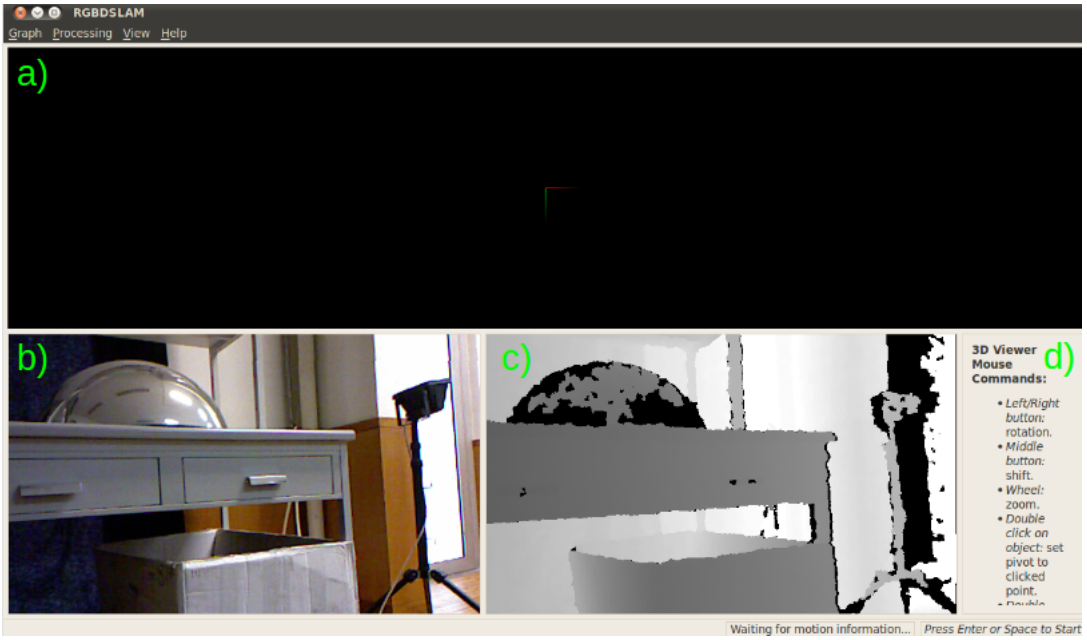

Figura 4.2 – GUI do software RGB-D SLAM. a) Visualizador 3D; b) Imagem RGB; c) Imagem de Profundidade; d) Comandos do rato para o Visualizador 3D; (Lab. de Visão por Computador no piso 0 - ISR UC)

Na Figura 4.3 é mostrado de novo o GUI, mas sem estar no modo "Pausa". Esta figura representa a captura de uma *frame* onde é possível observar diferenças no visualizador 3D Fig.4.3a) e na janela Fig.4.3b) que agora corresponde às *features* extraídas. O visualizador 3D mostra o modelo colorido 3D da cena capturada e a pose inicial da câmara.

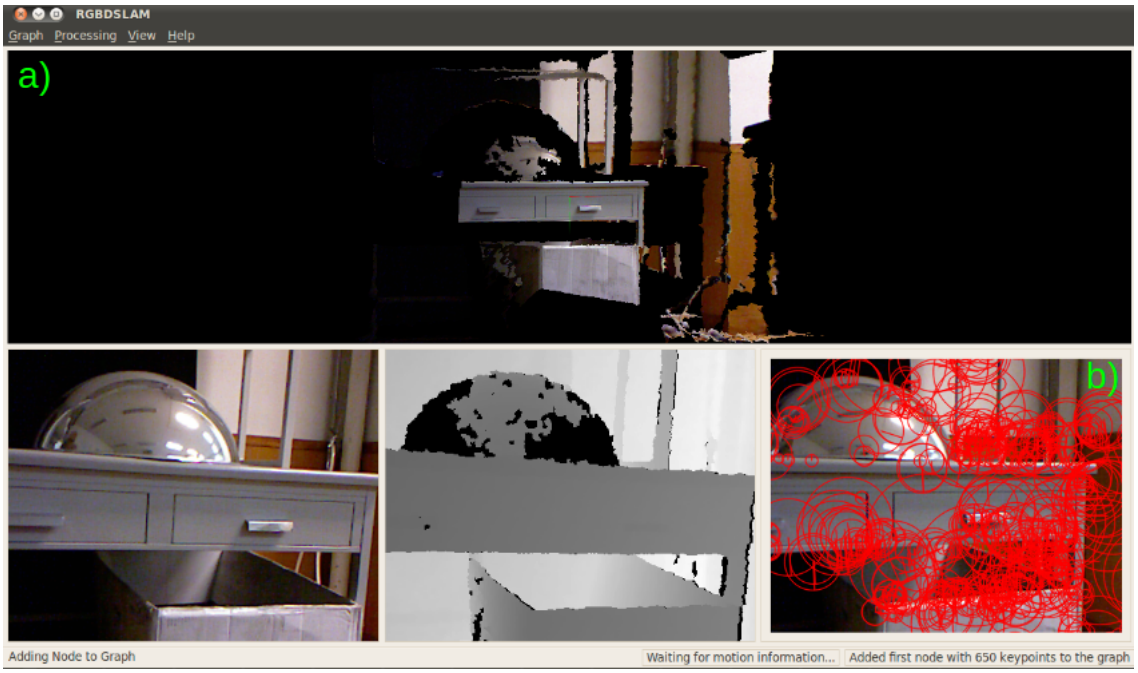

Figura 4.3 – Captura de uma *frame*. a) Visualizador 3D da *frame* capturada; b) Extracção de *features* visuais; (Lab. de Visão por Computador no piso 0 - ISR UC)

Para reduzir a redundância dos dados, as *frames* sequenciais capturadas na mesma posição não são incluídas no modelo final. O visualizador 3D corresponde sempre ao grafo global das poses otimizado. Assim é possível conhecer a trajectória global estimada do Kinect. No visualizador 3D as poses da câmara correspondem aos nós e as arestas às transformações 3D estimadas.

Na Figura 4.4 está representado um grafo das poses com três frames capturadas.

