UNIVERSIDADE ESTADUAL DO CEARÁ

CENTRO DE CIÊNCIAS E TECNOLOGIA

PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

MESTRADO ACADÊMICO EM CIÊNCIA DA COMPUTAÇÃO

AFONSO HENRIQUES FONTES NETO SEGUNDO

UM ALGORITMO PARA LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS COM UMA CÂMERA RGB-D.

FORTALEZA – CEARÁ

2017

AFONSO HENRIQUES FONTES NETO SEGUNDO

UM ALGORITMO PARA LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS COM UMA CÂMERA RGB-D.

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Orientador: Prof. Dr. José Everardo Bessa Maia

FORTALEZA – CEARÁ

2017

Dados Internacionais de Catalogação na Publicação

Universidade Estadual do Ceará

Sistema de Bibliotecas

Segundo, Afonso Henriques Fontes Neto.
UM ALGORITMO PARA LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS COM UMACÂMERA RGB-D. [recurso eletrônico] / Afonso Henriques Fontes Neto Segundo. - 2017.
1 CD-ROM: il.; 4 ¾ pol.

CD-ROM contendo o arquivo no formato PDF do trabalho acadêmico com 71 folhas, acondicionado em caixa de DVD Slim (19 x 14 cm x 7 mm).

Dissertação (mestrado acadêmico) - Universidade Estadual do Ceará, Centro de Ciências e Tecnologia, Mestrado Acadêmico em Ciência da Computação, Fortaleza, 2017.
Área de concentração: Ciência da Computação.
Orientação: Prof. Dr. José Everardo Bessa Maia.

1. SLAM. 2. Odometria. 3. RGB-D. I. Título.

AFONSO HENRIQUES FONTES NETO SEGUNDO

UM ALGORITMO PARA LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEOS COM UMA CÂMERA RGB-D.

Dissertação apresentada ao Curso de Mestrado Acadêmico em Ciência da Computação do Programa de Pós-Graduação em Ciência da Computação do Centro de Ciências e Tecnologia da Universidade Estadual do Ceará, como requisito parcial à obtenção do título de mestre em Ciência da Computação. Área de Concentração: Ciência da Computação

Aprovada em: 03 de Fevereiro de 2017

BANCA EXAMINADORA

Prof. Dr. José Everardo Bessa Maia
(Orientador/UECE)

Prof. Dr. George Andre Pereira The
(UFC)

Prof. Dr. Thelmo Pontes de Araújo
(UECE)

À minha esposa, por sua capacidade de acreditar e investir em mim. Carol, seu cuidado e dedicação foi que deram a força para seguir, sua presença significou segurança e certeza de que não estou sozinho nessa caminhada.

RESUMO

Atualmente, robôs móveis são capazes de realizar tarefas complexas de forma autônoma, enquanto que no passado, a interação humana era uma necessidade. Diversas áreas são beneficiadas por estes avanços, como militar, médica, espacial, entretenimento e inclusive a doméstica. Nessas aplicações, são esperados robôs móveis para realizar tarefas complicadas que requerem navegação em ambientes interiores e exteriores complexos e dinâmicos, sem qualquer intervenção humana. Para a execução de tarefas com precisão, muitas aplicações relevantes em robótica e visão computacional requerem a capacidade de adquirir modelos do ambiente e estimar a pose do robô neste modelo. Para navegar corretamente em um ambiente desconhecido, um autômato precisa saber a sua localização no mundo. Isto requer o modelo, ou mapa, do ambiente. Construir o modelo, por sua vez, requer a posição do robô, portanto ambos devem ser estimados simultaneamente. Esta técnica é chamada de SLAM (*Simultaneous Localization and Mapping*). De uma forma geral, algoritmos de SLAM são compostos por duas etapas principais: previsão e atualização. A primeira estima localização e mapeamento a partir dos deslocamentos de pose fornecidos pela odometria, enquanto a atualização identifica e corrige esta estimativa através da identificação de pontos de referência no ambiente (*Landmarks*).

Palavras-chave: SLAM, Odometria, RGB-D.

ABSTRACT

Currently, mobile robots are capable of performing complex tasks autonomously, whereas in the past, human interaction was a necessity. Several areas are benefited by these advances, such as military, medical, space, entertainment and even domestic. In these applications, mobile robots are expected to perform complicated tasks requiring navigation in complex and dynamic indoor and outdoor environments without any human intervention. To perform tasks accurately, many applications relevant to robotics and computer vision require the ability to acquire models of the environment and estimate the robot's pose in this model. To navigate properly in an unfamiliar environment, an automaton needs to know its location in the world. This requires the model, or map, of the environment. Building the model, in turn, requires the position of the robot, so both must be estimated simultaneously. This technique is called SLAM (Simultaneous Localization and Mapping). In general, SLAM algorithms are composed of two main steps: prediction and update. The first estimate location and mapping from the pose displacements provided by the odometry, while the update identifies and corrects this estimate by identifying landmarks in the environment. An accordion effect is generated because odometry is a ruinous process, requiring that each inaccurate prediction be corrected shortly thereafter. The problem of SLAM has been frequently studied and several techniques have been proposed to solve it. This work reports the implementation of a Visual SLAM approach using only information from an RGB-D camera navigating in a static environment. Data from a Kinect 360 RGB-D camera made available to the public by Sturm (STURM et al., 2012) were used for performance testing and evaluation. The results show that this approach is feasible and promising.

Keywords: SLAM, Odometry, RGB-D.

LISTA DE ILUSTRAÇÕES

LISTA DE ABREVIATURAS

SLAM	*Simultaneous Localization and Mapping*
VO	*Visual Odometry*
V − SLAM	SLAM Visual
SURF	*Speeded up Roboust Features*
RANSAC	*Random Sample Consensus*
SVD	*Single Value Decomposition*
SfM	*Structure From Motion*
KF	*Kalman Filter*
EKF	*Extended Kalman Filter*
PF	*Particle Filter*
ICP	*Iteractive Closest Point*
MCL	*Monte Carlo Localization*
RMSE	*Root Mean Square Error*

SUMÁRIO

1 INTRODUÇÃO

Navegação autônoma é uma habilidade fundamental para robôs móveis, seja ela em espaços abertos ou fechados, movimentando-se por rodas, patas, voando ou submergindo. Implementar uma lógica de autonomia robusta e eficaz é um desafio, pois requer a integração de uma variedade de processos. Dentre eles, pode-se dizer que a etapa mais importante é a tomada de uma decisões baseada em informações fornecidas pelo conjunto de sensores.

Para a execução de tarefas com precisão, muitas aplicações relevantes em robótica e visão computacional requerem a capacidade de adquirir modelos do ambiente e estimar a pose do robô neste modelo. Para navegar corretamente em um ambiente desconhecido, um autômato precisa saber a sua localização no mundo. Isto requer o modelo, ou mapa, do ambiente. Construir o modelo, por sua vez, requer a posição do robô, portanto ambos devem ser estimados simultaneamente. Esta técnica é chamada de SLAM (*Simultaneous Localization and Mapping*). De uma forma geral, algoritmos de SLAM são compostos por duas etapas principais: previsão e atualização. A primeira estima localização e mapeamento a partir dos deslocamentos de pose fornecidos pela odometria, enquanto a atualização identifica e corrige esta estimativa através da identificação de pontos de referência no ambiente (*Landmarks*). Note que, devido a odometria ser um processo ruidoso, um efeito sanfona é gerado, pois cada previsão imprecisa é corrigida em seguida pela etapa de atualização.

O conteúdo deste trabalho foi organizado em quatro capítulos, incluindo o atual. Esta introdução é um capítulo síntese que apresenta uma visão completa do trabalho. Ainda nesta Introdução será apresentada uma revisão conceitual das abordagens para o problema SLAM procurando destacar as características diferenciadoras de cada abordagem e os critérios de caracterização de uma boa solução. Após isso, são apresentadas as contribuições e abordagens mais influentes para este trabalho com o objetivo de localizá-lo no contexto, fornecendo um embasamento teórico sobre o assunto. Em seguida, a abordagem adotada e os critérios de avaliação são descritos. Um algoritmo de SLAM geralmente é uma combinação sequenciada de algoritmos básicos. Definições relevantes como a de representação 3D densa de uma cena e Odometria Visual são formalmente apresentados.

O Capítulo 2, **Algoritmos**, discorre sobre detalhes intrínsecos da formulação de cada etapa do problema. Além da formulação, será apresentado como os dados de sensoriamento são tratados em cada etapa do processamento, assim como as formulações matemáticas e algoritmos.

Por fim, os resultados dos experimentos realizados são apresentados no Capítulo

3, **Resultados e discussões**. A qualidade final do algoritmo de SLAM adotado é fortemente dependente da qualidade do passo de odometria, por isso, este capítulo apresentará resultados de experimentos experimentos específicos para odometria visual para em seguida apresentar os resultados com o algoritmo de SLAM. Ambos realizados em dados de *benchmarks* públicos. A dissertação é concluída no capítulo 4.

1.1 OBJETIVOS

Geral: Descrever, programar e avaliar um algoritmo de SLAM Geométrico baseado em Odometria Visual RGB-D.

Específicos:

a) Revisar a literatura estado da arte em SLAM Visual.
b) Propor uma estrutura de solução para o problema de SLAM e justificar sua racionalidade.
c) Programar, avaliar e interpretar resultados de um algoritmo de Odometria Visual utilizando dados de *benchmarks* publicamente disponíveis.
d) Programar, avaliar e interpretar de um algoritmo de SLAM Visual utilizando dados de *benchmarks* publicamente disponíveis.

1.2 DEFINIÇÃO DO PROBLEMA

1.2.1 Odometria Visual

Em robótica e em visão computacional, Odometria Visual (VO) é o processo de determinação de informações de deslocamento utilizando imagens sequenciais de uma câmera em movimento para estimar a distância percorrida. Como ilustra a Figura 1, VO tem sido utilizado em dois principais campos: reconstrução 3D (a) e medição de deslocamento (b), pois esta permite maior precisão na navegação em robôs ou veículos usando qualquer tipo de locomoção em qualquer superfície (NISTÉR et al., 2004).

Matematicamente (NISTÉR et al., 2004), o problema de Odometria Visual pode ser resolvido em duas etapas. Primeiramente um algoritmo deve ser capaz de encontrar um conjunto de pontos correspondentes, p e q, em duas imagens consecutivas, como na Figura 2 (a). Em seguida, considerando que a câmera realizou um movimento rígido, como no item (b) da Figura 2, deve-se encontrar uma rotação ($R*$) e uma translação ($t*$) que minimize a diferença entre os

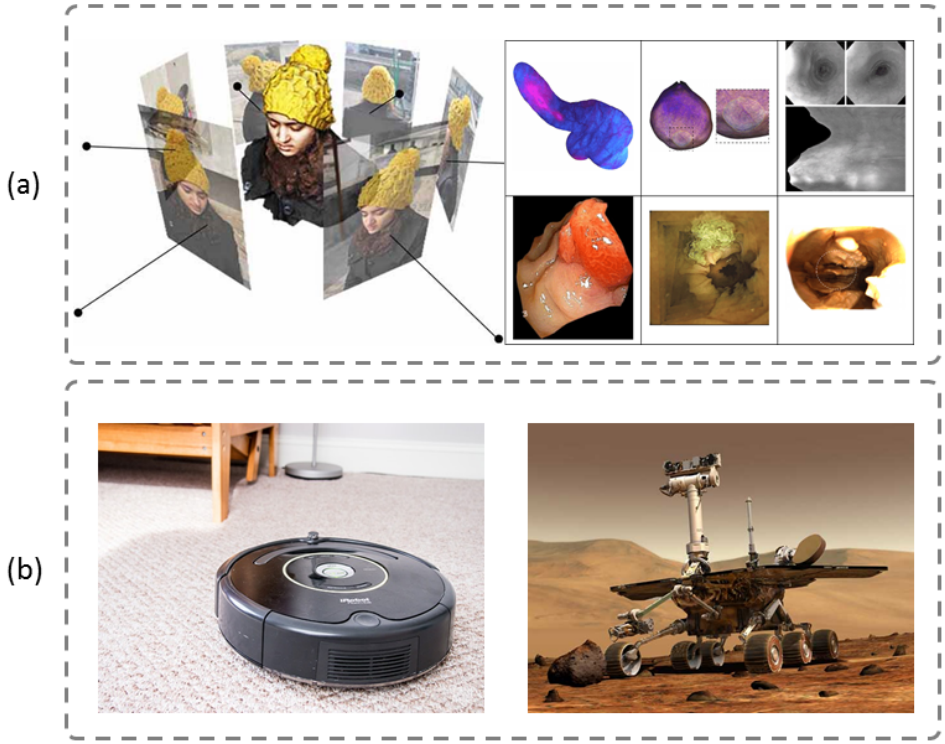


Figura 1 – Odometria Visual pode ser utilizada para reconstrução 3D (a) e para mensurar deslocamento (b). Aplicações convencionais, como a reconstrução de uma pessoa ou o deslocamento de um robô aspirador de pó, podem ser notadas pelas imagens à esquerda. Enquanto imagens à direita representam as aplicações inovadoras de reconstrução de órgãos através de imagens de um exame de endoscopia, e navegação de um robô espacial (*Mars Exploration Rover*).

dois conjuntos de pontos, p e q, no sentido de mínimos quadrados médios, conforme a Equação 1.1:

$$(R*,t*) = \min_{R,t} \sum_{i=1}^{n} |(R \times p_i + t) - q_i|^2 . \tag{1.1}$$

1.2.2 SLAM

Historicamente, SLAM é estudado em dois principais campos de pesquisa na área de visão computacional, divisão que consequentemente resultou em duas principais abordagens: SLAM probabilístico e SLAM geométrico. Alguns autores, (CHOSET et al., 2003) e (AHN et al., 2008), também nomeiam estas abordagens de SLAM por filtragem e SLAM por suavização, respectivamente. Em ambos os casos o objetivo principal é o mesmo, porém, as informações necessárias para executar cada método difere. Abordagens probabilísticas estão preocupadas com a solução do problema SLAM em que apenas o estado atual do robô e o mapa são estimados, incorporando medições do sensor assim que disponíveis. Enquanto abordagens geométricas

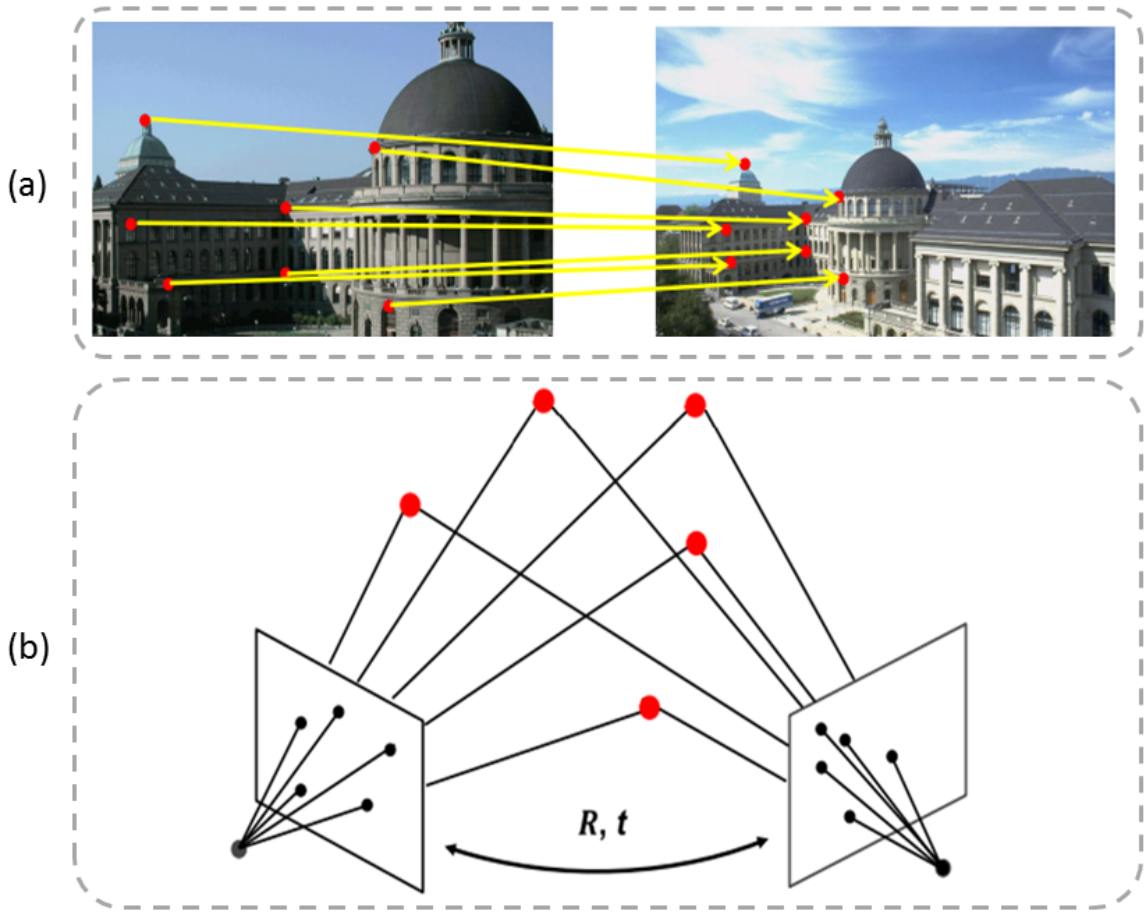


Figura 2 – O problema de Odometria Visual consiste em, uma vez encontrados pontos de referência em duas capturas de uma câmera em movimento (a), estimar uma rotação (R*) e uma translação (t*) que minimize a diferença entre os mesmos (b).

resolvem o problema SLAM como um todo, onde o posterior (mapa e pose) é estimado ao longo de todo o percurso e o caminho percorrido pelo agente é estimado junto com o mapa denso do ambiente. Segundo Hoppe (HOPPE et al., 2010), este problema é geralmente é resolvido usando uma técnicas de minimização de erro por mínimos quadrados. O autor também afirma que, devido ao avanço no desempenho computacional, tornou-se viável utilizar uma abordagem geométrica, que requer maior processamento, em uma variedade maior de aplicações.

Abordagens probabilísticas

Defina o estado do processo SLAM no instante t como sendo formado pelo mapa construído até então m_t e a posição atual do agente s_t. De uma forma geral, pode-se formular o problema SLAM com abordagem probabilística conforme expressa a equação 1.2, onde, dado um conjunto de medições z_t e a previsão de movimentação executada pelo agente u_t (odometria no tempo t), deve-se calcular a probabilidade p de que o agente assuma uma posição s_t e um

mapa m_t a um determinado tempo t (HOPPE et al., 2010):

$$p(s_t, m_t | z_t, u_t). \tag{1.2}$$

O objetivo nesta abordagem é encontrar o mapa e a pose de maior probabilidade entre o conjunto de todas as poses e mapas adquiridos até o momento. Este problema pode ser representado por uma rede Bayesiana dinâmica (HOPPE et al., 2010), conforme a estrutura ilustrada na Figura 3.

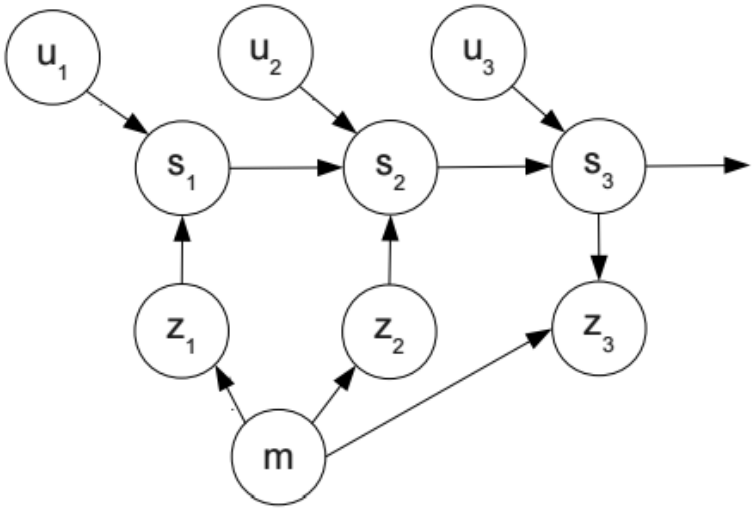


Figura 3 – Rede Bayesiana para formulação do problema SLAM com abordagem probabilística, onde deve-se estimar a provável pose *s* de um robô em movimento. Fonte: (HOPPE et al., 2010)

Este é um problema de estimação de estado bayeseano, para o qual existem varias tecnicas de solução tais como filtro de Kalman (KF), filtro de Kalman extendido (EKF) e filtros de partículas (PF). O problema é geralmente dividido em duas etapas: previsão e atualização. Maiores detalhes sobre esta abordagem podem ser encontrados nas referências (HOPPE et al., 2010) e (AHN et al., 2008). Este trabalho não adota abordagem probabilística.

Abordagens geométricas

A segunda grande classe de algoritmos SLAM é inspirada em métodos de otimização formalmente conhecidos como *Structure from Motion* (SfM). Para estas abordagens, a ideia principal é reformular o SLAM tradicional, o encarando como um problema de otimização que possa ser resolvido por métodos de mínimos quadráticos, também chamado de *bundle adjustment* (ajuste por blocos) por alguns autores (CHOSET et al., 2003) e (AHN et al., 2008). O grande desafio dessas abordagens é otimizar a cena 3D construída até o momento, assim como

a trajetória do sensor em movimento, ao minimizar o erro de reprojeção. Devido ao surgimento de novos sensores e ao incremento do poder de processamento, abordagens geométricas tem se tornado cada vez mais relevantes.

O *bundle adjustment* reduz-se a minimizar o erro de reprojeção entre pontos observados e previstos adquiridos através de *frames* consecutivos de uma câmera em movimento, que é expressa como a soma de quadrados de um grande número de funções não-lineares (quadráticas) de valor real. Assim, a minimização é conseguida utilizando algoritmos não-lineares de mínimos quadrados (LOURAKIS; ARGYROS, 2009). Destes, Levenberg-Marquardt tem provado ser um dos mais bem sucedidos devido à sua facilidade de implementação e sua capacidade convergir rapidamente a partir de uma ampla gama de suposições iniciais. Isso pode ser explorado para obter benefícios computacionais.

Formalmente, assuma que n pontos 3D foram vistos em m quadros. Seja x_{ij} a projeção do ponto i na imagem j e v_{ij} a variável binária que assume 1 se o ponto i for visível na imagem j e 0 caso contrário. Sejam também cada captura j parametrizada por um vetor a_j e cada ponto 3D i por um vetor b_i. *Bundle adjustment* minimiza o erro de reprojeção total levando em conta todos os pontos 3D e parâmetros da câmera. Utilizando essa notação o *bundle adjustment* pode ser expresso pela Equação 1.3, onde $Q(a_j, b_i)$ é a projeção prevista do ponto i na imagem j e $d(x,y)$ denota a distância Euclidiana entre os pontos relativos em cada imagem, representados pelos vetores x e y,

$$min \sum_{i=1}^{n} \sum_{j=1}^{m} v_{ij}\, d(Q(a_j, b_i), x_{ij})^2. \tag{1.3}$$

Bundle adjustment é, por definição, tolerante às projeções de imagem ausentes e minimiza um critério fisicamente significativo. (LOURAKIS; ARGYROS, 2009).

1.3 BREVE REVISÃO BIBLIOGRÁFICA

Esta revisão bibliográfica cobre apenas os trabalhos mais diretamente relacionados com essa dissertação. Outros trabalhos são citados ao longo do texto. Ao final são dadas indicações de trabalhos de revisão bibliográfica mais completos.

1.3.1 Odometria Visual vs. SLAM Visual

Técnicas SLAM que utilizam somente recursos visuais, como a utilizada neste trabalho, são também conhecidas como *Visual* SLAM ou V-SLAM. Por conta da quantidade de informações que podem ser obtidas a partir de sensores de vídeo, no entanto, seu custo computacional é superior pois exige algoritmos mais sofisticados para processar as imagens e extrair as informações necessárias. Há uma variedade de soluções utilizando diferentes sensores visuais, entre elas os sistemas monocular (DAVISON, 2003), ultrassom (MAHON et al., 2008), omnidirecional (KIM; OH, 2008), tempo de voo (TOF) (HOCHDORFER; SCHLEGEL, 2010) e, finalmente, câmeras RGB-D, como a utilizada neste trabalho, que combinam cor (RGB) com profundidade (D) (HENRY et al., 2012).

Fraundorfer (FRAUNDORFER; SCARAMUZZA, 2011) define odometria visual (VO) como o processo de estimativa da pose de um agente utilizando apenas a entrada de uma ou mais câmaras a ele associadas, enquanto SLAM como o processo em que, além de localizar-se em um ambiente desconhecido é necessário construir um mapa do mesmo, sem qualquer informação anterior. A principal diferença entre VO e V-SLAM está na consistência dos dados, VO foca na otimização local (entre capturas), já V-SLAM visa otimizar a estimativa global, quando, ao percorrer uma área previamente visitada, utiliza o conjunto de poses para reduzir o acumulo de erro. VO pode ser usada como um bloco de construção para um algoritmo de V-SLAM, no entanto, deve-se também adicionar alguma maneira para detectar fechamento de *loop* e, eventualmente, uma etapa de optimização global para obter um mapa consistente.

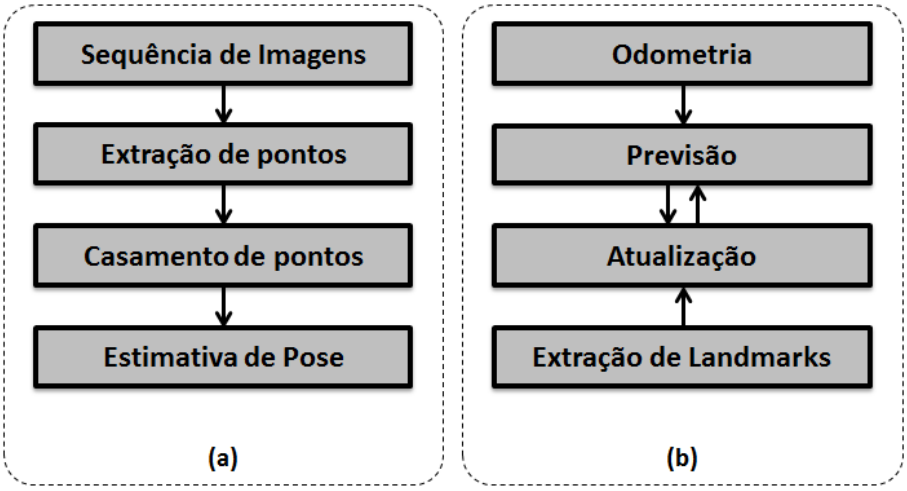


Figura 4 – Diagrama de blocos com os principais componentes de sistemas VO (a) e SLAM (b).

A Figura 4 ilustra uma visão generalista dos sistemas VO (a) e SLAM (b). Pode-se notar que o fluxo para VO (a) não possui realimentação, ou seja, as informações e estimativas anteriores são irrelevantes para a computação da estimativa atual. Enquanto SLAM (b) prevê o deslocamento através da odometria e o atualiza por extração, e consequentemente localização, de pontos relevantes feita utilizando dados do sensoriamento disponível. Um algoritmo de SLAM visual (V-SLAM) é obtido quando as etapas de Odometria e Sensoriamento utilizam apenas informações visuais, sendo assim, pode-se concluir que VO representa um bloco importante para a construção de V-SLAM.

1.3.2 Odometria Visual

Como explica Yousif (YOUSIF et al., 2015), a forma mais simples de localização é baseada em dados de sensoriamento binário, como codificadores (*enconders*) para medir a quantidade de rotação de rodas, ou detectores de obstáculos que podem utilizar ultrassom, luz infravermelha ou até contato seco. Estes métodos de odometria possuem diversas limitações, dentre elas a restrição à veículos terrestres de roda. Uma vez que a localização a partir da odometria é incremental, ou seja, sempre baseada na estimativa anterior, erros de medição são acumulados ao longo do tempo.

Para superar essas limitações, foram propostas outras estratégias de localização usando GPS, odometria LASER e, mais recentemente Odometria Visual, ou *Visual Odometry* (VO) (FRAUNDORFER; SCARAMUZZA, 2011), e *Simultaneos Localization and Mapping* (SLAM) (DURRANT-WHYTE; BAILEY, 2006). Apesar de VO não resolver o problema de acumulo de erro a cada medição, os pesquisadores mostraram que os métodos de odometria visuais possuem desempenho significativamente melhor do que as odometrias binárias anteriormente mencionadas. Além do custo de câmeras convencionais ser muito menor quando comparadas a sensores a laser de precisão, o que estimula ainda mais contribuições para campo de VO.

Existem abordagens de Odometria Visual baseadas somente em imagens providas por uma câmera RGB e outros capazes de realizar a tarefa através da minimização da diferença entre duas nuvens de pontos, requerendo somente de imagens profundidade de uma câmera *Time of Flight* (TOF), ou Tempo de Voo, que pode medir as profundidades de um ambiente utilizando luz ou som. Um exemplo destas abordagens são, respectivamente, a técnica de estrutura por movimento, ou *Structure from Motion* (SfM), executada por Benseddik (BENSEDDIK et al., 2014) e a técnica *Interative Closest Point* (ICP) utilizada nos trabalhos de Chetverikov

(CHETVERIKOV et al., 2002) e Pereira (PEREIRA et al., 2015).

Recentemente foram lançados diversos sensores híbridos categorizados como câmeras RGB-D. Com baixo custo e alto potencial, o sensor captura tanto imagens coloridas como mapas densos de profundidade em taxas de quadros de vídeo, o que possibilita combinar os pontos fortes de recursos visuais com as informações de profundidade para computar a movimentação feita pela câmera e também gerar densas representações em 3D do ambiente navegado. Para cada quadro capturado, uma câmera RGB-D, fornece uma imagem colorida (RGB) e uma imagem de profundidade (D). Isto possibilita trabalhar isoladamente com as técnicas citadas anteriormente e/ou mesclá-las, criando uma nova abordagem, como faz a contribuição de Milella (MILELLA; SIEGWART, 2006), que detecta pontos de interesse em e leituras RGB, os posiciona no espaço utilizando a captura de profundidade e, por fim, estima a movimentação por ICP.

A partir da observação de pontos com características fortes em uma sequencia de imagens, é possível estimar a transformação (rotação e translação) entre cada quadro através da triangulação dos mesmos. A terminologia VO estéreo é dada devido a utilização de duas câmeras de forma que a triangulação é calculada em um único par de captura, observando simultaneamente as características nas imagens esquerda e direita, que são separadas por uma distância fixa e conhecida, tornando possível estimar a transformação já no segundo par de leituras. Em VO monocular, estes mesmos pontos característicos devem ser observados em pelo menos três quadros diferentes para que a transformação possa ser estimada. Em cada captura são respectivamente executadas as seguintes etapas: observar pontos característicos no primeiro quadro, reobservá-los no segundo quadro, para só então calcular a transformação a partir do terceiro quadro (FRAUNDORFER; SCARAMUZZA, 2011).

A transformação entre os dois primeiros quadros consecutivos na visão monocular não é totalmente conhecida, o que torna impossível determinar a escala, e esta, por sua vez, é geralmente predefinida. Por outro lado, em visão estéreo, quando a distância para a cena é muito maior do que a distância entre as duas câmaras, se faz necessário a utilização de algorítimos de VO monocular, pois, neste caso, a visão estéreo torna-se ineficaz. (YOUSIF et al., 2015)

Extração e correspondência de características

Siegwart (SIEGWART et al., 2011) mostra que, na maioria dos casos, as imagens fornecidas por sensores visuais têm de ser processadas de modo a extrair informações úteis aos algoritmos. Tais informações podem ser expressas por pontos de interesse, como ilustra a Figura 5, que vão de características simples, como detecção de cantos e linhas, a recursos mais

elaborados, tais como bordas e conjuntos de pixels de mesma tonalidade, denominado manchas. Existem muitos detectores disponíveis na literatura sobre odometria visual, tais como:

- Detectores de cantos, como Moravec (MORAVEC, 1980), Harris (HARRIS; PIKE, 1988), Shi-Tomasi (SHI; TOMASI, 1994), e FAST (ROSTEN; DRUMMOND, 2006).
- Detectores de manchas, como SIFT (LOWE, 2004), SURF (BAY et al., 2006), e CENSURE (AGRAWAL et al., 2008).

O autor também aponta que cada detector tem suas próprias vantagens e desvantagens. Detectores de canto são mais rápidos de processar, mas é mais difícil distinguir os pontos, exatamente o oposto dos detectores de manchas, onde a distinção entre os pontos é mais precisa, porém consomem mais tempo de processamento. Mesmo que a demora na localização destes pontos não seja um problema, detecção por manchas não é sempre a escolha certa, por exemplo, SIFT ignora automaticamente cantos, o que pode não ser uma boa tática ao tratar imagens de ambientes urbanos. Por estas razões, a escolha do detector de recursos adequado deve ser cuidadosamente pensada para cada aplicação, levando em consideração as limitações computacionais, se o processamento deve ser em tempo real, o tipo de ambiente navegado e também o movimento realizado pelo sensor.

A Figura 5 ilustra a diferença entre detectores de cantos e detectores de manchas, representados por FAST (a) e SURF (b) respectivamente. Nela é possível notar que FAST (a) identifica um grande número de pontos através do encontro de linhas, enquanto SURF (b) localiza manchas por conjuntos de pontos da mesma cor. A circunferência em torno de cada ponto em (b) simboliza o tamanho da mancha pela quantidade de *pixels* a ela atribuída, sendo o cabelo do homem fotografado a mancha mais significativa.

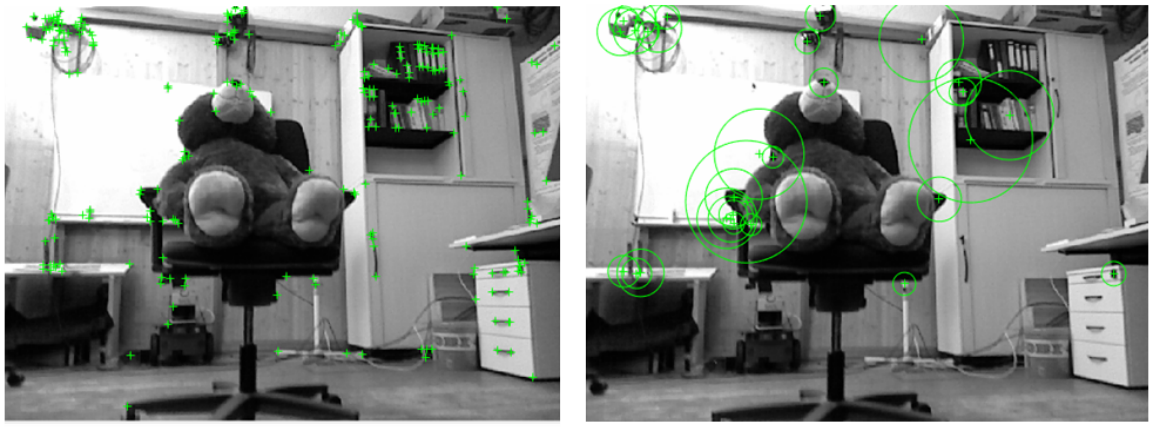


Figura 5 – Diferença entre os detectores de cantos FAST (a) e de manchas SURF (b).

Para modelos baseados em imagens, como VO, a estimativa de movimentação é computada pela transformação rígida (rotação e translação) que representa a variação de posição entre duas capturas consecutivas de uma câmera em movimento. Os dados utilizados neste calculo são justamente os pontos de interesse do primeiro quadro e suas correspondências no quadro seguinte.

Utilizando as denominações de Fraundorfer (FRAUNDORFER; SCARAMUZZA, 2011), o processo de casamento de características fortes entre pares de imagens é conhecido como *Feature Matching* (MUJA; LOWE, 2012), já o rastreamento dos mesmos pontos de interesse em múltiplos quadros subsequentes ocorre na tarefa de *Feature Tracking* (BAKER; MATTHEWS, 2004). *Feature Matching* é particularmente útil quando as alterações significativas na aparência das características ocorrem devido à observação das mesmas por sequências mais longas ou por movimentos bruscos. Enquanto *Feature Tracking* é útil quando ocorrem pequenas variações de movimentos entre quadros. (FRAUNDORFER; SCARAMUZZA, 2012)

Com o intuito de impedir falsas correspondências, se faz necessário filtrar aqueles pares de pontos cuja variação de localização nas imagens não seguiu um padrão. Normalmente utiliza-se a técnica de amostra aleatória consensual, no inglês *Random Sample Consensus* (RANSAC), proposta por Fischler (FISCHLER; BOLLES, 1981), que filtra estes pontos inconsistentes através da estimação de parâmetros por amostragem.