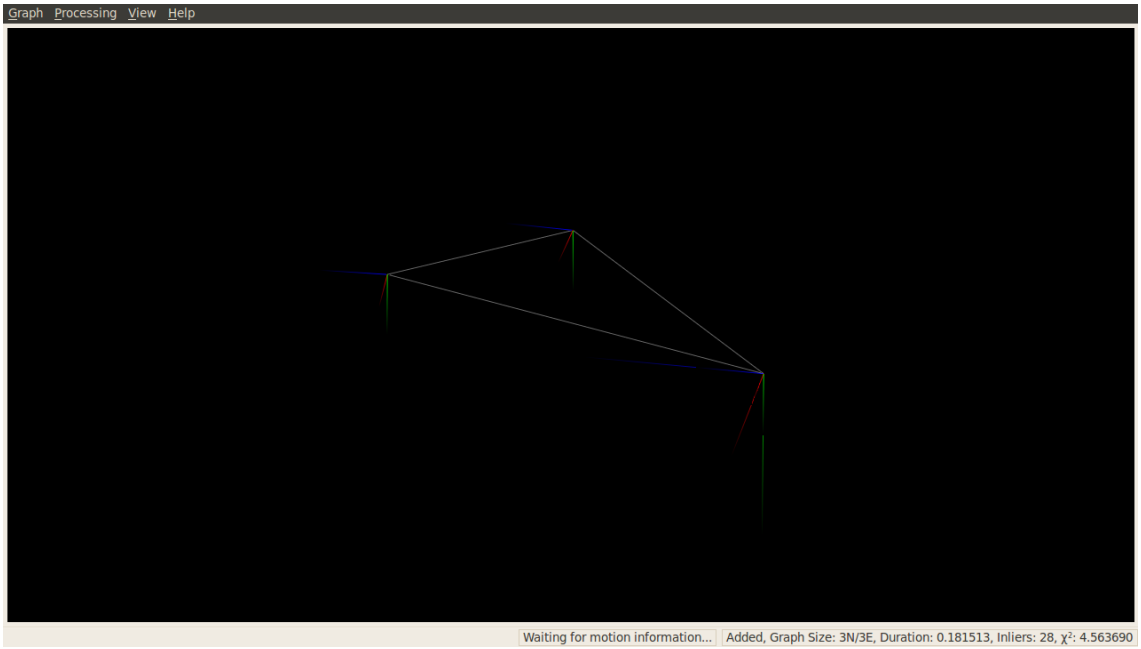

Figura 4.4 – Visualizador 3D do GUI contendo o Grafo das Poses otimizado. (Lab. de Visão por Computador no piso 0 - ISR UC)

Todas as transformações 3D efetuadas pela câmara podem ser guardadas num ficheiro de texto e o nome pode ser definido pelo utilizador. Para criar o ficheiro, primeiro encontrar o menu Processamento na parte superior do GUI e só depois escolher a opção *Save Trajectory Estimate.* O conteúdo do ficheiro de texto é da seguinte forma:

1 #TF coordinate *frame* ID:
2 nº ID *frame* T_z T_x T_y W_z W_x W_y W
3 ...

Neste ficheiro cada linha a partir da segunda tem dois vectores: o vector T_x,T_y,T_z que representa a translação da câmara e o vector quaternião W_x,W_y,W_z,W que representa a orientação (ou rotação) da câmara. A segunda linha corresponde à primeira *frame* capturada e inclui a pose inicial da câmara. As linhas seguintes correspondem às restantes *frames* capturadas, em que cada frame contém as transformações 3D da câmara em relação à pose inicial. A primeira coluna contém o número de identificação de cada *frame*. Esta coluna será ignorada porque não tem importância para este trabalho.

Os quaterniões do ficheiro são descritos por:

$$q = [q_0\, q_1\, q_2\, q_3]^T \tag{4.3}$$

$$|q|^2 = q_0^2 + q_1^2 + q_2^2 + q_3^2 = 1 \tag{4.4}$$

Pode-se associar um quaternião com uma rotação em torno de um eixo através da seguinte expressão:

$$\begin{aligned} q_0 &= cos\,(\alpha/2) \\ q1 &= sin\,(\alpha/2)\,cos\,(\beta_x) \\ q2 &= sin\,(\alpha/2)\,cos\,(\beta_y) \\ q3 &= sin\,(\alpha/2)\,cos\,(\beta_z) \end{aligned} \tag{4.5}$$

Onde α é um simples ângulo de rotação e $cos\,(\beta_x)$, $cos\,(\beta_y)$ e $cos\,(\beta_z)$ são os "cosenos da direção" localizados no eixo de rotação (Teorema de Euler).

Geralmente converte-se os quaterniões para ângulos Euler (em radianos ou rad) porque a representação no espaço euclidiano é simples e o cálculo computacional não requer muito tempo. A conversão de quaterniões para ângulos Euler (em rad) pode ser realizada com a seguinte fórmula:

$$\begin{bmatrix} \phi \\ \theta \\ \psi \end{bmatrix} = \begin{bmatrix} atan2\left(2\left(q_0q_1 + q_2q_3\right), 1 - 2\left(q_1^2 + q_2^2\right)\right) \\ arcsin\left(2\left(q_0q_2 - q_3q_1\right)\right) \\ atan2\left(2\left(q_0q_3 + q_1q_2\right), 1 - 2\left(q_2^2 + q_3^2\right)\right) \end{bmatrix} \tag{4.6}$$

A função trigonométrica *atan2* tem que ser usada para gerar todas as possíveis orientações. O ângulo ϕ (ou *Roll*) representa a rotação em torno do eixo X, o ângulo θ (ou *Pitch*) representa a rotação em torno do eixo Y e o ângulo ψ (ou *Yaw*) representa a rotação em torno do eixo Z.

No trabalho estes ângulos Euler representam as três rotações essencias da câmara. A conversão apresentada em cima (4.6) foi feita no software Matlab com o uso da função *quad2angle* e o resultado foi convertido em graus (°) com a função *radtodeg.* O Matlab também foi usado para a visualização destes ângulos em três gráficos. Desta forma, foi criado um ficheiro Matlab que faz a conversão dos ângulos, representa a trajectória real e estimada do Kinect, bem como a sua orientação ao longo do tempo. Os resultados da execução do ficheiro podem ser vistos no capítulo 5.

O sistema de coordenadas do Kinect tem que ser conhecido para analisar correctamente os dois vectores disponíveis no ficheiro de texto. Na figura 4.5 está representado o sistema de coordenadas usado.

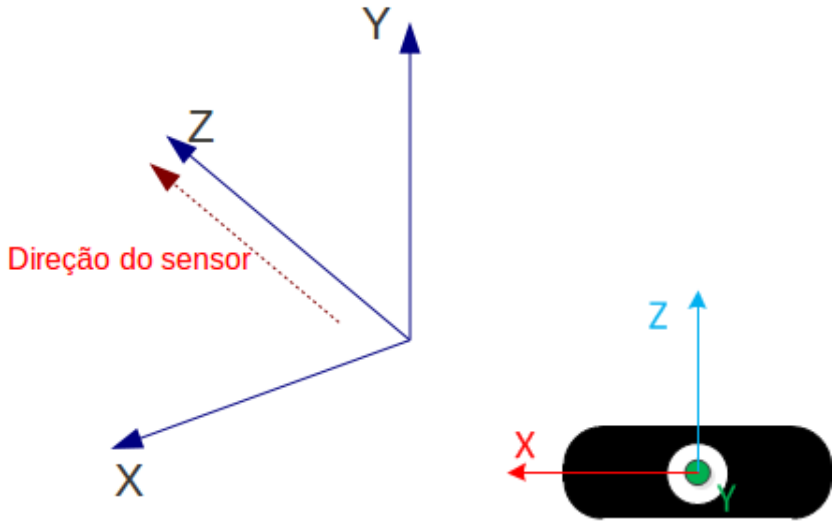

Figura 4.5 – Sistema de coordenadas do sensor Kinect.

O eixo Z é sempre perpendicular ao plano definido pelo Kinect. O eixo X está à esquerda e o eixo Y está em cima. Este sistema de coordenadas é definido pela regra da mão direita e é muito usado em aplicações com câmaras.