A Figura 6 ilustra a necessidade de utilizar o filtro RANSAC para remover falsas correspondências geradas ao localizar pontos de interesse nas imagens subsequentes, (a) em vermelho e (b) em verde, gerados através do detector de cantos FAST, e estimar por *Feature Matching* as correspondências entre estes pontos. A correspondencia entre os pontos de interesse é representada por uma linha amarela entre eles. Note que entre (a) e (b) existem correspondências computadas de forma errada. As correspondências remanescentes após o filtro são ilustradas entre (c) e (d).

1.3.3 SLAM

Yousif (YOUSIF et al., 2015) aponta que dentre os métodos categorizados como SLAM probabilísticos, o EKF-SLAM, onde EKF são siglas para *Extended Kalman Filter*,(SMITH et al., 1990) é, sem dúvida, a técnica mais utilizada para a estimação de estado. Ainda segundo o autor, ela é baseada em filtro de Bayes para a seleção e previsão de funções não lineares em que as suas aproximações lineares podem ser obtidas utilizando uma expansão em série de Taylor

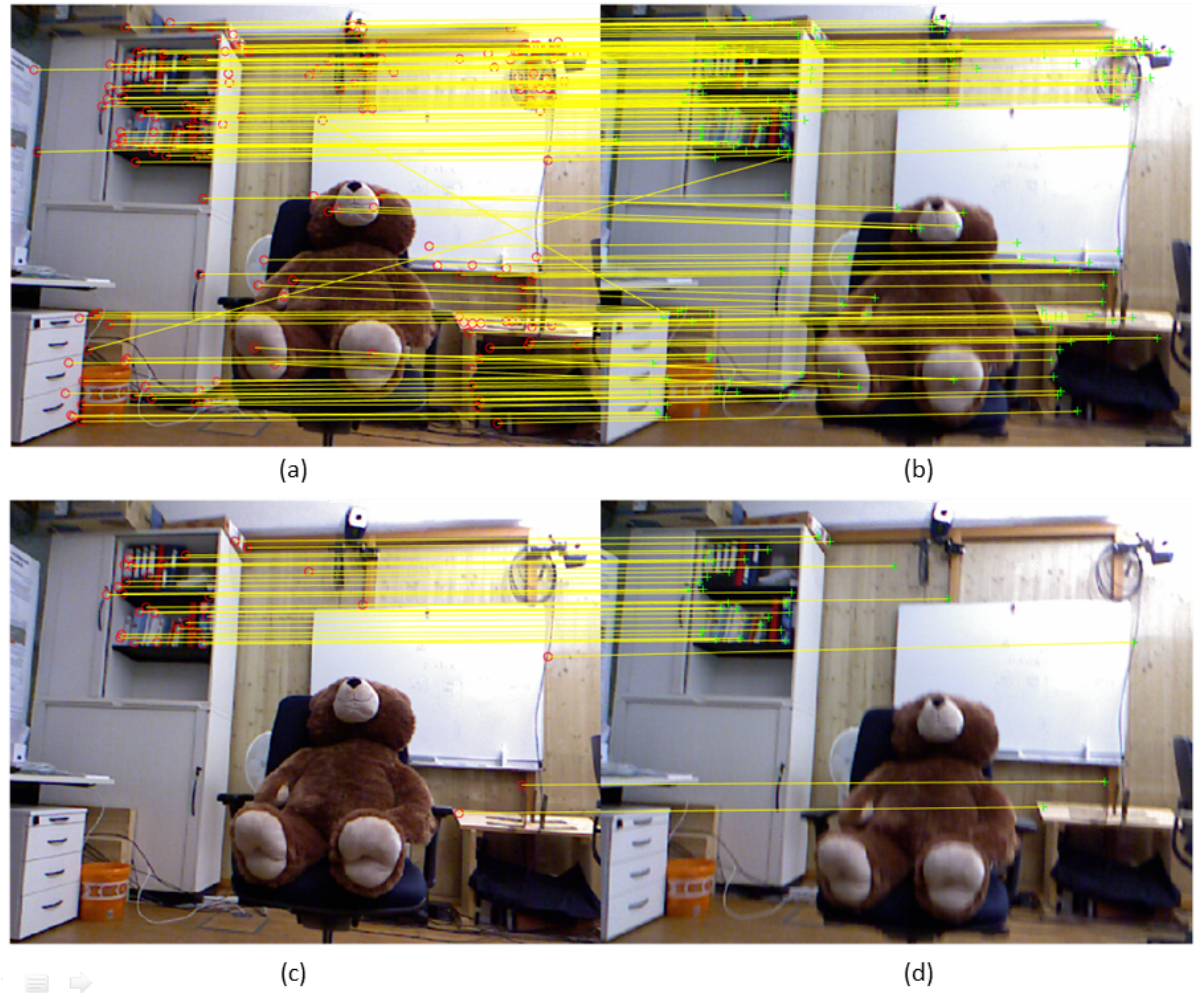


Figura 6 – Pontos com características fortes detectados em (a) e rastreados em (b) apresentam algumas inconsistências. A aplicação de RANSAC filtra as discrepâncias, resultando nas correspondências entre (c) e (d).

de primeira ordem. EKF basear-se na suposição de que toda leitura possui uma distribuição Gaussuana e, consequentemente, todos os estados estimados também teriam.

Abordagens que se utilizam de EKF tem a vantagem de ser computacionalmente mais eficientes do que outros filtros, como por exemplo o filtro de partículas, que na temática deste trabalho resulta em outra abordagem probabilística: *Particle Filter* SLAM (DOUCET et al., 2000). No entanto, a principal limitação de EKF-SLAM é justamente a sua aproximação linear de funções não lineares, utilizando a expansão de Taylor, que pode resultar em estimativas imprecisas em alguns casos. Outra limitação é causada ao estipular densidades Gaussianas, o que significa que o EKF-SLAM não pode lidar com as densidades multi-modais associados com localização global (localizar o robô sem o conhecimento da sua posição inicial).

Um exemplo de uma técnica SLAM baseada em filtro de partículas é a FastSLAM (MONTEMERLO et al., 2002). Esta consiste em três etapas principais: amostragem, atualização

e re-amostragem. Este presume que os recursos (*features*) extraídos do sensoriamento são condicionalmente independentes dado a representação do robô. Em outras palavras, em contraste com o EKF-SLAM que utiliza um único filtro EKF para estimar a pose do robô e representar o mapa, FastSLAM utiliza vários EKFs separados para cada *feature*. Isso significa que o número total de EKFs é $M \times N$ onde N é o número total de características observadas. (HOPPE et al., 2010)

Monte Carlo Localization (MCL) (DELLAERT et al., 1999) é um outro método de filtragem bastante conhecido para localização de um robô utilizando filtro de partículas (semelhante a FastSLAM). Este assume que o mapa do ambiente é fornecido e estima somente a pose de um robô em movimento.

Quando os modelos de movimento e de observação/movimento tem um grau elevado de não-linearidade, o filtro de Kalman estendido e suas derivações podem resultar num fraco desempenho. Isto é porque a covariância é propagada através de linearização do modelo não-linear subjacente.

1.4 METODOLOGIA

A localização de um robô depende fortemente do mapa construído, porém, para muitas aplicações o mapa do ambiente navegado não é fornecido, e mesmo para aquelas poucas outras onde o agente possui um mapa, este está sujeito a constantes alterações. Informações desatualizadas ou imprecisas podem resultar em um comportamento inesperado do robô autônomo, uma vez que as decisões de navegação dele são tomadas com base nestas informações. O principal desavio de qualquer algoritmo SLAM é gerar um mapa que, apesar das informações ruidosas dos sensores, seja o mais preciso possível.

Todo algoritmo SLAM possui duas principais etapas: a construção de um mapa e a localização do agente no mesmo. Com base na odometria ruidosa, a etapa de Mapeamento fornece, além do próprio mapa, uma estimativa de localização. Já a etapa de Localização utiliza-se de algoritmos para verificar a consistência global do mapa e o atualizar, consequentemente ajustando a trajetória percorrida pelo agente até o momento. Por estes motivos alguns autores, como (AHN et al., 2008) e (HOPPE et al., 2010), as chamam de Previsão e Atualização respectivamente.

A Figura 7 ilustra o fluxo simplificado do algoritmo de SLAM geométrico desenvolvido neste trabalho. Como descrito na seção anterior, Odometria Visual (VO) é utilizada como

um bloco de construção fundamental para um algoritmo de SLAM Visual, no entanto, para obter um mapa consistente, deve-se também adicionar alguma maneira se realizar uma optimização global, corrigindo o erro acumulativo da Odometria.

Dadas duas leituras consecutivas de uma câmera RGB-D em movimento, $I_{RGBD(i)}$ e $I_{RGBD(i-1)}$, o objetivo do bloco de Odometria é encontrar uma variação de pose, ou seja, uma rotação (R) e uma translação (t), que represente o deslocamento da câmera entre os quadros. Despondo de R e t, uma nova pose no quadro i pode ser calculada a partir da pose no quadro i-1. A partir das mesmas leituras, o bloco Previsão constrói nuvens de pontos, fixando a primeira na origem e deslocando as subsequentes conforme posições e orientações fornecidas pela Odometria, criando assim um Mapa 3D. Por fim, o bloco de Atualização fragmenta este mapa e o reconstrói com base em um algoritmo de minimização de erro por mínimos quadrados, retornando para o bloco de previsão um novo mapa 3D. O resultado de todo este processo é, além de um mapa globalmente consistente, a correção das poses calculadas inicialmente pela Odometria. A concatenação de todas as poses forma a trajetória percorrida pela câmera RGB-D em movimento. Esta será utilizada posteriormente como forma de mensurar a acurácia do processo SLAM.

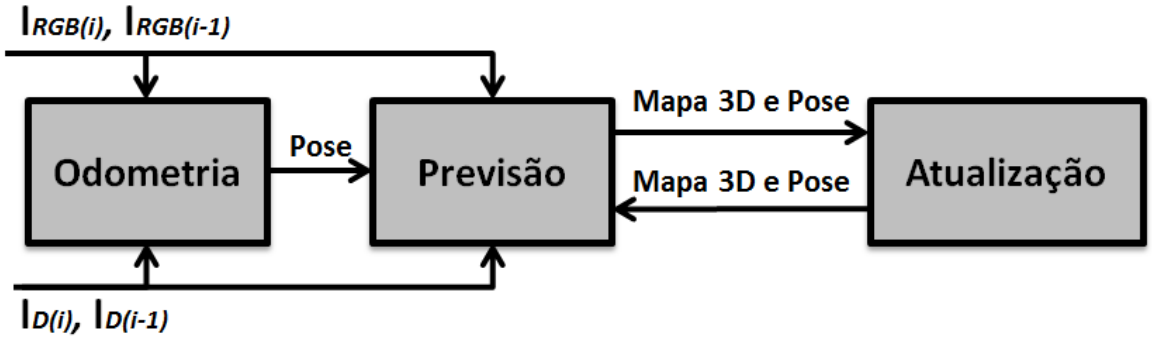


Figura 7 – Fluxograma simplificado do algoritmo de SLAM geométrico desenvolvido neste trabalho.

1.5 EXPERIMENTOS, DADOS E AVALIAÇÃO

Para avaliar qualitativamente o algoritmo de SLAM proposto neste trabalho, foram utilizados como *benchmark* os conjuntos de imagens RGB-D disponibilizadas ao público por Sturm (STURM et al., 2012), as quais foram adquiridas a partir de uma câmera Microsoft Kinect. Juntamente com as imagens, Sturm também disponibiliza a trajetória real feita pela câmera (*groundtruth*), esta adquirida por um sistema externo de captura de movimento de alta

precisão. Há uma grande variedade de sequencias de capturas disponíveis para *download*, porém, a avaliação de desempenho foi feita utilizando as sequências denominadas **freiburg1/floor** e **freiburg1/teddy**, pois estas contêm tanto movimento de translação quanto de rotação em um ambiente típico de escritório em velocidades e percursos diferentes. A câmera RGB-D foi movimentada manualmente em ambas as sequências com o diferencial de que em **freiburg1/floor** a mesma não perde a referência do chão na maioria dos seus quadros.

Os ensaios feitos consistem em percorrer todos os *frames* estimando a variação de pose da câmera entre eles e por consequência sua trajetória. O percurso da câmera RGB-D estimado utilizando as técnicas apresentadas pode ser comparado com a trajetória real (*grountruth*) e o erro de trajetória pode então ser calculado. Neste trabalho será utilizado o RMSE (*Root Mean Square Error*) como índice de avaliação dado por $RMSE(1:n) = \sqrt{(\frac{1}{n}\sum_{i=1}^{n} \left\| t_i^{GT} - t_i^{SLAM} \right\|^2)}$, onde n é o numero de quadros da sequência, t_i^{GT} os valores das três coordenadas cartesianas fornecidas (*Groundtruth*) no instante i e t_i^{SLAM} são os mesmos três valores estimados pelo algoritmo SLAM implementado. Note que, apesar da variação de pose ser representada por uma matriz de rotação e um vetor de translação, apenas a translação é considerada ao se calcular o RMSE. Isso se dá pois, ao longo de uma trajetória, quaisquer alterações na rotação, mesmo que pequenas, refletem diretamente nos valores de translação.

2 FUNDAMENTAÇÃO TEÓRICA E ALGORITMOS

Este capítulo apresenta a fundamentação teórica da estimação de pose de uma câmera em movimento e os algoritmos e pseudocódigos da Odometria Visual e SLAM RGB-D cujas avaliações das implementações são descritas no Capítulo 3.

2.1 REPRESENTAÇÃO 3D DE UMA CENA

Ma (MA et al., 2003) explica que o estudo da relação geométrica entre uma cena tridimensional e as imagens de uma câmera em movimento que a geraram se resume a interação entre duas transformações fundamentais: o movimento de corpo rígido que modela como a câmera se move, e a (re)projeção em perspectiva que descreve o processo de formação da imagem. As teorias que embasam cada uma destas duas transformações foram desenvolvidas independentemente muito antes da utilização conjunta das mesmas.

2.1.1 Espaço tridimensional Euclidiano

O espaço tridimensional Euclidiano, denotado por E^3, pode ser representado por um *frame* de coordenadas cartesianas (global): cada ponto p $\in E^3$ pode ser identificado com um ponto no $\mathbb{R}^3$ por três coordenadas: $X \doteq [X_1, X_2, X_3]^T$, ou, como preferem alguns autores, simplesmente representado por $[X, Y, Z]^T$. Através de tal atribuição de estrutura cartesiana, se estabelece uma correspondência de igual para igual entre E^3 e $\mathbb{R}^3$, o que permite falar com segurança sobre pontos e suas coordenadas como se fossem a mesma coisa (MA et al., 2003).

Coordenadas cartesianas são o primeiro passo no sentido de ser capaz de medir distâncias e ângulos. Com isso em vista, E^3 deve ser dotado de uma métrica, cuja definição é baseada na noção de vetor. No espaço euclidiano, um vetor é identificado por um par de pontos p e $q \in E^3$; isto é, um vetor v é definido como uma seta dirigida que liga p a q. O ponto p é normalmente chamado de ponto de base de v. Em coordenadas, o vetor v é representado pelo trio $[v_1, v_2, v_3]^T \in \mathbb{R}^3$, onde cada coordenada é a diferença entre as correspondentes coordenadas dos dois pontos que formam o vetor.

Um vetor livre é um vetor cuja definição não depende do seu ponto base. Por exemplo, um mesmo vetor livre pode ser representado por dois pares de pontos (p, q) e (p', q') com coordenadas que satisfaçam Y-X = Y'-X'. Intuitivamente, isto permite que um vetor v possa ser transportado a qualquer lugar em E^3. Sem perda de generalidade, pode-se supor que o ponto

de referência é a origem de uma nova moldura cartesiana, de modo que, para tal, X = 0 e v = Y. Esta notação pode ser um pouco confusa, uma vez que o espaço tridimensional euclidiano é representado por uma moldura cartesiana (global) e um vetor livre pode ser fixado à origem de uma nova moldura (local), e estas podem não ser a mesma.

Assim, o conjunto de todos os vetores livres em um espaço bidimensional, definido por dois vetores u e $v \in \mathbb{R}^3$, é dado por:

$$\alpha v + \beta u = (\alpha v_1 + \beta u_1, \alpha v_2 + \beta u_2, \alpha v_3 + \beta u_3)^T \in \mathbb{R}^3.$$

A métrica Euclidiana clássica para a E^3 é a simplesmente definida por um produto interno em seu espaço vetorial. A quantidade $\|v\| = \sqrt{\langle v, v \rangle}$ é chamado de norma euclidiana do vetor v. Pode-se mostrar que, por uma escolha apropriada da estrutura cartesiana, qualquer produto interno em E^3 pode ser convertido para o seguinte forma:

$$\langle u, v \rangle = u^T v = u_1 v_1 + u_2 v_2 + u_3 v_3. \tag{2.1}$$

Quando o produto interno canônico, dado por $\langle u, v \rangle = u^T v$, entre dois vetores é igual zero, eles são então definidos como ortogonais.

Finalmente, um espaço euclidiano E^3 pode ser descrito formalmente como um espaço em que, no que diz respeito a uma estrutura cartesiana, pode ser identificado ao $\mathbb{R}^3$ e tem uma métrica (no seu espaço vetorial) dada pela raiz quadrada do produto interno. Com uma tal métrica, pode-se medir não apenas as distâncias entre os pontos ou ângulos entre vetores, mas também o cálculo do comprimento de uma curva, ou o volume de uma região (MA et al., 2003).

Enquanto o produto interno de dois vetores retorna um escalar, o produto cruzado (vetorial), definido pela equação 2.2, retorna um vetor,

$$u \times v = \begin{bmatrix} u_2 v_3 - u_3 v_2 \\ u_3 v_1 - u_1 v_3 \\ u_1 v_2 - u_2 v_1 \end{bmatrix} \in \mathbb{R}^3. \tag{2.2}$$

É imediato verificar que o produto cruzado de dois vetores é linear, e que $\langle u \times v, u \rangle = \langle u \times v, v \rangle = 0$.

Ao fixar u, o produto cruzado pode ser interpretado como um mapa $v \mapsto u \times v$ entre $\mathbb{R}^3$ e $\mathbb{R}^3$ que pode ser representado por uma matriz. Tal matriz é denotada por $\widehat{u} \in \mathbb{R}^{3\times 3}$ dada

por:

$$\widehat{u} = \begin{bmatrix} 0 & -u_3 & u_2 \\ u_3 & 0 & -u_1 \\ -u_2 & u_1 & 0 \end{bmatrix} \in \mathbb{R}^3. \tag{2.3}$$

O produto cruzado permite definir um mapa entre um vetor u, e uma matriz anti-simétrica 3×3, $\widehat{u}$. O inverso também é verdadeiro: cada matriz anti-simétrica 3×3 pode ser associada a um vetor tri-dimensional.