Se for utilizado o Kinect sem estar ligado rigidamente ao robô, isto é, movimento livre no espaço 3D, então o ficheiro de texto pode ter valores em todas as variáveis de translação e de orientação. Na prática, atendendo a que o sensor RGB-D está acoplado a um robô móvel terrestre (P3DX) e como o deslocamento do robô é num piso plano sem elevações, então só podem existir valores para T_x, T_z e θ. Assim, apenas temos rotações em torno do eixo Y e translações segundo o plano XZ.

5 Análise dos resultados experimentais

No âmbito deste trabalho, um conjunto de percursos experimentais foi efetuado no laboratório de Visão por Computador (ambiente interior). Os percursos efetuados foram: movimento em linha recta, movimento em L (semi-quadrado), percurso quadrangular, percurso retangular e percurso circular. De notar que os percursos foram realizados com velocidades baixas. Assim, a câmara consegue extrair um maior número de *features* e por consequência o erro da trajectória estimada será menor.

A realização de um percurso experimental deve seguir uma sequência de passos que serão descritos em seguida.

O primeiro passo refere-se à preparação da plataforma experimental descrita no capítulo 2. Para efetuar esta operação foi colocado no topo do robô o sensor Kinect e o computador *onboard*. Esta plataforma começa sempre no ponto inicial da trajectória e o Kinect está alinhado horizontalmente, a "olhar" em frente na direção de condução do robô, de modo a que o cenário esteja aproximadamente localizado no centro da imagem.

Depois foram feitas medições à plataforma. De modo que, o raio da roda medido é aproximadamente de 2.4 cm e a distância do eixo entre rodas é de 27.4 cm.

O segundo passo consiste na análise do *ground truth* para cada percurso, isto é, a determinação da trajectória realmente percorrida pela plataforma. Para efetuar este procedimento foi necessário a marcação de pontos de auxílio com o recurso a medições no pavimento. Nos quatro primeiros percursos apresentados, o *ground truth* é obtido por medição directa, enquanto no percurso circular é necessário calcular o raio de curvatura (R) da plataforma através da fórmula:

$$R = \frac{d}{2}\frac{V_L + V_R}{V_R - V_L} \tag{5.1}$$

Onde V_L é a velocidade da roda esquerda, V_R é a velocidade da roda direita e d é a distância do eixo entre rodas. V_L e V_R são retirados do programa MobileEyes e o d por medição.

Depois da obtenção do *ground truth*, o próximo passo aborda a configuração do modo teleoperação, referido na secção 2.1.1, para permitir o controlo manual da plataforma a partir de um computador remoto (PC *offboard*).

Em seguida iniciar o algoritmo de OV apresentado no capítulo 4. De salientar que o algoritmo RGB-D SLAM determina em tempo real, *frame* após *frame*, uma trajectória estimada do Kinect e não do robô. No entanto, como o sensor RGB-D está ligado rigidamente ao robô, então estimar a trajectória do sensor é aproximado a estimar a trajectória do rôbo, pois, a diferença entre ambos consiste numa transformação de corpo rígido (translação e rotação). Neste trabalho este cálculo não foi feito, portanto o plano do chão foi limitado ao plano XZ do referencial do Kinect em vez do usual plano XY do robô. O referencial do robô pode ser visualizado na Figura 5.1. O eixo Z é sempre perpendicular ao plano definido pelo robô e o ponto de controlo está situado no meio do eixo entre rodas.

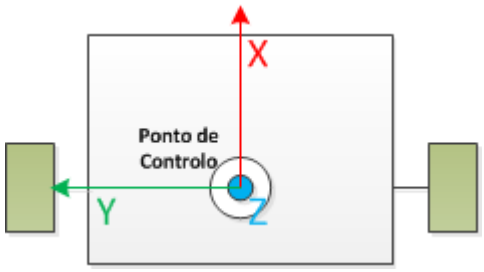

Figure 5.1 – Sistema de coordenadas do robô. (vista de cima)

O último passo consiste na realização dos percursos e consequente análise dos resultados obtidos.

Na Figura 5.2 encontra-se um diagrama de blocos com os passos descritos para realização de um percurso experimental.

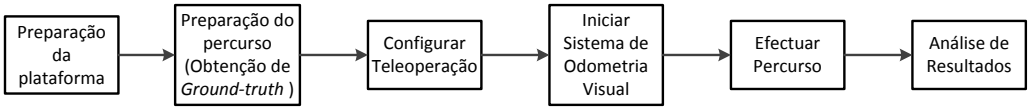

Figure 5.2 – Passos para a realização de um percurso experimental.

A avaliação do desempenho do sistema é feita por métricas criadas em Matlab, que têm a função de analisar o erro da trajectória estimada. Estas métricas vão ser explicadas na próxima secção.

5.1 Métricas de avaliação de erro

O cálculo do erro da trajectória estimada ou cálculo do erro lateral está representado na Figura 5.3. A plataforma não se encontra na trajectória real, portanto temos um erro lateral. Esta quanto mais afastada estiver em relação à trajectória real, mais significativo é o erro e vice-versa. O cálculo do erro requer duas trajectórias, uma real e outra estimada. A diferença entre as duas resulta no erro.

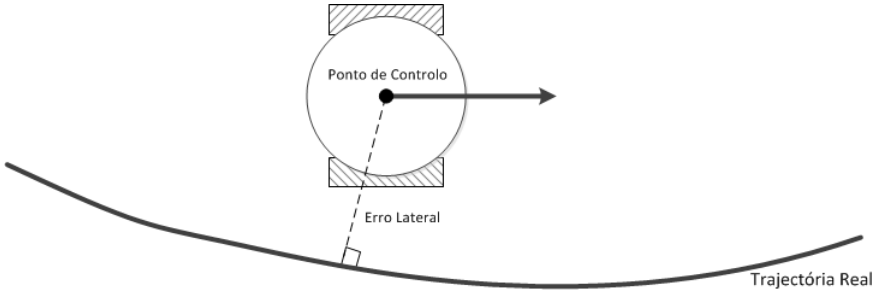

Figure 5.3 – Representação do erro da trajectória estimada.

Neste trabalho foram analisadas as seguintes métricas para avaliação do erro: máximo (máx), mínimo (min), valor quadrático médio (rms, do inglês *root mean square*), erro quadrático médio (mse) e raiz quadrada do erro quadrático médio (rmse).

Por definição, o valor quadrático médio para uma amostra de N valores $\{x_1, x_2, ..., x_N\}$ é dado pela fórmula:

$$rms = \sqrt{\frac{1}{N}\sum_{i=1}^{N} x_i^2} \tag{5.2}$$

O erro quadrático médio, em estatística, é uma forma de avaliar a diferença entre um estimador ($\hat{Y}$) e o verdadeiro valor da quantidade estimada (Y_i). Para uma amostra de N valores o mse é dado pela fórmula:

$$mse = \frac{1}{N}\sum_{i=1}^{N}\left(\hat{Y}_i - Y_i\right)^2 \tag{5.3}$$

A raiz quadrada do erro quadrático médio é uma medida da amplitude do erro e está mais sujeita a erros. Para uma amostra de N valores o rmse é dado pela fórmula:

$$rmse = \sqrt{\frac{\sum_{i=1}^{N}(\hat{Y}_i - Y_i)^2}{N}} \tag{5.4}$$

O estudo do erro permitiu averiguar quais os melhores percursos experimentais.