2.1.2 Movimento de Corpo Rígido

Considere um objeto em movimento na frente de uma câmera. Para descrever seu movimento se deve, em princípio, especificar a trajetória de cada ponto no objeto dando suas coordenadas como uma função do tempo $X(t)$. Felizmente, para obter a movimentação de um objeto rígido, é suficiente especificar o movimento de um único ponto e de três eixos de coordenadas associadas a esse ponto.

A condição que define um objeto rígido é que a distância entre quaisquer dois pontos pertencentes ao mesmo não altera ao longo do tempo, ou enquanto o objeto se move. Em outras palavras, se o vetor v é definido pelos pontos p e q, o comprimento de v permanece o mesmo: $\|v(t))\| = constante$. Um movimento do corpo rígido pode ser representado por uma família de transformações, *g(t)*, que descrevem como as coordenadas de cada ponto no objeto muda em função do tempo (MA et al., 2003)

$$, g(t) : \mathbb{R}^3 \to \mathbb{R}^3$$
$$X \mapsto g(t)(X).$$

No entanto, preservar distâncias entre os pontos de um corpo não é o único requisito que uma movimentação rígida deve satisfazer, pois existem transformações que preservam as distâncias e não são fisicamente realizáveis, como por exemplo o mapeamento descrito abaixo, que preserva a distância mas não a orientação:

$$f : [X_1, X_2, X_3]^T \mapsto [X_1, X_2, -X_3]^T.$$

Para representar corretamente um movimento de corpo rígido se faz necessário então ser capaz de conservar não só a distância entre pontos, mas também sua orientação. Em outras

palavras, deve-se preservar o produto interno ($\|g_*(v)\| = \|v\|$) e o produto cruzado (vetorial) ($g_*(u) \times g_*(v) = g_*(u \times v)$).

O fato de que as distâncias e as orientações são preservadas num movimento rígido significa que os pontos individuais não podem se distanciar um do outro, no entanto, eles podem colectivamente girar em relação uns aos outros. A figura 8 mostra um objeto, neste caso, uma câmara (C), em movimento em relação a um plano de coordenadas fixas (W). Quando a câmera se move, o plano cartesiano relativos à câmera também se move, como se fosse fixo a mesma. A configuração da câmara pode ser então determinada pelo vetor T que liga a origem do plano global (W) e a origem do plano relativo à camera (C), também denominado translação, e a orientação (R) de C relativa ao plano global W.

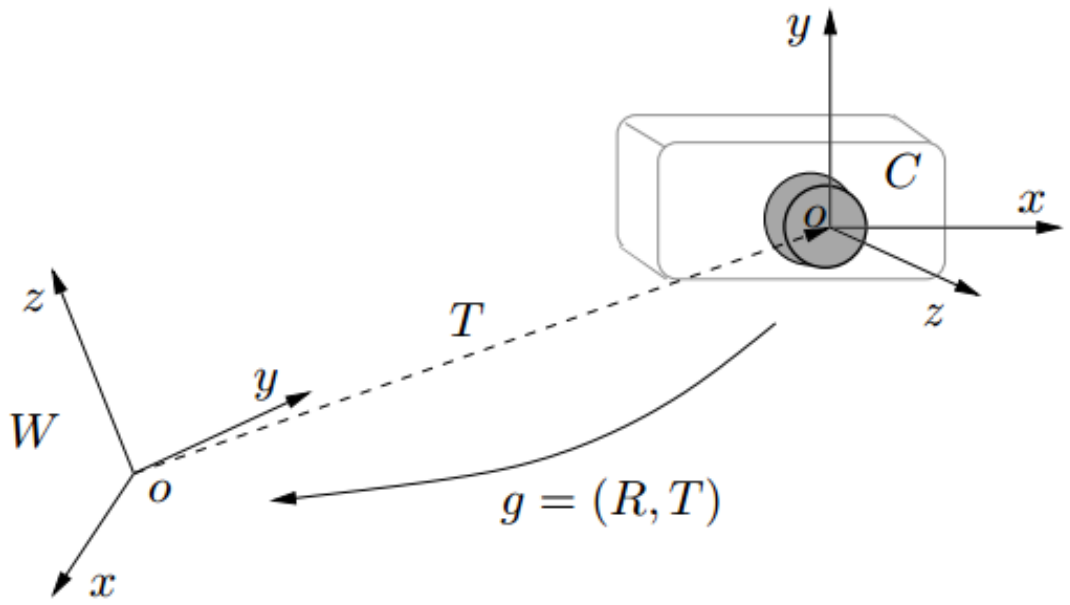


Figura 8 – Um movimento de corpo rígido entre uma câmera (C) e um plano cartesiano global (W).

Neste problema de visão computacional, não há escolha óbvia de plano global e plano de referência. Pode-se também trabalhar com o plano global fixo à câmera, tornando o desafio especificar a translação e rotação da cena (assumindo que esta é rígida). Tudo o que importa é o movimento relativo entre a cena e a câmera.

2.1.3 Movimento Rotacional e suas Representações

Suponha que temos um objeto rígido que gira em torno de um ponto fixo $o \in E^3$. Como é possível descrever a sua orientação relativa a um plano cartesiano global W? Sem perda

de generalidade, pode-se assumir sempre que a origem do plano tridimensional W é o centro da rotação. Se não for o caso, basta simplesmente movimentar a origem do plano cartesiano fixo ao corpo em movimento (C) até o ponto o (MA et al., 2003). A relação entre estes planos é ilustrado na figura 9.

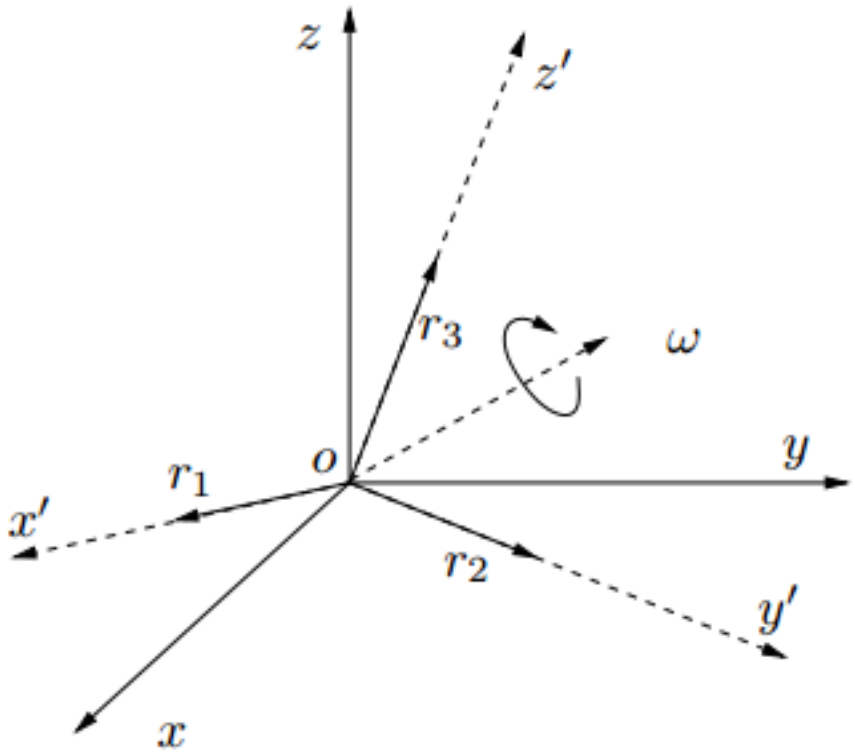


Figura 9 – Representação da rotação de um corpo rígido sobre um ponto fixo o. O plano de coordenadas globais (W) é fixo, enquanto o plano referente ao corpo em movimento (C) rotaciona.

A orientação do C em relação à estrutura W é determinada pelas coordenadas dos três vetores ortonormais $r_1 = g_*(e_1), r_2 = g_*(e_2), r_3 = g_*(e_3) \in \mathbb{R}^3$ em relação ao plano de coordenada global W. Estes três vetores correspondem aos eixos X', Y' e Z' do plano C. Os mesmos podem ser representados pela matriz 3×3 descrita abaixo:

$$R_{wc} = [r_1, r_2, r_3] \in \mathbb{R}^{3\times3}.$$

Uma vez que r_1, r_2 e r_3, formam um quadro ortonormal, é correto afirmar que o inverso da matriz R_{wc} é igual a sua transposta: R_{wc}^T. Além disso, como a configuração dos vetores segue a regra da mão direita, é possível afirmar que o determinante de R_{wc} deve ser 1 positivo (MA et al., 2003). O conjunto de matrizes que assumem tais características, também chamadas de matrizes de rotação, pode ser representado por:

$$SO(3) = \left\{R \in \mathbb{R}^{3\times3} | R^T R = I, det(R) = +1\right\}.$$

2.1.4 Projeção perspectiva e profundidade

O modelo de uma câmera é uma função que mapeia o mundo 3D real em um plano de imagem 2D. A perspectiva é a propriedade deste modelo que gera o efeito onde os objetos que estão longe pareçam menores do que objetos mais próximos. *Pinhole*, descrito pela equação 2.4, é o modelo mais comum de uma projeção em perspectiva, este assume que a imagem é formada por raios de luz provenientes dos objetos que passam através do centro da lente (YOUSIF et al., 2015):

$$\lambda \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = KX = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}. \tag{2.4}$$

Na equação 2.4, u e v são as coordenadas perspectivas na imagem 2D de um ponto 3D com coordenadas X, Y e Z, depois da sua projeção no plano da imagem. λ é o fator de profundidade, K é a matriz de calibração intrínseca que contém os parâmetros: f_x e f_y são os comprimentos focais na direções x e y, e c_x e c_y são coordenadas 2D do centro da projeção.

Os conjuntos contendo todos os pontos rastreados no quadro atual $P = \{p_1, p_2, ..., p_n\}$ e o anterior $Q = \{q_1, q_2, ..., q_n\}$ são utilizados para estimar a pose da câmera. Com base em 2.4, a reprojeção destes pontos em um espaço 3D será ser dada por,

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \lambda(u - c_x)/f_x \\ \lambda(v - c_y)/f_y \\ \lambda \end{bmatrix}.$$

2.1.5 Estimativa de Movimentação

Devido a facilidade de ilustração e compreensão, considere um ponto em um plano 2D, x = ($\boldsymbol{x}$,$\boldsymbol{y}$) $\in \mathbb{R}^2$. Segundo (FOLEY et al., 1982) todas as transformações possíveis de x podem ser escritas por 2.5, onde as variáveis de a a f são escalares,

$$x' = \begin{bmatrix} ax + by + c \\ dx + ey + f \end{bmatrix}. \tag{2.5}$$

Por exemplo, uma transformação de pura translação seria dada quando $a = 1$, $b = 0$, $d = 0$ e $e = 1$, reduzindo a equação para

$$x' = \begin{bmatrix} x + c \\ y + f \end{bmatrix}.$$

Já se a e e forem igual a $\cos\theta$, enquanto $b = -\text{sen}\theta$, $d = \text{sen}\theta$ e $c,f = 0$, computa-se uma transformação de pura rotação expressa por

$$x' = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}.$$

A figura 10 ilustra outras possíveis combinações além das citadas anteriormente. Entretanto, somente transformações rígidas, que não alterem formato e tamanho, são relevantes para a aplicação deste trabalho.

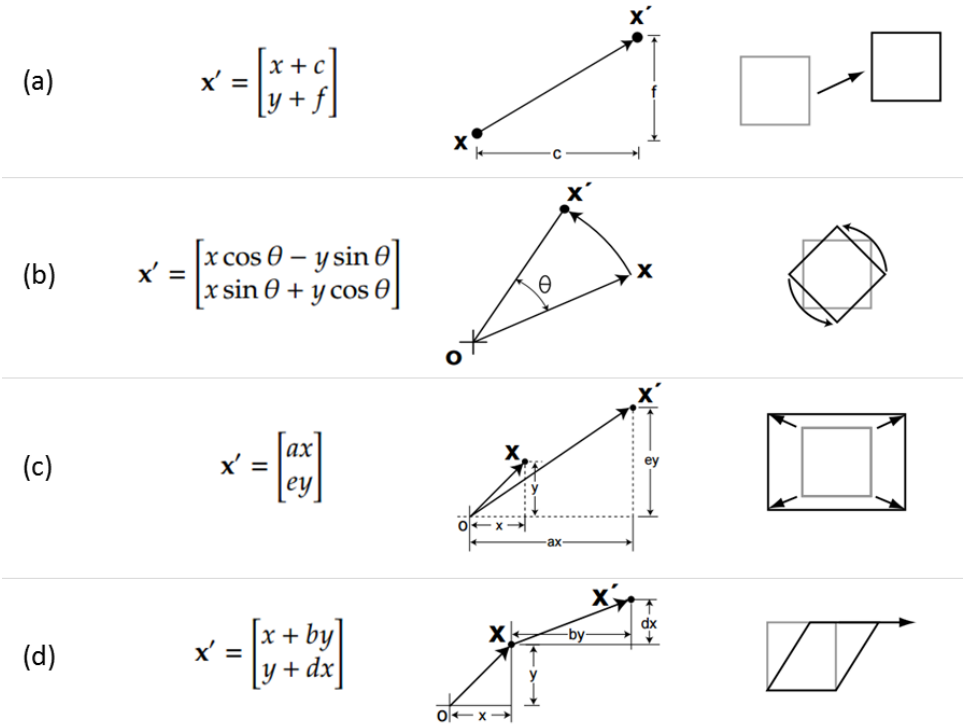


Figura 10 – Representação gráfica das possíveis transformações de um ponto x, seguido pela ilustração da mesma em uma forma geométrica: translação (a), rotação (b), escala (c) e distorção (d). Fonte: (FOLEY et al., 1982)

Conforme (FOLEY et al., 1982), transformações rígidas podem ser melhor representadas como uma multiplicação de matrizes de um ponto que represente um vetor x, onde a

transformação de x é dada por $x' = Mx$, M sendo a matriz, como em

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} ax+by \\ dx+ey \end{bmatrix} = \begin{bmatrix} a & b \\ d & e \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}.$$

No formato de matriz, pode-se considerar as transformações lineares 2D de rotação, escala e distorção expressas respectivamente nos itens (b), (c) e (d) da Figura 10, como

$$\text{Escala: } \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix}, \text{ Rotação: } \begin{bmatrix} cos\theta & -sen\theta \\ sen\theta & cos\theta \end{bmatrix}, \text{ Distorção: } \begin{bmatrix} 1 & h_x \\ h_y & 1 \end{bmatrix},$$

onde s_x e s_y são as escalas das coordenadas x ey de um ponto, θ é o angulo de rotação antihorário em torno da origem, em radiano, h_x é o fator de distorção horizontal e h_y o vertical.

Coordenadas homogêneas

Através de matrizes, transformações complexas podem ser expressadas de forma simples, porém, para ser capaz de representar todas as transformações possíveis deve-se converter o universo do problema de 2D em 3D (ou de 3D para 4D) utilizando coordenadas homogêneas (FOLEY et al., 1982). O primeiro passo é representar um ponto $p = (x,y)$ no formato de vetor e adicionar uma terceira coordenada de valor igual a 1: $\begin{bmatrix} x & y & 1 \end{bmatrix}^T$. Por convenção, esta coordenada extra é chamada de w para distinguir da coordenada 3D usual z. Também se faz necessário a inserção de w nas matrizes de transformação, como

$$\text{Escala: } \begin{bmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ Rotação: } \begin{bmatrix} cos\theta & -sen\theta & 0 \\ sen\theta & cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \text{ Distorção: } \begin{bmatrix} 1 & h_x & 0 \\ h_y & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

A multiplicação de matrizes e vetores, ambos 3D homogêneos, resulta na expressão de transformação mencionada anteriormente:

$$\begin{bmatrix} a & b & 0 \\ d & e & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by \\ dx+ey \\ 1 \end{bmatrix}.$$

Porém, a real vantagem da utilização de matrizes homogêneas pode ser vista com a inserção dos parâmetros de translação, c e f na terceira coluna da matriz, gerando o mesmo resultado expresso na equação

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by+c \\ dx+ey+f \\ 1 \end{bmatrix}.$$

Estimativa de pose

A etapa fundamental de um sistema de Odometria Visual é a estimativa da movimentação feita pela câmera entre cada captura. Esta movimentação pode ser representada pela variação de pose da câmera, ou seja, deve existir uma rotação (R) e uma translação (t), que represente o deslocamento da câmera entre os quadros. Como mostra Foley (FOLEY et al., 1982), utilizando matrizes homogêneas, R e t de pontos tridimensionais podem ser expressos por

$$\text{Translação: } \begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{ Rotação no eixo x: } \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta_x & -sen\theta_x & 0 \\ 0 & sen\theta_x & cos\theta_x & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\text{Rotação no eixo y: } \begin{bmatrix} cos\theta_y & 0 & sen\theta_y & 0 \\ 0 & 1 & 0 & 0 \\ -sen\theta_y & 0 & cos\theta_y & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \text{ Rotação no eixo z: } \begin{bmatrix} cos\theta_z & -sen\theta_z & 0 & 0 \\ sen\theta_z & cos\theta_z & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}.$$

Sejam então $P = \{p_1, p_2, ..., p_n\}$ e $Q = \{q_1, q_2, ..., q_n\}$ dois conjuntos de n pontos de coordenadas x, y e z. A rotação R e a translação t podem ser obtidas resolvendo o seguinte problema de mínimos quadrados (SORKINE, 2009):

$$(R*, t*) = \min_{R,t} \sum_{i=1}^{n} |(R \times p_i + t) - q_i|^2 .. \tag{2.6}$$

Sorkine (SORKINE, 2009) propõe cinco passos para o calculo de R e t, são eles:

- Computar o centroide dos conjuntos de pontos P e Q dado por

$$\bar{p} = \frac{\sum_{i=1}^{n} p_i}{n} \text{ e } \bar{q} = \frac{\sum_{i=1}^{n} q_i}{n}.$$

- Calcular os vetores centralizados

$$v1_i = p_i - \bar{p} \text{ e } v2_i = q_i - \bar{q},\ i = 1, 2, ...n.$$

- Encontrar a matriz de covariância S de dimensões d x d, onde d corresponde ao número de coordenadas cartesianas, 3 no caso deste trabalho, e M_{v1} e M_{v2} as matrizes $d \times n$ que possuem respectivamente $v1_i$ e $v2_i$ como colunas e I sendo a matriz identidade $d \times n$

$$S = M_{v1} \times I \times M_{v2}^T.$$

- Computar a decomposição em valores singulares (SVD), dada por $S = U\Sigma V^T$, e finalmente a rotação R utilizando

$$R* = V \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(VU^T) \end{bmatrix} U^T.$$

- Por último, o calculo da translação é dado por

$$t* = \bar{q} - R * \bar{p}.$$

2.2 ALGORITMOS

Esta descreve detalhadamente como foi implementado a lógica de programação ilustrada na Figura 7, que divide o problema de SLAM em três blocos: Odometria, Previsão e Atualização. Na próxima seção serão apresentados os pseudocódigos correspondentes explicando cada passo dos mesmos.

2.2.1 Odometria

Dado duas leituras consecutivas de uma câmera RGB-D em movimento, o objetivo da Odometria Visual (VO) é encontrar uma variação de pose, ou seja, uma rotação (R) e uma translação (t), que represente o deslocamento da câmera entre os quadros. A abordagem para VO proposta neste trabalho consiste das seguintes etapas de processamento: SURF (BAY et al., 2006), *Feature Matching* (MUJA; LOWE, 2012), *Feature Tracking* (BAKER; MATTHEWS, 2004), RANSAC (FISCHLER; BOLLES, 1981) e, finalmente, Estimativa de Pose por SVD proposta por Sorkine (SORKINE, 2009), como ilustra o fluxograma da Figura 11.

Primeiramente, a partir do par de imagens RGB, a técnica SURF identifica regiões com características fortes, extraindo pontos de interesse em ambas as capturas. Em seguida se faz necessário identificar as correspondências entre os pontos de interesse da leitura atual ($I_{RGB}(i)$) com a de referência ($I_{RGB}(i-1)$) utilizando *Feature Matching*. Filtrados aqueles pares pontos que não seguem um padrão de deslocamento utilizando RANSAC, os pontos de interesse remanescentes, até agora compostos somente por duas coordenadas (x, y), e suas devidas correspondências seguem para etapa de Estimativa de Pose. Este ultimo bloco requer a terceira coordenada (profundidade) dos conjuntos de pontos, que pode ser estimada por triangulação como no trabalho de Benseddik (BENSEDDIK et al., 2014). A abordagem proposta neste trabalho adquire a profundidade diretamente da leitura *Depth* (D), sendo desnecessário qualquer

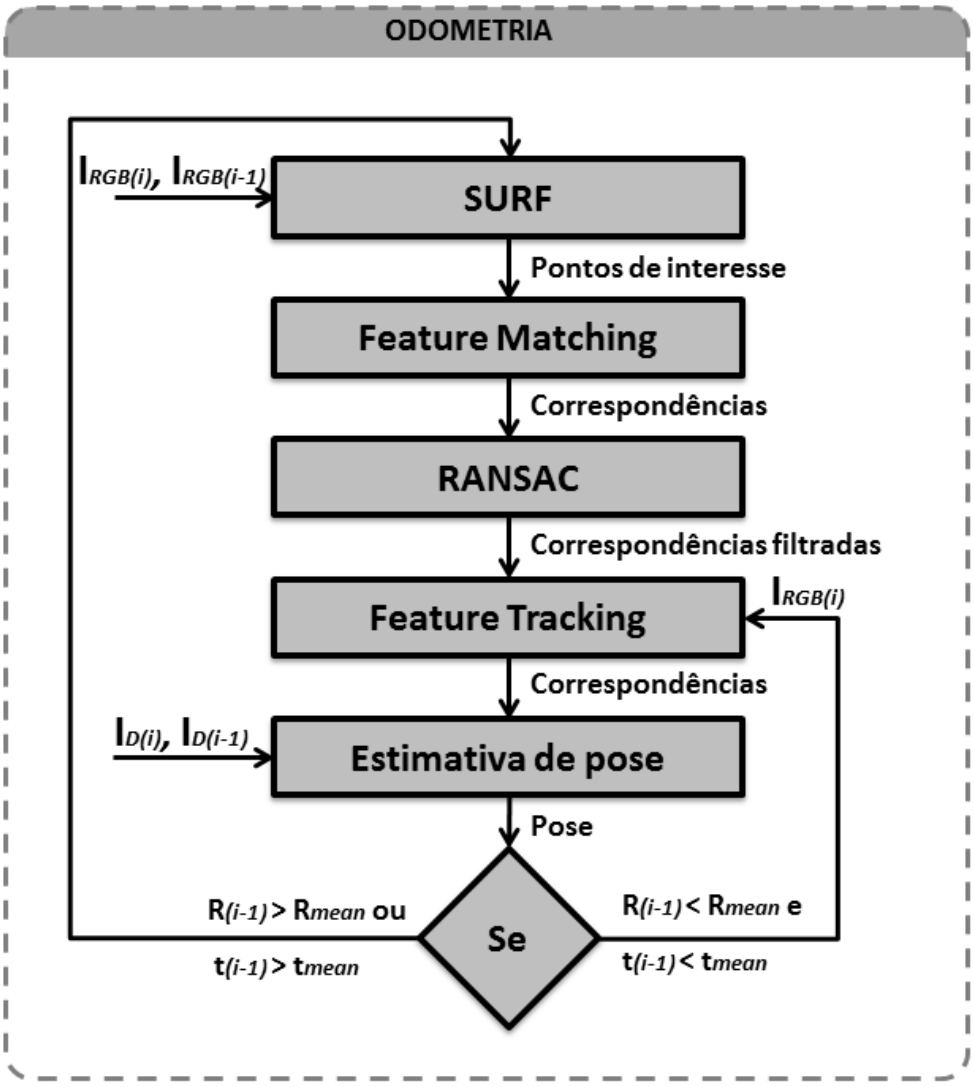


Figura 11 – Fluxograma de técnicas aplicadas para estimar a pose da câmera a partir de um par de imagens RGB, $I_{RGB}(i)$ e $I_{RGB}(i-1)$, e suas respectivas imagens de profundidade, $I_D(i)$ e $I_D(i-1)$.

processamento extra. Finalmente, a transformação rígida entre os quadros pode ser calculada utilizando SVD (*Single Value Decomposition*).

Esta sequência é utilizada para o primeiro *loop* de processamento, e *Feature Tracking* do segundo em diante. Quando o *loop* anterior resulta em uma rotação (R) e/ou uma translação (t) que fogem à média (R_{mean} e t_{mean}), pode-se detectar alterações significativas na aparência das características rastreadas, possivelmente devido à observação das mesmas por sequências mais longas ou por movimentos bruscos feitos pela câmera, em ambos os casos, se faz necessário redefinir os pontos a serem rastreados. É de notar que o algoritmo Feature Matchin é de fato uma implementação do algoritmo KLT (LUCAS et al., 1981).

Tendo a transformação rígida computada entre as poses da câmera nas posições atual

e anterior, *i* e *i-1*, obviamente, para estimar a pose global da câmera, deve-se concatenar todas as transformações anteriores.

2.2.2 Previsão

A etapa de Previsão tem como principal tarefa construir um mapa 3d a partir das leituras da câmera RGB-D e das informações de posição e orientação da câmera fornecidas pela etapa de Odometria. Para isto, divide-se esta etapa em três blocos, conforme ilustra a Figura 12.

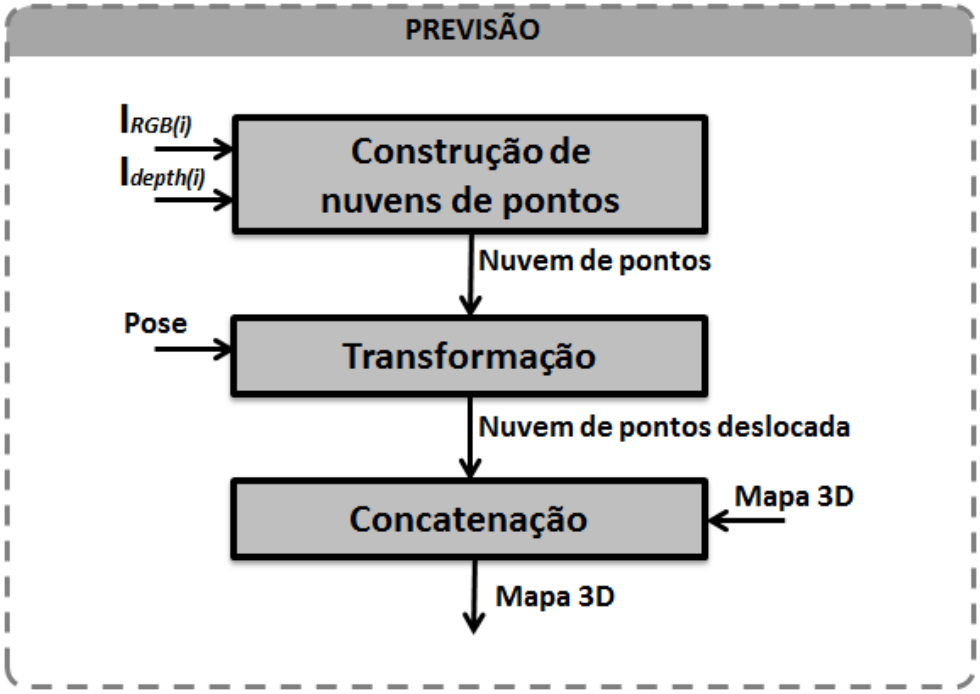


Figura 12 – Fluxograma de etapas que compõem o bloco de Previsão da Figura 7.

A Figura 13 mostra um modelo de câmera RGB-D e a as duas imagens referentes a captura de um único quadro: colorida (RGB) à esquerda e de profundidade (*depth*) à direita. São leituras como esta que o primeiro bloco, Construção de nuvens de pontos, necessita para executar sua função. Neste processo, a imagem de profundidade, que, para maioria dos sensores no mercado, pode conter valores de medições entre um e quatro metros, pode ser facilmente convertida em uma nuvem de pontos 3D, onde cada *pixel* se torna um ponto no espaço. Através da posição de cada *pixel*, cores podem ser facilmente associadas a estes pontos. No caso deste trabalho, as imagens trabalhadas possuem resolução de 640×480 *pixels*. Para evitar trabalhar com um grande número de pontos, estes são reduzidos através de um *grid* tridimensional de aresta igual a um centímetro, reduzindo a nuvem que superava 300 mil pontos para cerca de 3

mil pontos por quadro.

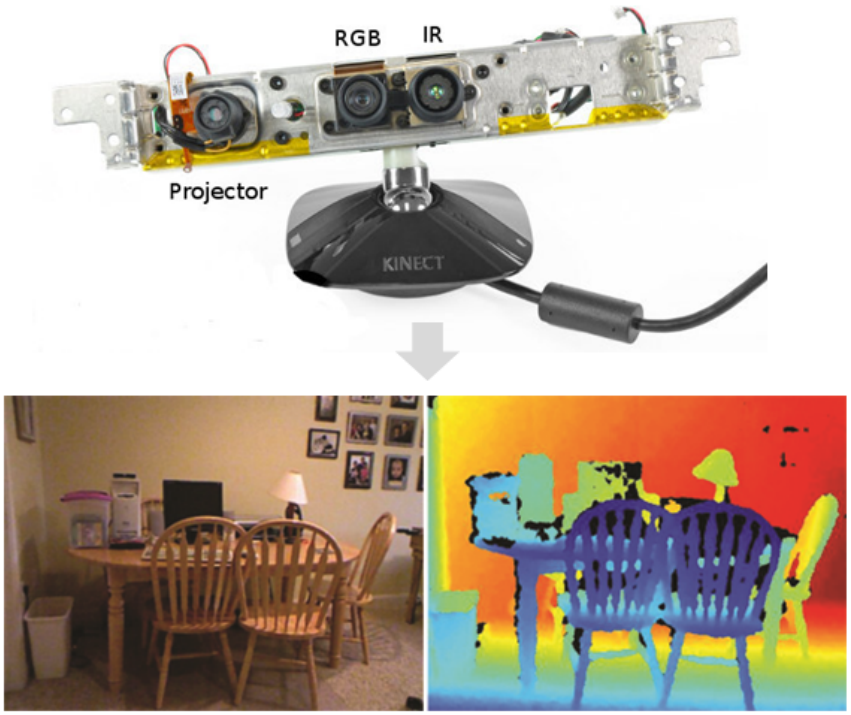


Figura 13 – *Hardware* de uma câmera RGB-D (topo) que captura tanto imagens RGB (esquerda) quanto imagens de profundidade (direita).

O segundo bloco, Transformação, recebe a posição e orientação (pose) do quadro atual em relação ao anterior, vinda da Odometria, e desloca a nuvem de pontos gerada pelo bloco anterior. Neste trabalho, o primeiro *frame* lido sempre é posicionado na origem com os valores de rotação respectivos a cada eixo zerados. Enquanto da segunda leitura em diante se faz necessário concatenar todas as poses previamente estimadas.

Por fim, a ultima etapa do processo de Previsão, chamada de Concatenação, une as nuvens de pontos deslocadas, gerando um Mapa 3D relativo a cena como um todo. A concatenação de dois quadros pode ser claramente visualizada na Figura 14. Para evitar o crescimento exponencial do número de pontos neste mapa, um *grid* tridimensional de 1 centímetro é novamente utilizado, assim, a cada quadro, concatena-se ao mapa 3D somente novos pontos. Além disso, afim de gerar um mapa globalmente consistente, o mapa gerado, e por consequência a trajetória calculada, precisam ser atualizados e reenviados para o bloco de Concatenação. O processo de atualização destas informações será explicado na próxima seção.

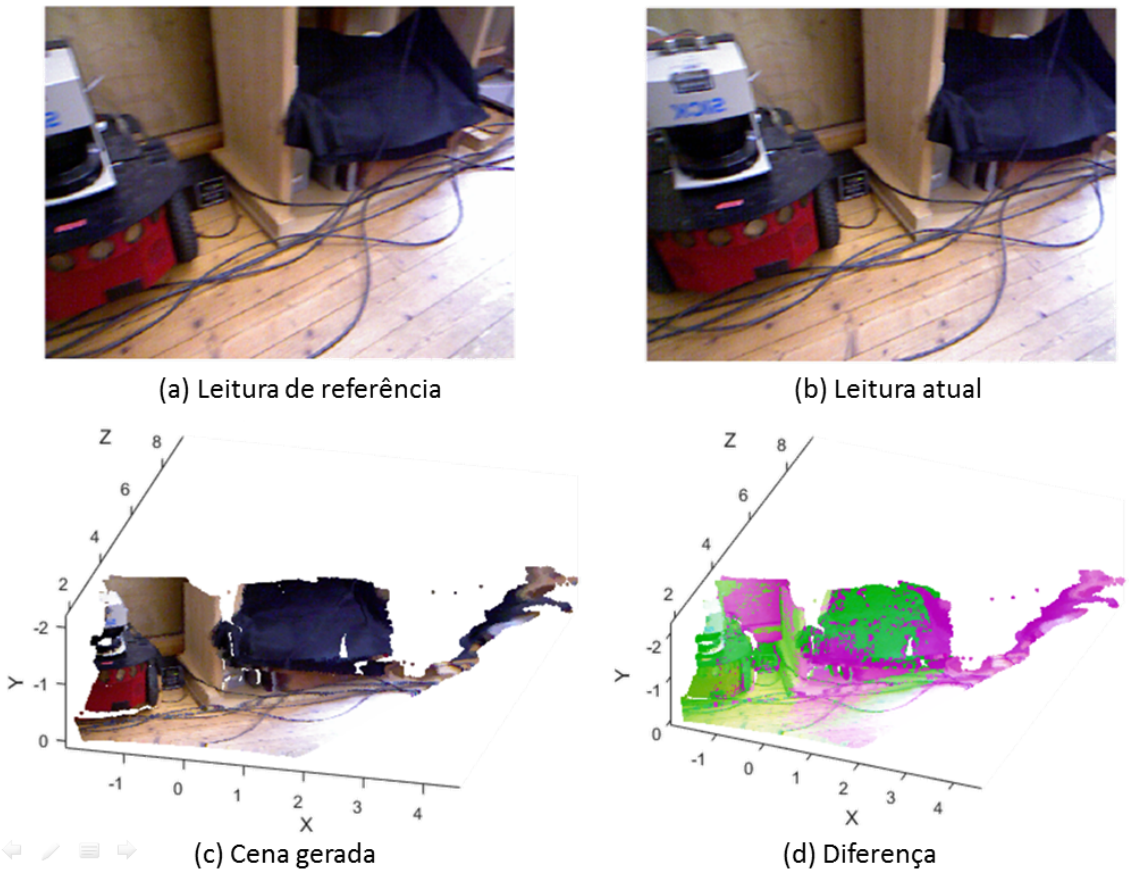


Figura 14 – A etapa de Odometria estima a movimentação feita pela câmera RGB-D entre duas capturas (a) e (b) ao calcular o movimento de corpo rígido entre elas. Com as informações de cor, profundidade e a pose estimada, pode-se gerar um mapa 3D (c), resultante da concatenação das profundidades verdes referentes a (a) e magenta a (b) que podem ser visualizadas em (d).

2.2.3 Atualização

Como dito anteriormente, o bloco de Atualização tem como principal função dar consistência global ao Mapa 3D gerado anteriormente e, consequentemente, atualizar a trajetória percorrida pela câmera até o momento. O fluxo de etapas executadas aqui pode ser verificado na Figura 15.

A primeira etapa executada, Fragmentação do mapa, fragmenta o Mapa 3D gerado pelo passo anterior: Previsão. Resultando em blocos de pontos tridimensionais coloridos referentes a um numero predefinido de quadros. Estes blocos alimentam as etapas de Ponderação por Cor e ICP.