5.2 Resultados experimentais

Nesta secção são analisados os diferentes percursos efetuados. Em cada percurso os resultados obtidos experimentalmente são: a trajectória estimada e a orientação da plataforma.

5.2.1 Movimento em linha recta

Este movimeto é simples de materializar, consiste em percorrer aproximadamente 1m em linha recta. Os resultados experimentais deste movimento são apresentados nas Figuras 5.4 e 5.5. Da análise da Figura 5.4, é possível verificar que a trajectória estimada com o algoritmo RGB-D SLAM é semelhante à trajectória real, com um erro aproximadamente nulo (ver Figura 5.6).

A Figura 5.5 mostra os ângulos da câmara ao longo do percurso em relação aos seus três eixos de orientação X, Y e Z. A plataforma móvel executa a trajectória sem alterar a orientação da câmara, o que significa que os ângulos são praticamente nulos.

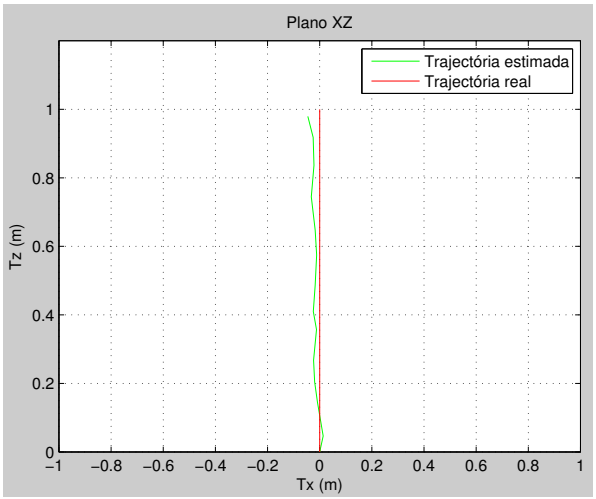

Figure 5.4 – Movimento em linha recta.

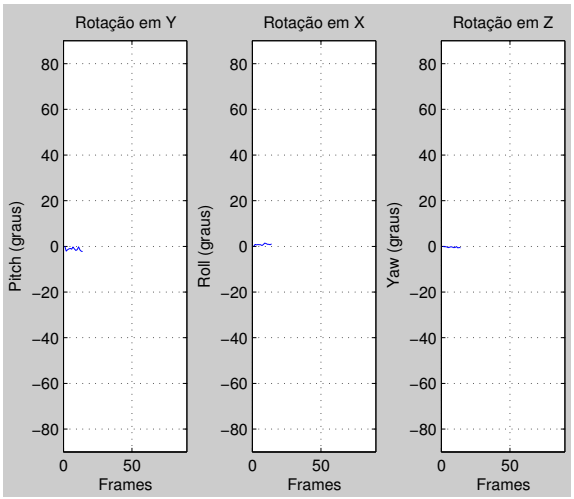

Figura 5.5 – Orientação da plataforma ao longo do percurso.

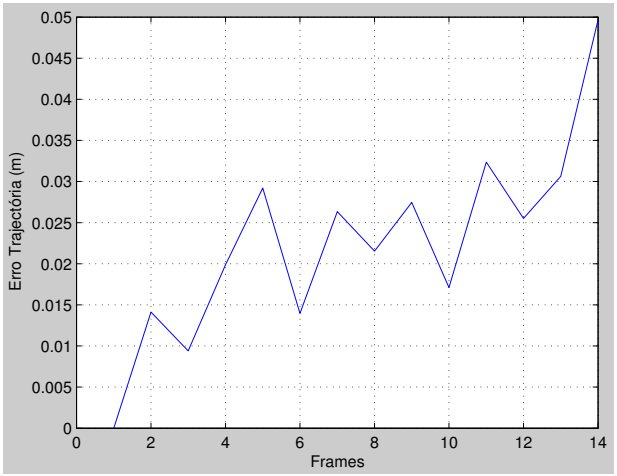

Figura 5.6 – Erro da trajectória estimada.

Com erro médio do percurso de 0.023 metros e desvio padrão de 0.012 metros.

5.2.2 Movimento em L (semi-quadrado)

Este movimento consiste em efectuar metade de um quadrado. Primeiro é percorrido 1m em frente (igual à subsecção 5.2.1), de seguida a plataforma roda 90º segundo o eixo Y no sentido positivo e finalmente percorre 1m em frente. Os resultados deste movimento são visíveis nas Figuras 5.7 e 5.8.

Na Figura 5.7 constata-se que a partir da rotação de 90º, a trajectória estimada afasta-se da trajectória real, e só tende a aproximar-se desta perto do final do movimento, havendo por isso um erro de estimação. No entanto, o erro não é muito significativo, uma vez que o seu valor máximo é aproximadamente de 0.087 m ou 8,7 cm (ver Figura 5.9), que corresponde ao pior caso.

Quanto à orientação, é visível na Figura 5.8 uma rotação aproximada de 90º segundo o eixo Y no sentido positivo. Os outros ângulos teoricamente deviam ser nulos, porém na prática não o são. Este problema vai ser explicado mais tarde no capítulo 6.

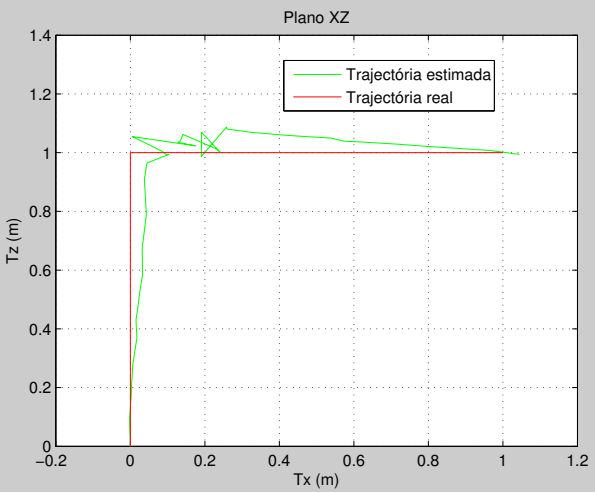

Figura 5.7 – Movimento em L.

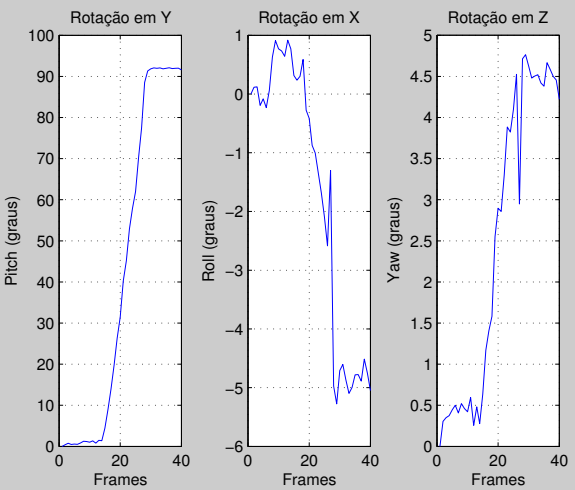

Figura 5.8 – Orientação ao longo do percurso.

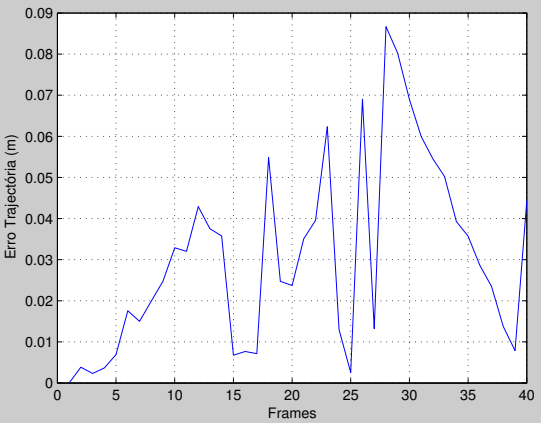

Figura 5.9 – Erro da trajectória estimada.

Com erro médio do percurso de 0.031 metros e desvio padrão de 0.023 metros.

5.2.3 Percurso quadrangular

Neste percurso é percorrido um quadrado marcado no piso de lado 1m. Os resultados experimentais são visualizados nas Figuras 5.10 e 5.11.

Da análise da Figura 5.10 é possível constatar que a curva da trajectória estimada tem um comportamento semelhante à curva da trajectória real. No entanto, num dos lados do quadrado existe uma discrepância significativa da posição, com um erro máximo aproximado de 0.25 m ou 25 cm (ver Figura 5.12).

A Figura 5.11 mostra o ângulo *pitch* a variar bruscamente. Com base neste resultado, conclui-se que a plataforma executa três rotações aproximadas de 90^o segundo o eixo Y no sentido positivo.

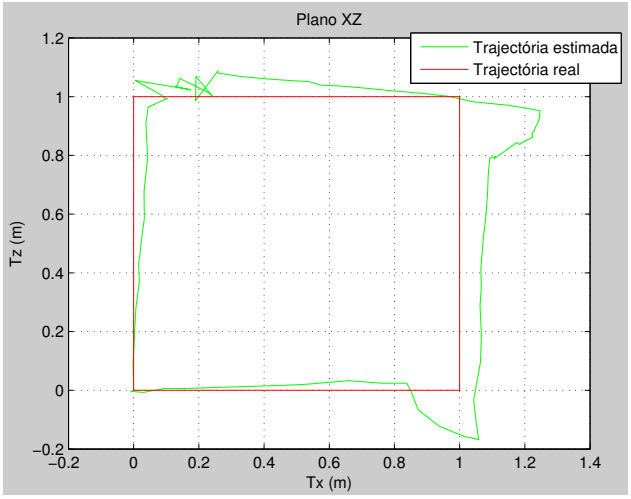

Figura 5.10 – Percurso quadrangular.

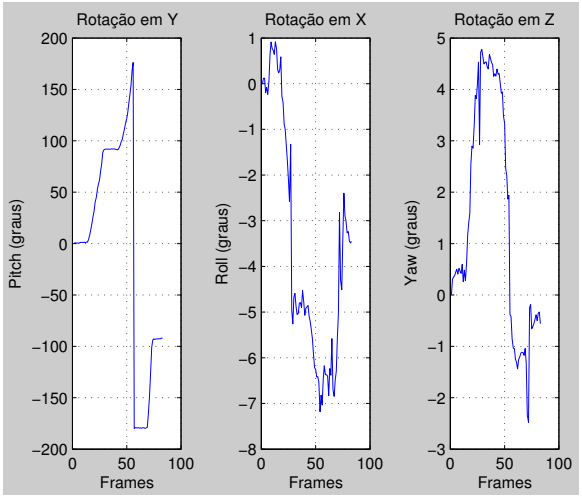

Figura 5.11 – Orientação ao longo do percurso.

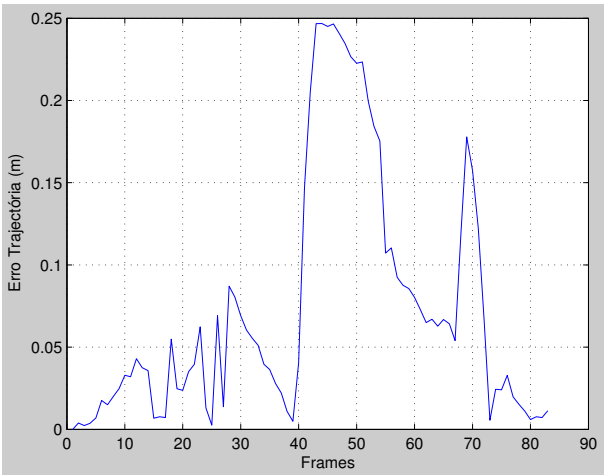

Figura 5.12 – Erro da trajectória estimada.

Com erro médio do percurso de 0.073 metros e desvio padrão de 0.075 metros.

5.2.4 Percurso retangular

Neste percurso é percorrido um retângulo marcado no piso de dimensoẽs $1 \times 1,5$ (em m). Os resultados experimentais são apresentados nas Figuras 5.13 e 5.14.

Ao analisar a Figura 5.13, é de notar uma grande divergência das posições no penúltimo troço do percurso, com um erro máximo aproximado de 0.37 m ou 37 cm (ver Figura 5.15).

A orientação deste percurso (Figura 5.14) é idêntica à orientação do percurso quadrangular (Figura 5.11). Mais uma vez a plataforma executa três rotações aproximadas de 90^o segundo o eixo Y no sentido positivo.

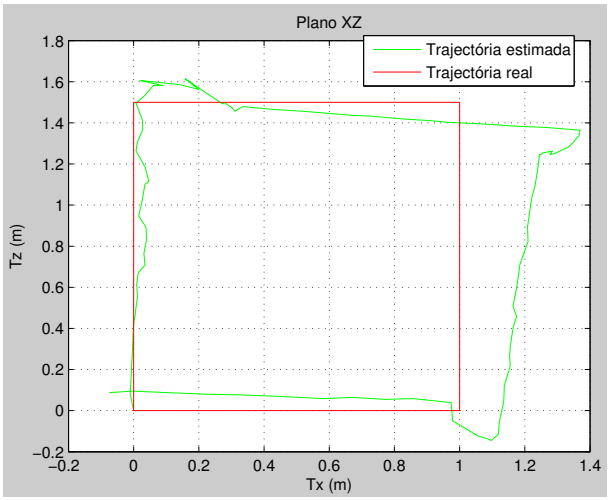

Figura 5.13 – Percurso retangular.