ICP, ou *Iterative Closest Points* (BESL; MCKAY, 1992), é uma técnica de minimização de diferenças entre duas nuvens de pontos amplamente utilizada no campo de visão computacional, principalmente como uma forma de Odometria. O algoritmo iterativo revê a

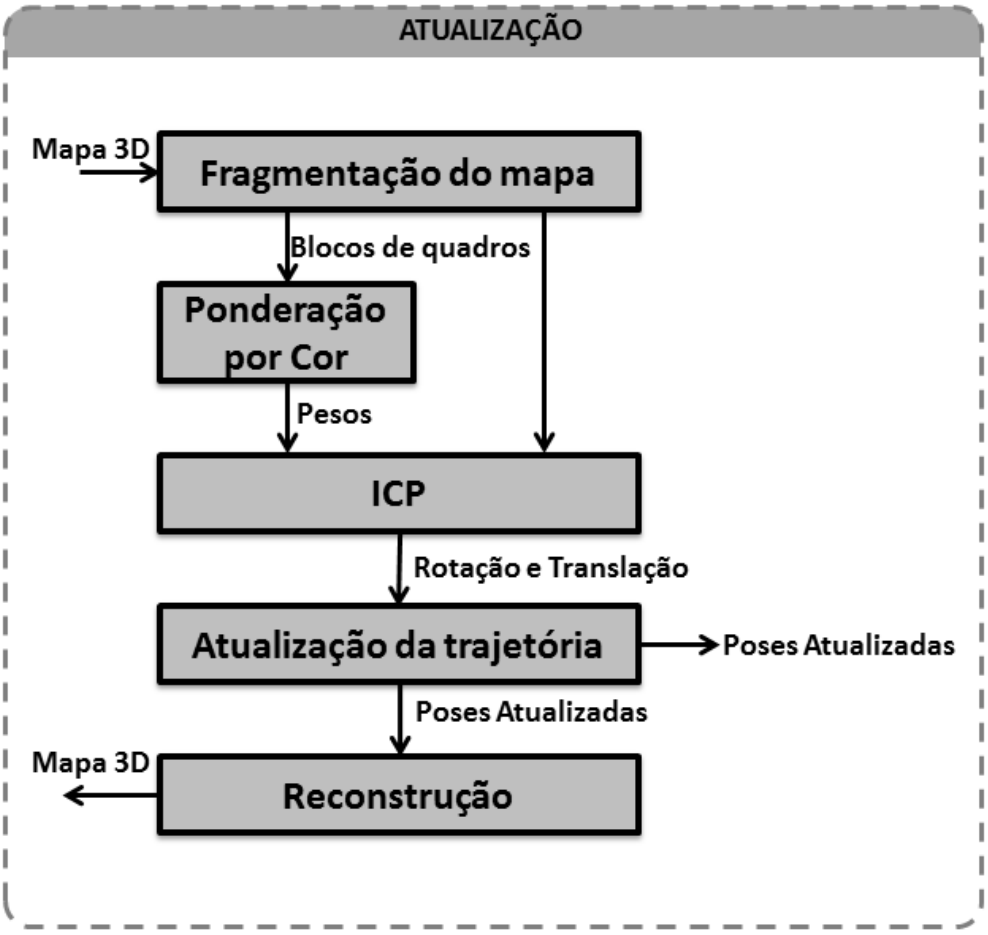


Figura 15 – Fluxograma de etapas que compõem o bloco de Atualização da Figura 7.

transformação (combinação de translação e rotação) necessária para minimizar a distância entre duas nuvens de pontos, como ilustra a Figura 16. Neste trabalho, a etapa de Ponderação por cor atribui pesos a cada par de ponto que será utilizado no algoritmo ICP, assim, diminui-se a probabilidade de que o algoritmo case pontos de cores ou tonalidades diferentes. As etapas de Ponderação por cor e ICP são executadas em todos os pares de nuvens de pontos (blocos de quadros), estando com isso proparada para detectar fechamento de *loop*. A saída do bloco ICP é o conjunto de rotações e translações entre os blocos de quadros, os quais são lidos pela etapa de Atualização da trajetória que, como o próprio nome diz, atualiza as poses referentes a cada quadro, inicialmente calculadas pela Odometria, e consequentemente também atualiza a trajetória da câmera RGB-D.

A última etapa, Reconstrução, concatena novamente os blocos de quadros, resultando em um Mapa 3D atualizado que agora deve ser utilizado pelo bloco de Previsão, no próximo ciclo.

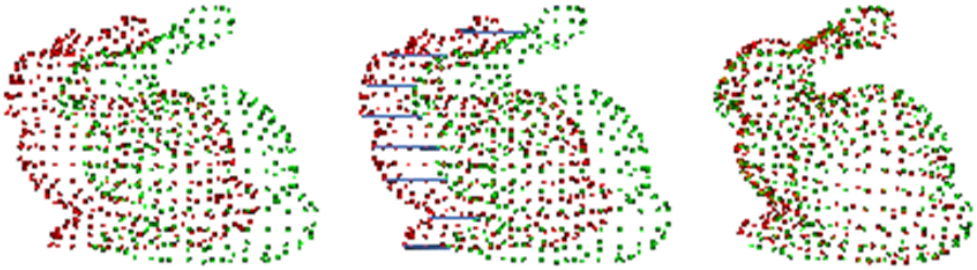


Figura 16 – Execução do algoritmo ICP recebe como entrada duas nuvens de pontos e minimiza a diferença entre as mesmas de forma densa.

2.3 PSEUDOCÓDIGOS

Esta seção apresenta os procedimentos na forma de pseudocódigos. Destaque-se que procurou-se utilizar as implementações testadas dos algoritmos clássicos como SURF, ICP, RANSAC e KLT, sendo que este trabalho se concentrou na composição e sintonia conjunta destes algoritmos para compor as tarefas de Odometria e SLAM.

2.3.1 Procedimento Principal

O procedimento principal é inicializado no Algoritmo 1 e seu *loop* é detalhado pelo Algoritmo 2. É a partir destes que as demais rotinas são chamadas. Para melhor entendimento do leitor será primeiramente apresentado o procedimento principal por completo, para só então detalhar as subrotinas que o compõe.

Primeiramente, nas linhas 1 e 2 do Algoritmo 1, são inicializados os apontadores das imagens RGB e de profundidade (*depth*) respectivamente. São através destas que a leitura dos n quadros são feitas. Esta leitura se dá ao especificar o endereço completo de onde as imagens estão armazenadas no computador. A linha 3 inicializa o contador nomeado frames, que armazena o numero de elementos lidos em uma das variáveis anteriores. Inicializa-se então, na linha 4, o vetor trajetória onde cada componente do mesmo contém uma matriz identidade 4 por 4, representando uma matriz afim de rotação e translação zeradas. As linhas 5 e 6 leem o primeiro par de imagens RGB que serão utilizadas para encontrar pontos de interesse na imagem nomeada anterior e armazená-los na variável trackpoints, na linha 7. A rotina Pontos de Interesse será explanada posteriormente.

A linha 8 inicializa um rastreador utilizando o *Computer Vision Toolbox* para MATLAB. O mesmo pede como parâmetro o número de vezes que uma imagem fornecida terá sua resolução reduzida afim de rastrear os pontos em questão e trangulá-los entre as imagens de

resoluções diferentes. Este processo será descrito em maiores detalhes durante a apresentação da rotina Transformação. O rastreador é devidamente inicializado na linha 9, que pede os parâmetros definidos na linha 8, os pontos de interesse e a imagem de onde os mesmos foram extraídos. Com este procedimento a variável *tracker* passou a armazenar também a coordenada destes pontos, extraídas na linha 10 e salvas na variável PontosRastreados. Note que o que é armazenado em *trackpoints* é somente as características SURF, chamada de SURFFeatures pelo MATLAB, e que as coordanadas destas características na imagem em questão só são extraídas uma vez inicializado o rastreador armazenado na variável *tracker*.

Sem dúvida uma das rotinas de maior importância está descrita na linha 13: Transformação. Nela são fornecidas as imagens RGB atual e anterior, assim como suas respectivas capturas de profundidade, lidas nas linhas 11 e 12. Além destes, se faz necessário enviar o rastreador e os pontos de interesse, que podem ser reinicializados pela rotina Transformação caso o número de pontos encontrados na imagem atual seja muito inferior aos pontos procurados. Para uma melhor compreensão do leitor, a rotina de transformação será descrita posteriormente. Por hora o importante é saber que a mesma retorna uma rotação e uma translação armazenadas em uma matriz afim na variável pose, e as variáveis *tracker* e PontosRastrados, que podem ter sido reinicializadas.

Note que, quando tratamos da segunda captura, a movimentação feita pelo agente e a câmera a ele associado são representados pela própria pose estimada, uma vez que a primeira captura encontra-se na origem. Sendo assim, a linha 14 armazena diretamente a pose estimada na posição 2 do vetor trajetória.

As linhas 15 a 18 inicializam as duas posições do vetor ptCloud que recebe as nuvens de pontos respectivas ao primeiro e ao segundo *frame* em suas duas primeiras posições. Ao termino do *loop* principal, que será descrito em seguida pelo Algoritmo 2, este vetor terá comprimento igual ao numero de quadros da sequência em questão. O mapa 3D é então inicializado na linha 20 ao se concatenar as nuvens de pontos dos dois primeiros quadros, fixando a primeira na origem e deslocando a segunda o relativo a pose estimada entre eles.

Por fim, as linhas 21 e 22 inicializam as variáveis *i*, que aponta número do quadro atual, e *j*, que irá ser reinicializada de 20 em 20 *loops*. A variável *j* será utilizada para apontar a posição de um vetor que armazena as ultimas 20 variações de pose. É através da média dos valores armazenados neste vetor que será possível detectar uma rotação e/ou uma translação que foge o padrão.

Algoritmo 1: Procedimento principal - Inicialização

```
1  ImRGB ← 'C:/Users.../RGB/*.PNG';
2  ImDEPTH ← 'C:/Users.../DEPTH/*.PNG';
3  frames ← length(ImRGB);
4  trajetória(1:frames) ← affine3d(eye(4)); % matriz identidade 4×4
5  anterior ← imread(ImRGB(1));
6  atual ← imread(ImRGB(2));
7  trackPoints ← Pontos de Interesse (atual, anterior);
8  tracker ← vision.PointTracker('NumPyramidLevels', 3);
9  initialize(tracker, trackPoints, anterior);
10 PontosRastreados ← tracker.Location;
11 antDepth ← imread(ImDEPTH(1));
12 atDepth ← imread(ImDEPTH(2));
13 pose, tracker, PontosRastreados ←
       Transformação (atual, atDepth, anterior, antDepth, tracker, PontosRastreados);
14 trajetória(2) ← pose;
15 ptCloud(1) ← pointCloud(antDepth);
16 ptCloud(1).Color ← anterior;
17 ptCloud(2) ← pointCloud(atDepth);
18 ptCloud(2).Color ← atual;
19 ptCloud(2) ← pctransform(ptCloud(2), trajetória(2));
20 Mapa3D ← pcmerge(ptCloud(1), ptCloud(2));
21 i ← 3;
22 j ← 1;
```

O Algoritmo 2 é uma continuação do Algoritmo 1, dedicado somente à lógica contida dentro do *loop*. A linha 1 mostra que estes comandos serão executados até que *i* alcance o numero de quadros, armazenado anteriormente em *frames*. A linha 29 incrementa o valor de *i* em uma unidade e a linha 30 encerra o comando *while* iniciado na linha 1. Note que *i* foi inicializado com o valor 3, uma vez que já foram executados os passos necessários para os dois quadros iniciais.

As linhas 2 e 3 armazenam as capturas RGB e *Depth* do quadro anterior em *anterior* e *antDepth*, respectivamente. Enquanto as linhas 4 e 5 realizam as leituras das novas capturas.

Na linha 6 é executado novamente a subrotina de Transformação, idêntica a executada na linha 13 do Algoritmo 1. Como previamente explicado, esta retorna, entre outros valores, a variação de pose entre as capturas *i* e *i-1* armazenada na forma de uma matriz afim 4×4 na variável *pose*.

Na linha 7, a variável *t* recebe a média dos três valores referentes a translação, que no MATLAB são armazenados nas três primeiras colunas da quarta linha da matriz afim *pose*. A linha 8, por sua vez, primeiramente converte a matriz de rotação, de dimensões 3×3, em *euler angles*, ou seja, em três ângulos em radianos referentes as rotações nos eixos *x*, *y* e *z* de um plano fixado na origem. A variável *R* recebe a média destes três ângulos. Os valores de *t* e *R* são então armazenados na posição *j* dos vetores *Rmean* e *tmean* nas linhas 9 e 10, respectivamente. A linha seguinte realiza uma verificação condicional em que, caso já existam 20 leituras e a estimativa atual de traslação ou rotação ultrapasse em 3 vezes a média das ultimas 20 estimativas, serão executados os comandos das linhas 12 a 15. Caso esta condicional seja positiva, se faz necessário executar novamente a rotina Pontos de Interesse, na linha 12, reinicializar o rastreador, linha 13, definindo os novos parâmetros do mesmo na linha 14. A linha 15 por sua vez executa novamente a subrotina Transformação, armazenando novos valores na variável *pose*.

As linhas 17 a 20 incrementam o apontador *j* e, caso o mesmo ultrapasse o valor 20, o reinicia.

A linha 21 realiza a concatenação das poses estimadas até o momento ao multiplicar a matriz afim *pose*, referente ao quadro atual, com o ultimo valor no vetor trajetória. O vetor trajetória armazena o percurso feito pela câmera em cada quadro *i*. Estes são os valores que serão utilizados para o calculo do RMSE, quando comparados com o *groundtruth*, como explicado no capítulo anterior.

O vetor ptCloud recebe a nuvem de pontos da posição *i* nas linhas 22 e 23 e, na linha 24, a desloca o referente a movimentação feita pela câmera até o momento. O Mapa3D é então atualizado na linha 25 através do comando pcmerge, que une duas nuvens de pontos. Neste caso as nuvens em questão são o próprio Mapa3D e a referente ao quadro atual, armazenada em ptCloud(*i*).

A linha 26 realiza outra verificação condicional, onde, caso *i* seja múltiplo de 100 ou igual ao numero de quadros, deve-se executar a rotina de Atualização, linha 27, que será descrit posteriormente.

Algoritmo 2: Procedimento principal - LOOP

```
1  while i < frames do
2      anterior ← atual;
3      antDepth ← atDepth;
4      atual ← imread(ImRGB(i));
5      atDepth ← imread(ImDEPTH(i));
6      pose, tracker, PontosRastreados ←
         Transformação (atual, atDepth, anterior, antDepth, tracker, PontosRastreados);
7      t ← mean(pose(4,1:3));
8      R ← mean(rotm2eul(pose(1:3,1:3));
9      Rmean(j) ← R;
10     tmean(j) ← t;
11     if i>20 & ((t > mean(tmean) × 3) || (R > mean(Rmean) × 3)) then
12         trackPoints ← Pontos de Interesse (atual, anterior);
13         release(tracker);
14         initialize(tracker, trackPoints, anterior);
15         pose, - ← Transformação (atual, atDepth, anterior, antDepth, tracker);
16     end
17     j ← j+1;
18     if j=21 then
19         j ← 1;
20     end
21     trajetória(i) ← affine3D(pose × trajetória(i-1));
22     ptCloud(i) ← pointCloud(atDepth);
23     ptCloud(i).Color ← atual;
24     ptCloud(i) ← pctransform(ptCloud(i), trajetória(i));
25     Mapa3D ← pcmerge(Mapa3D, ptCloud(i));
26     if mod(i, 100)=0 || i=frames then
27         Mapa3D, trajetória, ptCloud ← Atualização(trajetória, ptCloud);
28     end
29     i ← i+1;
30 end
```

Os pontos referentes a primeira leitura são sempre fixados na origem, isto faz com que a trajetória da câmera na segunda leitura seja exatamente a transformação, ou seja, a Rotação e Translação, entre o primeiro e o segundo quadro. Esta trajetória é armazenada na variável de mesmo nome, que guarda os valores de pose da câmera referente a cada *frame*. Ou seja, se existem *n frames*, devem existir *n* poses que os representem. A partir da terceira leitura, se faz necessário concatenar a transformação até o momento com a transformação atual para obter a posição global da câmera.

O mapa 3D é inicializado ao se concatenar os pontos referentes ao primeiro quadro com os pontos já deslocados do segundo quadro. Para as demais leituras, o mapa 3D deve sempre concatenar ele mesmo aos pontos referentes ao quadro atual, somente após deslocados com o valor calculado pela trajetória até o momento.

O *loop* do procedimento principal navega sobre todos os *frames* executando a mesma lógica feita para os dois primeiros. Com a exceção de que, para cada nova pose estimada através do comando **Transformação**, deve-se verificar se houve uma variação brusca de rotação e/ou translação. Esta pode ser consequência do rastreamento dos mesmos pontos por uma longa sequencia de *frames* ou simplesmente por uma movimentação da câmera em uma velocidade que foge o padrão. Em ambos os casos, se faz necessário reiniciar o rastreador e recalcular a variação de pose.

A ultima tarefa realizada em cada interação é a atualização do mapa 3D através do comando **Atualização**, que recebe a trajetória até o momento e retorna o mapa 3D atualizado. Este comando esta descrito com maiores detalhes no Algoritmo 6.

2.3.2 Pontos de Interesse

O Algoritmo 3 mostra o procedimento utilizado para a extração de pontos de interesse entre duas imagens. Este pseudocódigo recebe como entrada as imagens coloridas (RGB) dos quadros atual e anterior, e retorna pontos com características fortes da primeira imagem. Estes pontos são utilizados pelo procedimento principal, Algoritmo 1, para inicializar um rastreador.

Primeiramente, na linha 1, aplica-se SURF, um detector de características fortes, na imagem de referência. Isto pode ser feito pelo comando do Matlab *detectSURFFeatures*, que retorna um conjunto de *n* caracteristicas armazenadas na variável *PontosFortes*. O próximo passo é converter estas características SURF em coordenadas referentes a primeira imagem e a imagem

subsequente através do comando *matchFeatures*, executado nas linhas 2 e 3.

Para melhorar a acurácia, se faz necessário filtrar estas coordenadas, removendo aquelas cuja variação de posição da primeira para a segunda imagem não segue um padrão. Isto é feito na linha 4 utilizando RANSAC, técnica explicada na seção anterior. Esta etapa torna o algoritmo mais robusto para ambientes estáticos, porém piora seu desempenho caso objetos se movam durante os a captura da cena.

As coordenadas da primeira imagem, *pts1*, cujas características podem ser encontradas na imagem subsequente e seguem um mesmo padrão de deslocamento são então salvos na variável *Pontos de Interesse* na linha 5, que será utilizado pelo procedimento principal.

Algoritmo 3: Pontos de Interesse

Entrada : *atual*, *anterior*

Saída : *PontosdeInteresse*

1 *PontosFortes* ← detectSURFFeatures(anterior);
2 *pts1* ← *FeatureMatching*(*PontosFortes*, *anterior)*;
3 *pts2* ← *FeatureMatching*(*PontosFortes*, *atual)*;
4 - , *inlier* ← estimateFundamentalMatrix(pts1, pts2, 'Method', 'RANSAC') ;
5 PontosdeInteresse ← pts1(inlier, :)

2.3.3 Transformação

O procedimento que estima a pose entre duas capturas RGB-D, **Transformação**, é explicado no Algoritmo 4. Este recebe como entrada as imagens RBG atual (*i*) e anterior (*i-1*), além do rastreador *tracker* e os pontos a serem rastreados, armazenados em *PontosRastreados*. Como saída, o algoritmo fornece uma rotação e uma translação, cujas variáveis tem o mesmo nome e estão armazenadas em uma matriz afim 4×4 salva na variável *pose*, além de retornar as variáveis *tracker* e *PontosRastreados*. Note que o rastreador pode ser reiniciado no *loop* principal ao se detectar uma rotação e/ou translação que fuja a média das ultimas 20 estimativas, como explicado anteriormente. Porém existe outra situação de provável erro de estimativa, onde também se faz necessário reinicia-lo: caso o numero de pontos rastreados no quadro atual seja muito inferior ao numero de pontos armazenados em *PontosRastreados*. Por isso este procedimento solicita e retorna as variáveis *tracker* e *PontosRastreados*. Uma vez que a câmera

está em movimento, é natural que os pontos a serem rastreados sejam redefinidos de tempos em tempos.

A linha 1 salva os pontos rastreados em uma nova variável para comparação futura. Enquanto na linha 2 o comando *step* que é parte da biblioteca *Computer Vision System Toolbox* do Matlab (MATHWORKS, 2016) rastreia os novos pontos de interesse. Segundo o MATLAB, a tarefa de rastreamento é feita utilizando o algoritmo KLT, siglas para Kanade-Lucas-Tomasi. Originalmente apresentado em 1981 pelos autores que o batizam (LUCAS et al., 1981), o algoritmo funciona particularmente bem para rastrear objetos que não mudam de forma e para aqueles que apresentam textura visual.

A implementação de rastreador de pontos utilizando KLT usa pirâmides de imagem, como ilustra a Figura 17, onde cada nível tem resolução reduzida por um fator fixo em comparação com o nível inferior. Uma vez selecionado um nível de pirâmide maior que 1, o algoritmo passa a rastrear os pontos em vários níveis de resolução, começando no nível mais baixo, isto, como comprova o autor, aumenta a crença da correta localização de cada ponto de interesse. O aumento do número de níveis de pirâmide permite que o algoritmo lide com deslocamentos maiores de pontos entre os quadros, no entanto, o custo computacional também aumenta. Este trabalho utiliza três níveis de resolução, o padrão do comando *step*.

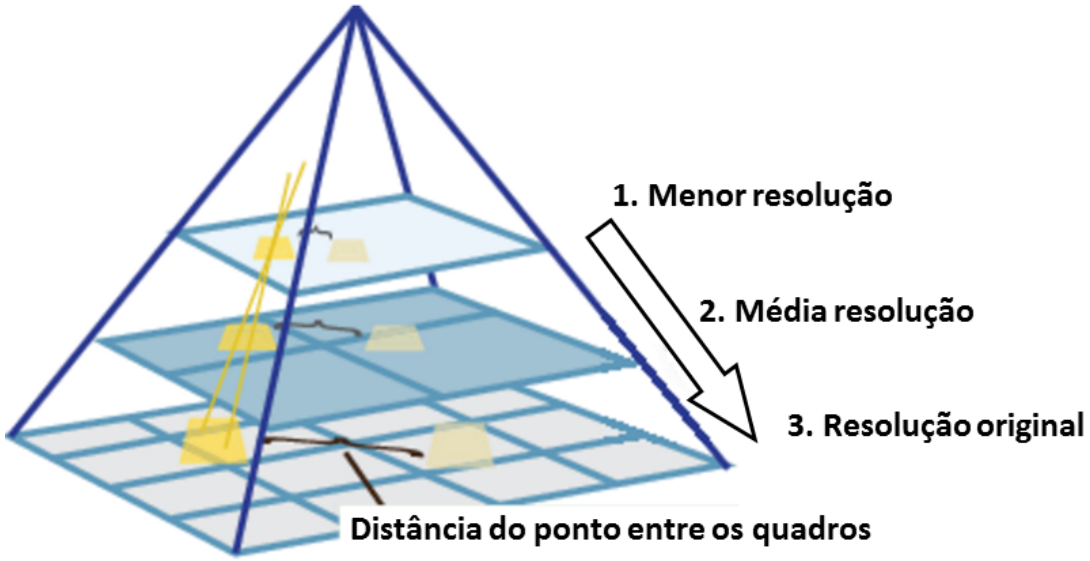


Figura 17 – Níveis de resolução utilizados pelo algoritmo Kanade-Lucas-Tomasi para rastrear pontos te interesse. A ordem de computação é feita da menor resolução para a resolução original, conforme seta indicativa.

Note que seria possível descartar o rastreador de pontos e simplesmente repetir o Algoritmo 3 para todos os pares de quadros, porém, experimentos feitos mostraram que isto

aumenta significativamente o tempo de processamento.

A linha 3 executa a verificação de perca de pontos, reiniciando o rastreador entre as linhas 4 e 6, como feito no *loop* principal do Algoritmo 2, enquanto as linha 7 e 8 executam novamente o comando *step* para as imagens *i-1* e *i*, respectivamente.

O comando *step* retorna também um vetor booleano de cumprimento igual a *Pontos-Rastreados*, nele é salvo o valor 1 caso o ponto procurado seja encontrado na imagem atual e 0 caso contrário. As linhas 10 e 11 removem os pontos não encontrados em ambas as imagens e salva os remanescentes nas variáveis *matached1* e *matched2*. Até agora possuímos dois conjuntos de coordenadas de mesmo tamanho: $2 \times n$, sendo necessário encontrar suas respectivas profundidades. Isto é feito ao ler o valor de profundidade, na imagem *depth*, nas mesmas coordenadas, feitos nas linhas 12 e 13. As linhas 14 a 17 convertem as imagens em nuvens de pontos, as projetando em um plano 3D. A quantidade de coordenadas é salva na variável *size*, na linha 18. Isto se faz necessário para armazenar em p e q, linhas 19 e 20, somente as coordenadas de interesse.

A rotação e translação podem ser então estimadas na linha 21, ao seguir os passos propostos por Sorkine (SORKINE, 2009). Estas são então armazenadas na variável *pose* no formato de matriz afim 4×4. Somente para manter a organização, estes passos foram armazenados na subrotina Minimizar, que será apresentada em sequencia.

A linha 22 retorna os pontos rastreados originais a variável *PontosRastreados* para que os mesmos possam ser utilizados corretamente na próxima iteração.

Algoritmo 4: Transformação

Entrada : *atual*, *atDepth*, *anterior*, *antDepth*, *tracker*, *PontosRastreados*

Saída : *pose*, *tracker*, *PontosRastreados*

1 *PontosRastreadosANT* ← PontosRastreados;
2 *PontosRastreados, IDs* ← **step** (*tracker*, atual);
3 **if** *length(PontosRastreados)* $\times 3 <$ *length(tracker.Location)* **then**
4 *trackPoints* ← **Pontos de Interesse** (atual, anterior);
5 release(tracker);
6 initialize(tracker, trackPoints, anterior);
7 *PontosRastreadosANT* ← **step** (*tracker*, anterior);
8 *PontosRastreados, IDs* ← **step** (*tracker*, atual);
9 **end**
10 *matched1* ← PontosRastreados(IDs, :);
11 *matched2* ← PontosRastreadosANT(IDs, :);
12 *indi* ← sub2ind(atual, matched1(: , 1), matched1(: , 2) ;
13 *indj* ← sub2ind(anterior, matched2(: , 1), matched2(: , 2) ;
14 XYZi ← pointCloud(antDepth);
15 XYZi ← XYZi.Location;
16 XYZj ← pointCloud(atDepth);
17 XYZj ← XYZj.Location;
18 *size* ← XYZi.Count;
19 p ← [XYZi(indi), XYZi(indi+size), XYZi(indi+ $2 \times$ *size*)];
20 q ← [XYZj(indj), XYZj(indj+size), XYZj(indj+ $2 \times$ *size*)];
21 Rotação, Translação ← **Minimizar**(p, q);
22 *pose* ← [*Rotação*; Translação];
23 *PontosRastreados* ← PontosRastreadosANT;

O Algoritmo 5, Minimizar, apresenta os comandos executados na linha 21 do Algoritmo 4, Transformação. O mesmo recebe como entrada dois conjuntos de coordenadas 3D, p e q, e retorna uma rotação e uma translação que os minimize através da técnica proposta por Sorkine (SORKINE, 2009).

Na subrotina Minimizar, as linhas 1 e 2 armazenas nas variáveis n e m o comprimento de p e q respectivamente. A centroide dos mesmos são salvos em *pmean* e *qmean* nas linhas 3 e 4. Enquanto as linhas 5 e 6 centraliza os conjuntos de pontos na origem. A linha 7 realiza uma verificação se uma matriz de pesos, W, foi fornecida. Caso negativo, W recebe uma matriz identidade de dimensão n na linha 8, igualando o peso igual a 1 para todas os conjuntos de pontos. A matriz de covariância 3×3 é calculada na linha 10 ao se multiplicar os conjuntos de pontos centralizados e armazenada em *cov*. A linha 11 decompõe a variável *cov* utilizando SVD e salva o primeiro e o ultimo valor nas variáveis U e V respectivamente. As linhas 12 e 13 são a aplicação da formula proposta por Sorkine para extração da rotação entre os conjuntos de pontos. Por fim, a linha 14 encontra a translação entre os pontos.

Note que a subrotina Transformação não fornece valores para W. Esta atribuição de pesos será utilizada futuramente durante o algoritmo de Atualização, onde a função minimizar irá ser chamada novamente.

Algoritmo 5: Minimizar

Entrada : p, q, W

Saída : *Rotação*, *Translação*

1 $n \leftarrow$ size(p, 1);
2 $m \leftarrow$ size(q, 1);
3 $pmean \leftarrow$ sum(p)/n;
4 $qmean \leftarrow$ sum(q)/m;
5 $pCent \leftarrow$ p - repmat(pmean, n, 1);
6 $qCent \leftarrow$ q - repmat(qmean, m, 1);
7 **if** $W == empty$ **then**
8 $\quad W \leftarrow eye(n)$;
9 **end**
10 $cov \leftarrow (pCent)^T \times W \times$(qCent);
11 *(U, -, V)* $\leftarrow$ svd(cov);
12 M $\leftarrow$ diag(ones(1, size(cov)), det(V$\times$ U^T));
13 $Rotação \leftarrow V \times M \times$U^T;
14 $Translação \leftarrow qmean^T$ - ($Rotação \times pmean$);

2.3.4 Atualização

O procedimento nomeado de *Atualização* é ilustrado pelo Algoritmo 6. Este recebe a trajetória percorrida pela câmera RGB-D até o momento e as nuvens de pontos coloridos que compõem o Mapa3D e foram previamente salvas na variável *ptCloud*, e, como o próprio nome diz, atualiza ambas as informações, as retornando para o procedimento principal. A real aplicação do algoritmo de atualização é evitar sobreposição de informação, a qual normalmente ocorre quando a câmera percorre uma área anteriormente registrada ou quando há fechamento de *loop*. Sendo assim, não se faz necessário a execução do mesmo em todas as interações de processamento. Este trabalho executa esta etapa a cada cem quadros e/ou no ultimo quadro de cada sequência.

O primeiro passo do algoritmo, na linha 1, é criar 10 fragmentos de mapa, independente do tamanho do mapa, e iniciar um *loop* para reconstruí-lo. Por exemplo, se a variável *ptCloud* possuir 200 entradas, a função Fragmentar, que será explicada posteriormente, retorna 10 fragmentos formados pela concatenação de 20 entradas cada um. Isto é feito ao estimar a rotação e translação entre cada fragmento do mapa. Como estes fragmentos são compostos por nuvens densas de pontos, utiliza-se aqui o algoritmo ICP (BESL; MCKAY, 1992), que minimiza a diferença entre conjuntos de pontos. A linha 2 armazena em *frag* o numero de entradas que compõem cada fragmento.

A linha 3 executa pcregrigid, um comando interno do MATLAB que, dados duas nuvens de pontos 3D, permite extrair dois conjuntos de coordenadas, p e q, de mair provável correspondência. Isto é feito de forma iterativa e seria o primeiro passo para todo algoritmo ICP.

Com o acréscimo de cor, além do conjunto de três coordenadas dos pontos que compõem os fragmentos de mapa, torna-se possível atribuir pesos aos pontos a serem minimizados. Cada *pixel* de uma imagem RGB é formado por três valores entre 0 e 255 referentes as tonalidades das cores vermelho, verde e azul, respectivamente. O peso nada mais é do que a subtração da média destes três valores para os dois pontos a serem casados. A linha 4 realiza esta operação e divide a média por 255, assim os pesos assumem valores entre 0 e 1. A linha 5 chama novamente a subrotina Minimizar, explicada anteriormente, porém desta vez fornecendo a matriz de pesos, W, e salvando a matriz afim contendo informações de rotação e translação na variável *pose*.

Se faz então necessário converter as transformações calculadas entre os fragmentos em uma nova trajetória da câmera RGB-D. Isto é possível ao transformar os valores armazenados

no vetor Trajetória entre frag e 2×*frag* o equivalente à *pose*. Isto é feito na multiplicação da linha 6. Enquanto as linhas 7 e 8 realizam a mesma atualização para ptCloud e Fragmento, respectivamente.

O *loop* que se inicia na linha 10 tem como objetivo realizar os mesmos passos explanados até agora para os fragmentos subsequentes.

Algoritmo 6: Atualização

Entrada : *Trajetória* e *ptCloud*

Saída : *Trajetória* e *ptCloud*

1 Fragmento(1:10) ← Fragmentar(ptCloud);
2 $frag$ ← length(Trajetória)/10;
3 *p, q* ← **pcregrigid** (Fragmento(1), Fragmento(2));
4 W ← (1-(abs(mean(p.Color^T)^T-mean(q.Color^T)^T)/255));
5 pose ← **Minimizar**(W, p, q) ;
6 Trajetória(frag:(2×frag)) ← Trajetória(frag:(2×frag))×pose;
7 ptCloud(frag:(2×frag) ← **pctransform**(ptCloud(frag:(2×frag), pose);
8 *Fragmento(2:10)* ← **pctransform**(Fragmento(2:10), pose);
9 i ← 3;
10 **while** i < *length(Fragmento)* **do**
11 *p, q* ← **pcregrigid** (Fragmento(i), Fragmento(i+1));
12 W ← (1-(abs(mean(p.Color^T)^T-mean(q.Color^T)^T)/255));
13 pose ← **Minimizar**(W, p, q) ;
14 Trajetória(frag:(i×frag)) ← Trajetória(frag:(i×frag))×pose;
15 ptCloud(frag:(i×frag) ← **pctransform**(ptCloud(frag:(i×frag), pose);
16 *Fragmento(i:10)* ← **pctransform**(Fragmento(i:10), pose);
17 i ← i+1;
18 **end**

O Algoritmo 7, nomeado Fragmentar, é chamado somente na linha 1 da rotina Atualização. Este recebe como entrada o vetor ptCloud, que armazena uma nuvem de pontos para cada captura RGB-D, e retorna 10 fragmentos de um mapa 3d denso, salvos no vetor

fragmentos.

A linha 1 recebe o comprimento de ptCloud, que deve ser igual ao numero de *frames* lidos até o momento. Enquanto o comprimento de cada fragmento é salvo na variável *n* na linha 2. O *loop* entre as linhas 4 e 11 tem como função armazenar *n* nuvens de pontos em cada *i* fragmento.

Algoritmo 7: Fragmentar

Entrada : *ptCloud*

Saída : fragmentos(1:10)

```
1  comprimento ← length(ptCloud);
2  n ← comprimento/10;
3  i ← 1;
4  while i<=10 do
5  |  j ← 1;
6  |  fragmentos(i) ← ptCloud(j);
7  |  j ← j+1;
8  |  while j<=n do
9  |  |  fragmentos(i) ← pcmerge(fragmentos(i),ptCloud(j));
10 |  end
11 end
```

3 RESULTADOS E DISCUSSÃO

Um algoritmo de SLAM é de fato uma pilha de algoritmos conforme mostrado nos capítulos anteriores. O SLAM descrito foi implementado e diversos experimentos foram realizados. Sendo que alguns dos algoritmos componentes do SLAM são parametrizados, o objetivo destes experimentos é entender os efeitos de alguns deste blocos construtivos e parâmetros sobre os resultados do SLAM. Espera-se com isso poder tirar algumas recomendações básicas para novos implementadores de SLAM geométrico. Este capítulo avalia o algoritmo implementado descrito nos capítulos anteriores utilizando o *benchmark* TUM RGB-D disponibilizado por Sturm et al (STURM et al., 2012). Em seguida, será apresentado resultados qualitativos e reconstruções em 3D de diversos ambientes internos, tanto aqueles resultantes dos experimentos com o conjunto de dados utilizados, quanto outros adquiridos por ensaios próprios.

3.1 DADOS E MÉTRICA DE DESEMPENHO

Para avaliar esta implementação foram utilizados os conjuntos de dados disponibilizadas ao público por Sturm (STURM et al., 2012), que apresenta uma pesquisa voltada para a avaliação de sistemas de SLAM Visual. Este *dataset* consiste de diversas sequencias de imagens RGB-D adquiridas a partir de uma câmera Microsoft Kinect se movimentando em ambientes diferentes, as vezes utilizando o auxilio de um robô controlado remotamente, outras movida manualmente. Juntamente com as imagens, Sturm também disponibiliza a trajetória real feita pela câmera, *groundtruth* (GT), adquirida por um sistema externo de captura de movimento de alta precisão, como ilustrado na Figura 18. A disponibilidade da trajetória real permite uma comparação precisa de diferentes abordagens SLAM. Além disso, para facilitar ainda mais esta comparação, um conjunto de ferramentas de avaliação também foi disponibilizado pelos criadores do *dataset*.

Há uma grande variedade de sequencias de capturas disponíveis para *download*, porém, a avaliação de desempenho foi feita utilizando as sequências descritas abaixo e na Tabela 1, pois estas contêm tanto movimento de translação quanto de rotação em ambientes, velocidades e percursos diferentes. Todos os conjuntos de dados foram adquiridos com uma resolução de 640×480 *pixels* a uma taxa de quadros de 30 Hz.

- **FR1/floor**: Uma varredura manual simples sobre o chão de madeira em um ambiente de escritório. Nesta sequencia, a referência do piso está constante em maioria dos quadros.