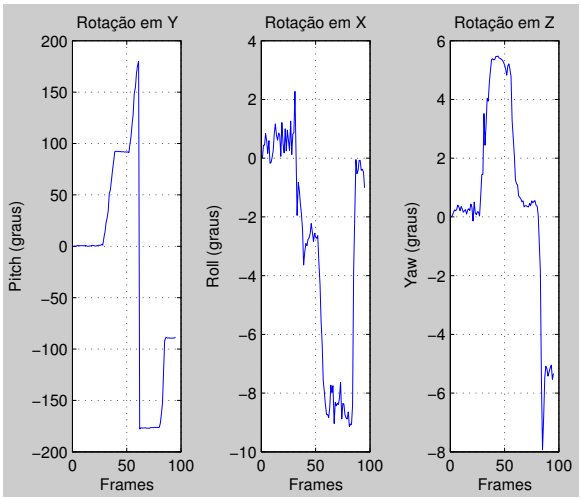

Figura 5.14 – Orientação ao longo do percurso.

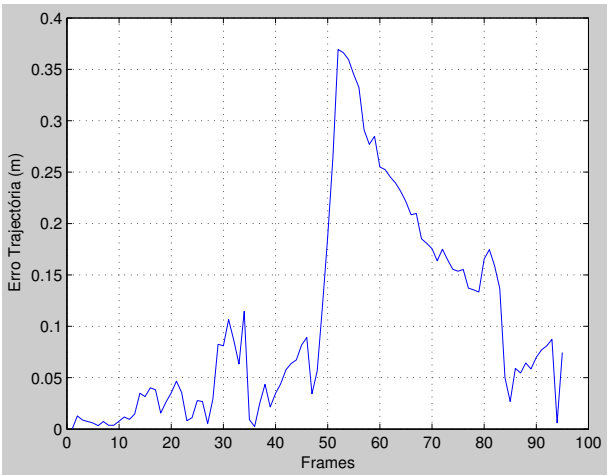

Figura 5.15 – Erro da trajectória estimada.

Com erro médio do percurso de 0.105 metros e desvio padrão de 0.100 metros.

5.2.5 Percurso circular

Neste percurso é percorrido uma circunferência com raio de curvatura (R) aproximado de 89 cm. O processo para o cálculo deste raio já foi apresentado no início do capítulo.

Os resultados experimentais são exibidos nas Figuras 5.16 e 5.17. Da análise da Figura 5.16, verifica-se que as duas trajectórias têm comportamentos diferentes. O erro máximo entre as duas trajectórias é de aproximadamente 0.4 m (visível na Figura 5.18).

A orientação pose ser vista na Figura 5.17. O ângulo *pitch* está em constante movimento, e roda sempre segundo o eixo Y no sentido positivo, perfazendo uma volta completa de 360^o.

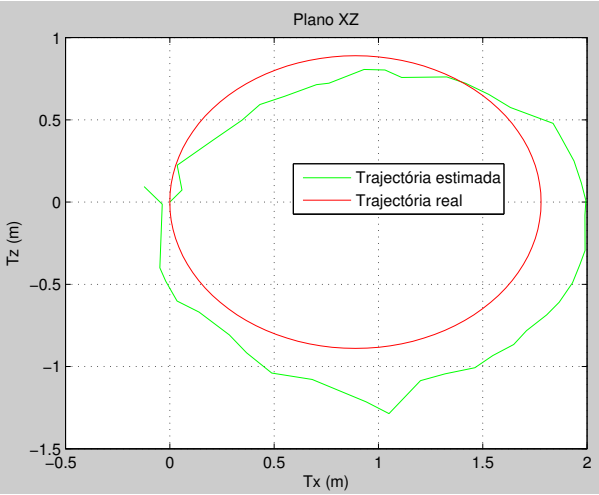

Figura 5.16 – Percurso circular.

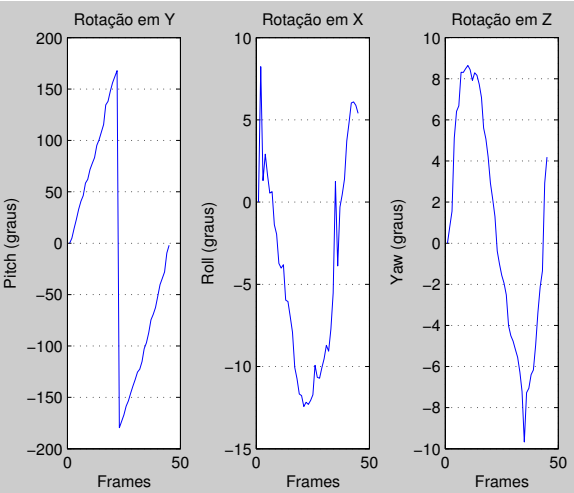

Figura 5.17 – Orientação ao longo do percurso.

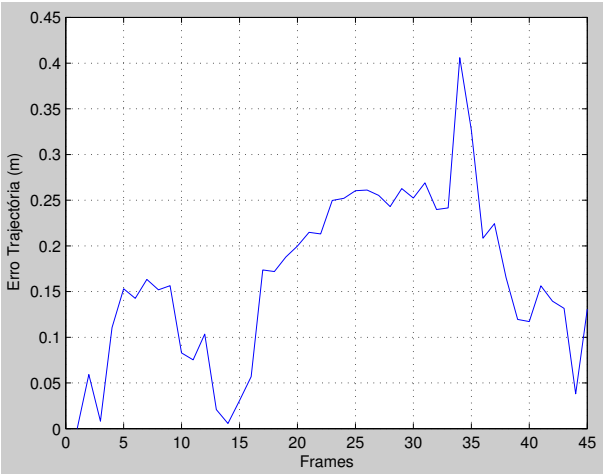

Figura 5.18 – Erro da trajectória estimada.

Com erro médio do percurso de 0.165 metros e desvio padrão de 0.091 metros.

6 Conclusões e trabalho futuro

Neste capítulo são apresentadas as conclusões do trabalho efetuado e algumas sugestões para trabalho futuro.

6.1 Conclusões

Em resumo, na tabela 6.1 estão as métricas usadas que permitiram avaliar o desempenho do algoritmo de OV nos diferentes percursos.

Percursos	máx (m)	rms (m)	rmse (m)	mse (m)
linha recta	0.050	0.025	0.012	0.00013
L (semi-quadrado)	0.087	0.038	0.023	0.00052
quadrangular	0.247	0.105	0.075	0.00557
retangular	0.369	0.145	0.099	0.00985
circular	0.406	0.188	0.090	0.00818

Tabela 6.1 – Estudo do erro para os diferentes percursos.

Da análise da Tabela 6.1, conclui-se que os resultados experimentais estão de acordo com o esperado. Na generalidade dos percursos efetuados, a pose estimada pelo algoritmo de OV descreve o comportamento da trajectória real. O erro máximo (máx) permitiu saber o pior caso para cada percurso. Com base neste erro, verificou-se que o percurso circular apresenta os piores resultados, ou seja, é o percurso onde o algoritmo de OV apresenta o pior desempenho, tendo um erro máximo de 0.406 metros. Verificou-se também que o movimento em linha recta apresenta os melhores resultados, pois o erro máximo é pouco significativo, cerca de 0.050 metros. Isto é devido às características do algoritmo de OV, que na presença de percursos lineares apresenta uma maior percentagem de sucesso no processo correspondência das *features*.

O erro máximo não é suficiente para classificar o desempenho do algoritmo, isto porque o erro máximo é susceptível a *outliers*. Pelo que as outras medidas apresentadas (rms, rmse e mse) foram introduzidas para auxiliar na avaliação.

Da analise da Figura 5.16, constata-se que o erro piora significativamente se a plataforma tiver que percorrer curvas. A minimização deste erro é possível com o uso do algoritmo *bundle adjustment*, que permite a correção das poses da câmara de forma iterativa, tornando-as mais precisas. Porém a adição desta etapa ao sistema apresenta algumas desvantagens, como por exemplo o aumento do custo computacional.

A orientação da plataforma é calculada em todos os percursos ao longo da trajectória e apresenta bons resultados. Tanto o *roll* como o *yaw* apresentam valores baixos, não nulos mas próximos de zero. Estas variações são devidas a tremores aleatórios que podem acontecer devido às irregularidades do solo (piso escorregadio, e existência de cabos e outros obstáculos inesperados) que em conjunto ou individualmente são suscetíveis de prejudicar o movimento, isto é, de desviar a orientação do Kinect e consequentemente afectar as imagens capturadas.

O algoritmo mostrou ter qualidade para complementar a informação da Odometria das rodas. Porém, em termos computacionais demonstrou ser lento. Quanto maior o percurso, maior o número de *frames* capturadas, e por consequência maior o tempo de computação e vice-versa. Isto acontece porque ele precisa de mais tempo para processar as imagens, isto é, para estabelecer correspondências entre todas as *features*.

O robô P3DX usado nos percursos experimentais revelou ter grandes capacidades para ser utilizado noutros trabalhos, devido às seguintes características: estabilidade, robustez, peso e mobilidade. Além disto, permite a integração de outros sensores ligados a ele.

O dispositivo Kinect mostrou ser uma grande valia em aplicações robóticas que exigem permanente monitorização da pose do robô.

Em conclusão, o algoritmo de OV apresenta bons resultados e ainda pode ser melhorado com pequenos ajustes, como por exemplo o uso do algoritmo *bundle adjustment*. Na generalidade dos percursos lineares mostrou ser capaz de estimar a pose, apresentando erros baixos. Uma das suas limitações são os percursos que contêm curvas uma vez que o erro aumenta significativamente.

6.2 Trabalho futuro

Na sequência desta dissertação, são apresentadas algumas sugestões para trabalho futuro:

- Integração de um laser rangefinder na plataforma (por exemplo o Hokuyo URG-04LX [8]), permitindo assim mitigar as limitações do sensor 3D Kinect. Deste modo, a combinação dos dois (fusão sensorial) permite a captura de maior quantidade de informação e a estimação da pose torna-se mais robusta.
- Avaliação do desempenho do algoritmo RGB-D SLAM em ambientes externos estruturados. Porventura, realizar os mesmos percursos com a mesma plataforma. Posteriormente, comparar o desempenho do algoritmo entre dois ambientes (interior e exterior).
- Utilizar o algoritmo detector de *features* SURF em vez do SIFT e comparar o desempenho dos dois.
- Utilizar os encoders que estão incorporados no robô P3DX para efectuar a fusão sensorial, de modo a introduzir uma maior redundância na estimação da posição do robô ao longo da trajectória. A fusão sensorial é realizada com base no filtro de Kalman estendido (EKF, do inglês *extended Kalman filter*).
- Testar outros algoritmos existentes de OV, como por exemplo, o LIBVISO2 [4]. Esta biblioteca permite substituir o sensor Kinect por uma câmara com visão monocular ou visão estéreo.

Bibliografia

[1] Adept mobilerobots pioneer 3-dx (p3dx) differential drive robot for research and education. Disponível em `http://www.mobilerobots.com/researchrobots/pioneerp3dx.aspx`.

[2] Blob detection. Disponível em `http://www.nada.kth.se/~tony/cern-review/cern-html/node19.html`.

[3] File:corner.png - wikipedia, the free encyclopedia. Disponível em `http://en.wikipedia.org/wiki/File:Corner.png`.

[4] Libviso2: C++ library for visual odometry 2. Disponível em `http://www.cvlibs.net/software/libviso2.html`.

[5] Pioneer3dx datasheet. Disponível em `http://www.mobilerobots.com/Libraries/Downloads/Pioneer3DX-P3DX-RevA.sflb.ashx`.

[6] Repositories/ubuntu - community ubuntu documentation. Disponível em `http://help.ubuntu.com/community/Repositories/Ubuntu`.

[7] Rgbdslam. Disponível em `http://openslam.org/rgbdslam.html`.

[8] Scanning laser range finder urg-04lx specifications. Disponível em `http://www.hokuyo-aut.jp/02sensor/07scanner/download/products/urg-04lx/data/URG-04LX_spec_en.pdf`.

[9] Ubuntu install of diamondback. Disponível em `http://mediabox.grasp.upenn.edu/roswiki/diamondback(2f)Installation(2f)Ubuntu.html`.

[10] Webots: robot simulator - documentation - user guide. Disponível em `http://www.cyberbotics.com/guide/section8.3.php`.

[11] S. Ameling. An introduction to local feature extraction, description and tracking in matching applications.

[12] K.S. Arun, T.S. Huang, and S.D. Blostein. Least-squares fitting of two 3-d point sets. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (5):698–700, 1987.

[13] K. Baker. Singular value decomposition tutorial. *200S. Disponível em www. ling. ohio-state. edul-kbaker/... ISingular Value Decomposition Tutorial. pdf*, 2005.

[14] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *Computer Vision–ECCV 2006*, pages 404–417, 2006.

[15] Jasmin. Blanchette and Mark. Summerfield. *C++ GUI programming with Qt 4*. Prentice Hall.

[16] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O'Reilly Media, Incorporated, 2008.

[17] A.I. Comport, E. Malis, and P. Rives. Accurate quadrifocal tracking for robust 3d visual odometry. In *Robotics and Automation, 2007 IEEE International Conference on*, pages 40–45. IEEE, 2007.

[18] P. Corke, D. Strelow, and S. Singh. Omnidirectional visual odometry for a planetary rover. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 4, pages 4007–4012. IEEE, 2004.

[19] F. Dellaert, S.M. Seitz, C.E. Thorpe, and S. Thrun. Structure from motion without correspondence. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 557–564. IEEE, 2000.

[20] F. Endres, J. Hess, N. Engelhard, J. Sturm, D. Cremers, and W. Burgard. An evaluation of the rgb-d slam system. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 1691–1696. IEEE, 2012.

[21] M. Fiala and A. Ufkes. Visual odometry using 3-dimensional video input. In *Computer and Robot Vision (CRV), 2011 Canadian Conference on*, pages 86–93. IEEE, 2011.

[22] M.A. Fischler and R.C. Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[23] W. Förstner. A feature based correspondence algorithm for image matching. *International Archives of Photogrammetry and Remote Sensing*, 26(3):150–166, 1986.

[24] J.M. Frahm, P. Fite-Georgel, D. Gallup, T. Johnson, R. Raguram, C. Wu, Y.H. Jen, E. Dunn, B. Clipp, S. Lazebnik, et al. Building rome on a cloudless day. *Computer Vision–ECCV 2010*, pages 368–381, 2010.

[25] F. Fraundorfer, D. Scaramuzza, and M. Pollefeys. A constricted bundle adjustment parameterization for relative scale estimation in visual odometry. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1899–1904. IEEE, 2010.

[26] R. Goecke, A. Asthana, N. Pettersson, and L. Petersson. Visual vehicle egomotion estimation using the fourier-mellin transform. In *Intelligent Vehicles Symposium, 2007 IEEE*, pages 450–455. IEEE, 2007.

[27] M.J. Hannah. Computer matching of areas in stereo images. Technical report, DTIC Document, 1974.

[28] P.M. Hans. Towards automatic visual obstacle avoidance. In *Proceedings of the 5th international joint conference on Artificial intelligence*, pages 584–584, 1977.

[29] C. Harris and M. Stephens. A combined corner and edge detector. In *Alvey vision conference*, volume 15, page 50. Manchester, UK, 1988.

[30] CG Harris and JM Pike. 3d positional integration from image sequences. *Image and Vision Computing*, 6(2):87–90, 1988.

[31] R. Hartley and A. Zisserman. *Multiple view geometry in computer vision*, volume 2. Cambridge Univ Press, 2000.

[32] A. Howard. Real-time stereo visual odometry for autonomous ground vehicles. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 3946–3952. IEEE, 2008.

[33] B. Kitt, A. Geiger, and H. Lategahn. Visual odometry based on stereo image sequences with ransac-based outlier rejection scheme. In *Intelligent Vehicles Symposium (IV), 2010 IEEE*, pages 486–492. IEEE, 2010.

[34] T. Kühn. The kinect sensor platform. *Advances In Media Technology*, page 1, 2011.

[35] R. Kummerle, G. Grisetti, H. Strasdat, K. Konolige, and W. Burgard. G< sup> 2</sup> o: A general framework for graph optimization. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3607–3613. IEEE, 2011.

[36] S. Lacroix, A. Mallet, R. Chatila, and L. Gallo. Rover self localization in planetary-like environments. In *Artificial Intelligence, Robotics and Automation in Space*, volume 440, page 433, 1999.

[37] T. Lemaire, C. Berger, I.K. Jung, and S. Lacroix. Vision-based slam: Stereo and monocular approaches. *International Journal of Computer Vision*, 74(3):343–364, 2007.

[38] M. Lhuillier. Automatic structure and motion using a catadioptric camera. In *Proceedings of the 6th Workshop on Omnidirectional Vision, Camera Networks and Non-Classical Cameras*, 2005.

[39] M.I.A. Lourakis. A brief description of the levenberg-marquardt algorithm implemented by levmar. *Institute of Computer Science, Foundation for Research and Technology*, 11, 2005.

[40] D.G. Lowe. Object recognition from local scale-invariant features. In *Computer Vision, 1999. The Proceedings of the Seventh IEEE International Conference on*, volume 2, pages 1150–1157. Ieee, 1999.

[41] M. Maimone, Y. Cheng, and L. Matthies. Two years of visual odometry on the mars exploration rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.

[42] L. Matthies and S. Shafer. Error modeling in stereo navigation. *Robotics and Automation, IEEE Journal of*, 3(3):239–248, 1987.

[43] L.H. Matthies and S.A. Chairman-Shafer. Dynamic stereo vision. 1989.

[44] A. Milella and R. Siegwart. Stereo-based ego-motion estimation using pixel tracking and iterative closest point. In *Computer Vision Systems, 2006 ICVS'06. IEEE International Conference on*, pages 21–21. IEEE, 2006.

[45] H.P. Moravec. Obstacle avoidance and navigation in the real world by a seeing robot rover. Technical report, DTIC Document, 1980.

[46] E. Mouragnon, M. Lhuillier, M. Dhome, F. Dekeyser, and P. Sayd. Real time localization and 3d reconstruction. In *Computer Vision and Pattern Recognition, 2006 IEEE Computer Society Conference on*, volume 1, pages 363–370. IEEE, 2006.

[47] D. Nistér. An efficient solution to the five-point relative pose problem. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(6):756–770, 2004.

[48] D. Nistér, O. Naroditsky, and J. Bergen. Visual odometry. In *Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on*, volume 1, pages I–652. IEEE, 2004.

[49] C.F. Olson, L.H. Matthies, H. Schoppers, and M.W. Maimone. Robust stereo ego-motion for long distance navigation. In *Computer Vision and Pattern Recognition, 2000. Proceedings. IEEE Conference on*, volume 2, pages 453–458. IEEE, 2000.

[50] C.F. Olson, L.H. Matthies, M. Schoppers, and M.W. Maimone. Rover navigation using stereo ego-motion. *Robotics and Autonomous Systems*, 43(4):215–229, 2003.

[51] A. Pretto, E. Menegatti, and E. Pagello. Omnidirectional dense large-scale mapping and navigation based on meaningful triangulation. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 3289–3296. IEEE, 2011.

[52] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, 2009.

[53] M. ROBOTS. Operations manual. 1999.

[54] D. Scaramuzza, F. Fraundorfer, and R. Siegwart. Real-time monocular visual odometry for on-road vehicles with 1-point ransac. In *Robotics and Automation, 2009. ICRA'09. IEEE International Conference on*, pages 4293–4299. IEEE, 2009.

[55] A. Segal, D. Haehnel, and S. Thrun. Generalized-icp. In *Proc. of Robotics: Science and Systems (RSS)*, volume 25, pages 26–27, 2009.

[56] J. Shi and C. Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, pages 593–600. IEEE, 1994.

[57] J. Smisek, M. Jancosek, and T. Pajdla. 3d with kinect. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 1154–1160. IEEE, 2011.

[58] F. Steinbrucker, J. Sturm, and D. Cremers. Real-time visual odometry from dense rgb-d images. In *Computer Vision Workshops (ICCV Workshops), 2011 IEEE International Conference on*, pages 719–722. IEEE, 2011.

[59] N. Sünderhauf, K. Konolige, S. Lacroix, and P. Protzel. Visual odometry using sparse bundle adjustment on an autonomous outdoor vehicle. *Autonome Mobile Systeme 2005*, pages 157–163, 2006.

[60] J.P. Tardif, Y. Pavlidis, and K. Daniilidis. Monocular visual odometry in urban environments using an omnidirectional camera. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, pages 2531–2538. IEEE, 2008.

[61] S. Thrun, M. Montemerlo, H. Dahlkamp, D. Stavens, A. Aron, J. Diebel, P. Fong, J. Gale, M. Halpenny, G. Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *The 2005 DARPA Grand Challenge*, pages 1–43, 2007.

[62] B. Triggs, P. McLauchlan, R. Hartley, and A. Fitzgibbon. Bundle adjustment - a modern synthesis. *Vision algorithms: theory and practice*, pages 153–177, 2000.

[63] C. Urmson, J. Anhalt, D. Bagnell, C. Baker, R. Bittner, MN Clark, J. Dolan, D. Duggins, T. Galatali, C. Geyer, et al. Autonomous driving in urban environments: Boss and the urban challenge. *Journal of Field Robotics*, 25(8):425–466, 2008.

[64] M. Zuliani. Ransac for dummies. Disponível em `http://vision,ece.ucsb.edu/-zuliani/Research/RANSAC/docs`.

Índice Remissivo