Figura 18 – Câmera (a) utilizada para criar o banco de dados TUM RGB-D utilizado na avaliação de desempenho deste trabalho, movimentada manualmente ou por um robô (b). A trajetória real foi gravada por um sistema de detecção de movimento alta precisão (c). Fonte: (STURM et al., 2012).

Tabela 1 – Características das sequencias RGB-D utilizadas.

Sequencia	Categoria	Quadros	Duração	Total navegado	Dimensões (m)
FR1/floor	SLAM Manual	1245	44.27 s	12.56	2.30×1.31 ×0.58
FR1/teddy	Reconstrução 3D	1418	50.78 s	15.70 m	2.42×2.24×1.43

Sequência ilustrada pela Figura 23.

- **FR1/teddy**: Sequência manual que realiza movimentos bruscos, tanto de translação quanto de rotação, ao redor de um grande urso de pelúcia. Sequência ilustrada pela Figura 25.

A avaliação de um sistema SLAM utilizando o *benchmark* TUM RGB-D é realizado através da medição do erro de trajetória absoluto, ou *absolut trajetory error* (ATE), entre as trajetórias estimada pelo algoritmo implementado e a *groundtruth* (GT) fornecida. Esta métrica de avaliação permite determinar a consistência global do mapa gerado ao alinhar as trajetórias e, em seguida, computar as diferenças de pose absolutas.

Como explica Sturm (STURM et al., 2012), para o cálculo do ATE, compara-se uma sequência de poses da estimativa de trajetória feita pela câmera $P_1,...,P_n \in SE(3)$ com as respectivas poses fornecidas pela trajetória GT $Q_1,...,Q_n \in SE(3)$. Assume-se que as duas trajetórias estão sincronizadas, ou seja, são igualmente amostradas e de mesmo comprimento. As poses de cada trajetória são representadas por matrizes de transformação homogêneas, que são expressas em relação a referenciais globais arbitrários, portanto deve-se primeiro alinhar as mesmas. Este alinhamento é calculado ao encontrar o movimento de corpo rígido $S \in SE(3)$ entre as trajetórias usando o método apresentado na Seção 2.1. Sendo assim, o erro de trajetória absoluto F_i, para um par de poses Q_i e P_i, é dado pela Equação 3.1, onde S é a transformação que alinha P a Q:

$$F_i = \frac{SP_i}{Q_i}. \tag{3.1}$$

Finalmente, o autor define uma métrica para comparar a trajetória estimada com o *groundtruth*, que consiste em calcular o RMSE (*Root Mean Square Error*) sobre todas as poses. Para isto utiliza-se somente a representação translacional, $t_i \in \mathbb{R}^3$, do ATE previamente calculado, $F_i = (R_i, t_i)$, como mostra a Equação 3.2:

$$RMSE(F_{1:n}) = \sqrt{(\frac{1}{n}\sum_{i=1}^{n} \|t_i\|^2)}. \tag{3.2}$$

3.2 RESULTADOS

Os ensaios feitos consistem em percorrer todos os *frames* do banco de dados em questão estimando a variação de pose da câmera entre eles e por fim calculando o RMSE da trajetória. Para comparação de desempenho, além da execução dos algoritmos descritos neste trabalho, também foi computada a variação de pose utilizando duas outras técnicas bem distintas: *Interative Closest Points* (ICP), que leva em consideração apenas imagens de profundidade (*Depth*), e *Structure from Motion* (SfM), onde são necessárias apenas imagens coloridas (RGB) para estimar a pose.

Odometria Visual

Embora odometria seja um bloco construtivo do algoritmo SLAM, em algumas aplicações ela é uma função final. Por isso interessa analisar o desempenho da odometria em separado: tanto para conhecer seu impacto sobre o SLAM quanto para conhecer seu desempenho como função autônoma. O algoritmo de Odometrial Visual implementado é comparado na Tabela 2 com outros dois algoritmos distintos: ICP e SfM. O método ICP implementado é baseado no trabalho de Besl et al (BESL; MCKAY, 1992), este identifica a transformação que minimize as diferenças entre duas nuvens de pontos de forma densa, ou seja, levando em consideração todos os pontos em questão. Como as imagens de profundidade da câmera RGB-D trabalhada gera mais de 300 mil pontos, foi criado um *grid* tridimensional para viabilizar a execução do algoritmo. Este *grid* reduz em cem vezes a quantidade de pontos, resultando em 3 mil pontos. Já o algoritmo SfM comparado consiste, em sua maioria, das mesmas etapas do algoritmo de Odometria Visual implementado, com a diferença de que a profundidade é estimada por triangulação, como no trabalho de Benseddik et al (BENSEDDIK et al., 2014).

Note da Tabela 2 que a odometria implementada supera as duas técnicas usadas para comparação. Isso é devido a que os métodos comparados não utilizarem todos os recursos disponíveis, é de se esperar que a odometria RGB-D obtenha um desempenho melhor.

Tabela 2 – Comparação de RMSE entre algoritmos de Odometria Visual testados.

Algoritmo	RMSE FR1/floor	RMSE FR1/teddy
Odometria Visual implementada	0.283	0.103
ICP (BESL; MCKAY, 1992)	0.324	0.549
SfM (BENSEDDIK et al., 2014)	0.431	0.115

Tabela 3 – Comparação do RMSE com outros métodos baseados em imagens RGB-D

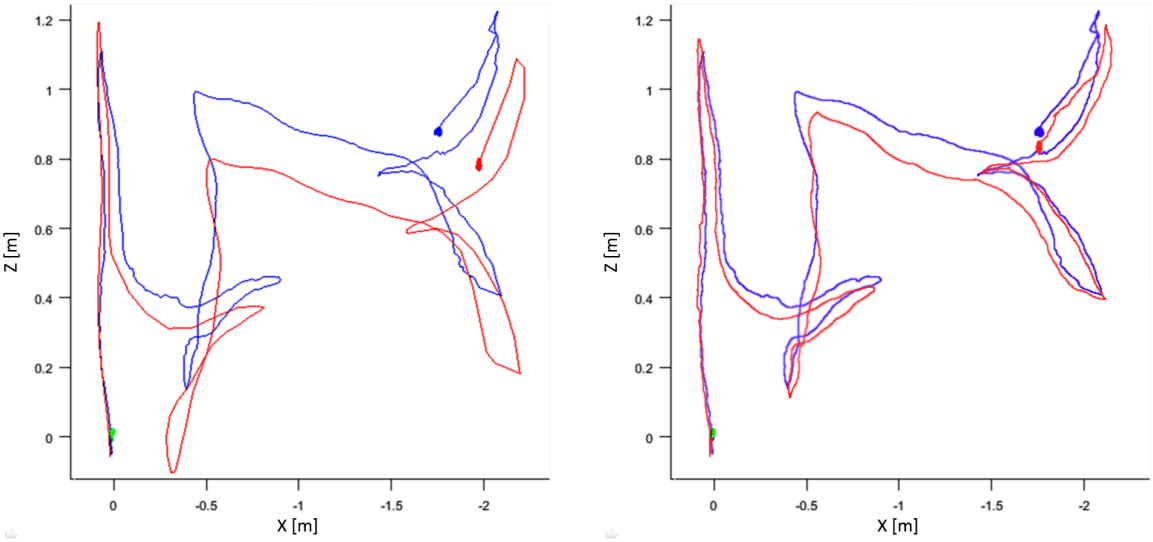
Algoritmo	RMSE FR1/floor	RMSE FR1/teddy
SLAM implementado	0.117	0.055
ICP baseado em intensidade (LI; LEE, 2016)	0.100	0.048
RGB+D (KERL et al., 2013)	falhou	0.060
RGB+D+KF (KERL et al., 2013)	0.090	0.067
RGB+D+KF+Opt (KERL et al., 2013)	0.232	0.043

SLAM

A Tabela 3 apresenta a comparação do SLAM implementado com outros métodos de SLAM baseados em profundidade e cor. Uma vez que existem diversos fatores que podem gerar uma alteração nos resultados, como configurações do computador e linguagem de programação utilizada, tal comparação é apenas uma forma indireta de mensurar os desempenhos das técnicas estado da arte mencionadas através dos números publicados. O método de SLAM Visual implementado não fornece a melhor precisão para as sequências testadas, porém obteve resultados satisfatórios e dentro da média.

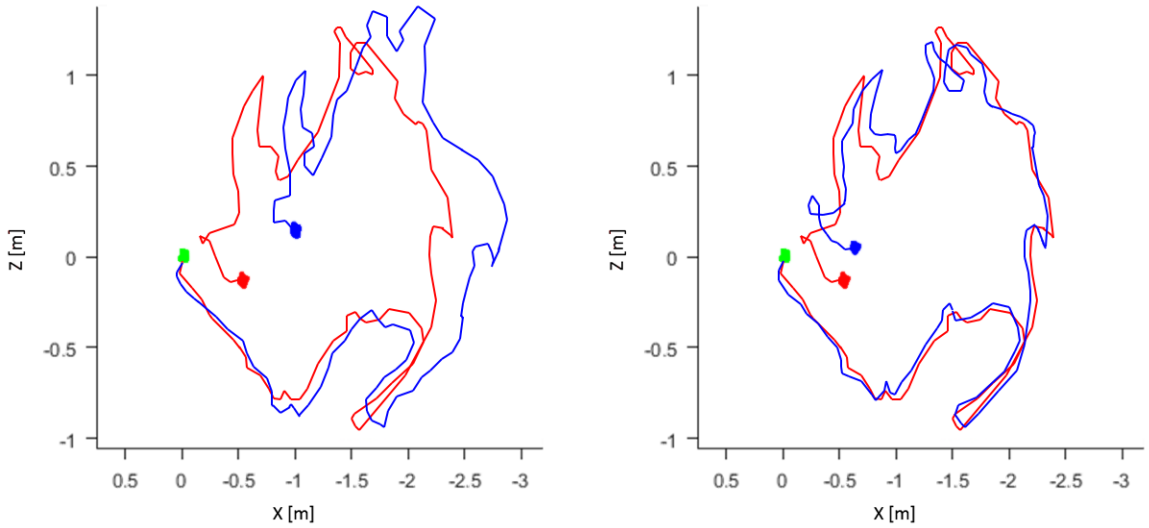
A Figura 19 compara a vista superior das trajetórias real (*groundtruth*), em azul, com as trajetórias estimadas utilizando os algoritmos implementados de Odometria Visual (a) e SLAM Visual (b) para a sequência de imagens FR1/floor fornecida por Sturm (STURM et al., 2012). Enquanto a Figura 20 ilustra a mesma comparação para a sequência FR1/teddy, sendo que nesta a trajetória real está em vermelho e a estimada em azul. Em ambos os ensaios, a trajetória real permanece inalterada, enquanto a estimada apresenta uma melhoria de acurácia de (a) para (b). Isto também pode ser visto ao se analisar os valores relativos as linhas nomeadas Odometria Visual implementada e Slam implementado das Tabelas 2 e 3, respectivamente.

Embora importante em aplicações reais, o foco deste trabalho não foi o desempenho computacional e sim o teste das funcionalidades. Isso porque a literatura dá conta de que implementações em Matlab podem ser de 2 a 20 vezes mais lentas que implementações em em C, por exemplo, dependendo da máquina, da versão do software, da eficiência da programação, etc. A comparação, embora possível, fica muito difícil. Por isso, apesar de apresentarmos tempos de processamento na Tabela 4, estaremos mais interessados na proporção do tempo total consumido por cada método. As três etapas principais do algoritmo SLAM implementado,



(a) Odometria Visual (b) SLAM Visual

Figura 19 – Trajetória real (em azul) é comparada com a trajetória estimada (em vermelho) utilizando Odometria Visual (a) e SLAM Visual (b) para a sequencia de imagens FR1/floor.



(a) Odometria Visual (b) SLAM Visual

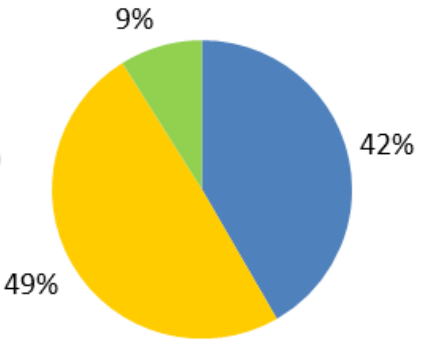
Figura 20 – Trajetória real (em vermelho) é comparada com a trajetória estimada (em azul) utilizando Odometria Visual (a) e SLAM Visual (b) para a sequencia de imagens FR1/teddy.

Odometria, Previsão e Atualização, foram apresentadas no Capítulo 1 pela Figura 7. As rotinas que compõem estas etapas são apresentadas pelas Figuras 11, 12 e 15 respectivamente. Os tempos de execução de cada subrotina do algoritmo SLAM para a sequencia de imagens do *dataset* FR1/floor, que possui 1245 quadros, podem ser vistos na Figura 21. Ela também mostra um gráfico de consumo de tempo de processamento para cada etapa principal em (a) e uma tabela com os tempos totais em (b). Note que a etapa mais significativa é a de Previsão, com 49% do

tempo total, seguida pela Odometria com 42% e por ultimo a Atualização, que, apesar de possuir tempos individuais significativos, somente contribui com 9% do tempo de processamento total devido ao fato desta rotina ser executada poucas vezes. Em (b) é possível notar que o tempo de processamento para todos os 1245 *frames* é de 1083 segundos, o que dá uma média de 0,87 segundos por *frame*.

Rotina	Tempo total (s)	Vezes acionado	tempo medio (s)
Odometria			
SURF	29,96	107	0,28
Feature matching	20,33	107	0,19
ransac	22,47	107	0,21
feature tracking	135,2	1352	0,1
Estimativa de pose	243,36	1352	0,18
Previsão			
construção de nuvens de pontos	112,05	1245	0,09
Transformação dos pontos	186,75	1245	0,15
concatenação	236,55	1245	0,19
Atualização			
fragmentação do mapa	7,41	13	0,57
ponderação por cor	34,71	13	2,67
ICP	24,96	13	1,92
atualização da trajetória	11,96	13	0,92
reconstrução	17,81	13	1,37

(a)



(b)

	Tempo total (s)
Odometria	451,32
Previsão	535,35
Atualização	96,85
TOTAL	1083,52
Numero de frames	1245
Tempo/frame	0,87

Figura 21 – Tempo de processamento referente a cada etapa de execução do algoritmo SLAM implementado para a sequência de imagens FR1/floor, que possui 1242 quadros. (a) mostra o consumo de processamento para cada etapa principal. (b) mostra os valores totais para cada etapa principal, o tempo total gasto para computar todos os *frames* e o tempo médio por *frame*.

A Tabela 4 compara o tempo médio de execução do algoritmo implementado com os algoritmos de outros autores que utilizam as mesmas sequências de imagens de Sturm (STURM

Tabela 4 – Comparação de tempo médio de processamento por quadro × configurações de *hardware* entre contribuições que utilizaram o mesmo *dataset* deste trabalho. Legendas: [1] Slam Visual implementado, [2] ICP baseado em intensidade (LI; LEE, 2016), [3] RGB+D+KF (KERL et al., 2013), [4] ICP+RGB-D (WHELAN et al., 2013) e [5] *Inverse depth* (GUTIÉRREZ-GÓMEZ et al., 2015).

Algoritmo	Tempo médio (ms)	CPU	GPU	SO	Framework
[1]	875	i3-4005U @ 1.7GHz	-	Windows	Matlab
[2]	12	i7-4790K @4.0GHz	-	Linux	ROS
[3]	32	i7-2600 @3.4GHz	-	Linux	ROS
[4]	18	i7-3960X @3.3GHz	NVIDIA GeForce 680GTX	Linux	-
[5]	47	i5-2500 @3.3GHz	NVIDIA GeForce 660GTX	Linux	Kinect Fusion

et al., 2012). Além dos tempos, a tabela também compara o poder de processamento dos computadores onde os ensaios foram realizados, neste caso, a demarcação '-' no campo GPU significa que a mesma não é utilizada no algoritmo em questão. É claro visualizar que o algoritmo implementado possui um tempo de processamento que foge à média, estando bem distante de ser executado em tempo real, ou seja, 30 quadros por segundo ou 0.03 segundos por quadro. Apesar do poder de processamento do computador executado ser inferior ao comparado, existem outros fatores que aumentam ainda mais este tempo:

- Execução em um ambiente de programação gráfico, Matlab, e com ele a impossibilidade de executar tarefas em paralelo, utilizando dois cores da CPU.
- Utilização de *toolbox* visuais que consomem muita memória.
- Construção e atualização de um mapa denso a cada passo.

Algumas alterações de parâmetros podem ser feitas para melhorar o tempo de processamento, porém sacrificando acurácia. O gráfico de tempo de processamento versus acurácia, impresso na Figura 22, mostra três possíveis alterações de parâmetros, de legendas 2, 3 e 4, e duas combinações de alterações. Neste gráfico, a legenda 1 representa os valores de tempo de processamento e RMSE reportados nas Tabelas 3 e 4.

A etapa de odometria implementada executa SURF, *Feature Matching* e RANSAC antes de inicializar um rastreador (*Feature Tracking*). Estas etapas são executadas na varredura inicial e depois somente quando acionados por uma condicional que detecta uma rotação e/ou uma translação que fogem o padrão. Enquanto a condicional não detecta isto, a estimativa de pose é feita somente com informações fornecidas pelo *Feature Tracking*, o que acelera o

processamento. A primeira alteração, de legenda 2, propõe dobrar os multiplicadores que definem o que é uma rotação e uma translação brusca, executando as etapas iniciais menos vezes. Houve assim uma redução de tempo de processamento e uma perca de acurácia.

A legenda 3 propõe remover o filtro RANSAC do fluxo de etapas da Odometria. Isto acelera as etapas de identificação de pontos característicos a serem rastreados, porém, como ilustra a Figura 21, esta rotina é executada, em média, somente uma vez a cada dez *frames*. A remoção do filtro piora a acurácia.

Um dos parâmetros fornecidos ao detector de manchas SURF é a resolução desejada, quanto maior a resolução, maior o numero de respostas a rotina fornece. A segunda alteração, de legenda 4, propõe reduzir a resolução em 30%, fazendo com que todas as rotinas subsequentes trabalhem com menos pontos e, portanto, reduzindo o tempo de processamento. Há uma perca de acurácia maior, quando comparada com a alteração anterior.

Por ultimo, a alteração de numero 5 propõe reduzir o tamanho das imagens trabalhadas pela metade, o que acarreta em uma redução significativa de tempo de processamento em todas as rotinas do algoritmo SLAM implementado. Note que estas imagens foram reduzidas anteriormente e que o tempo impresso não soma o tempo levado na compactação das mesmas.

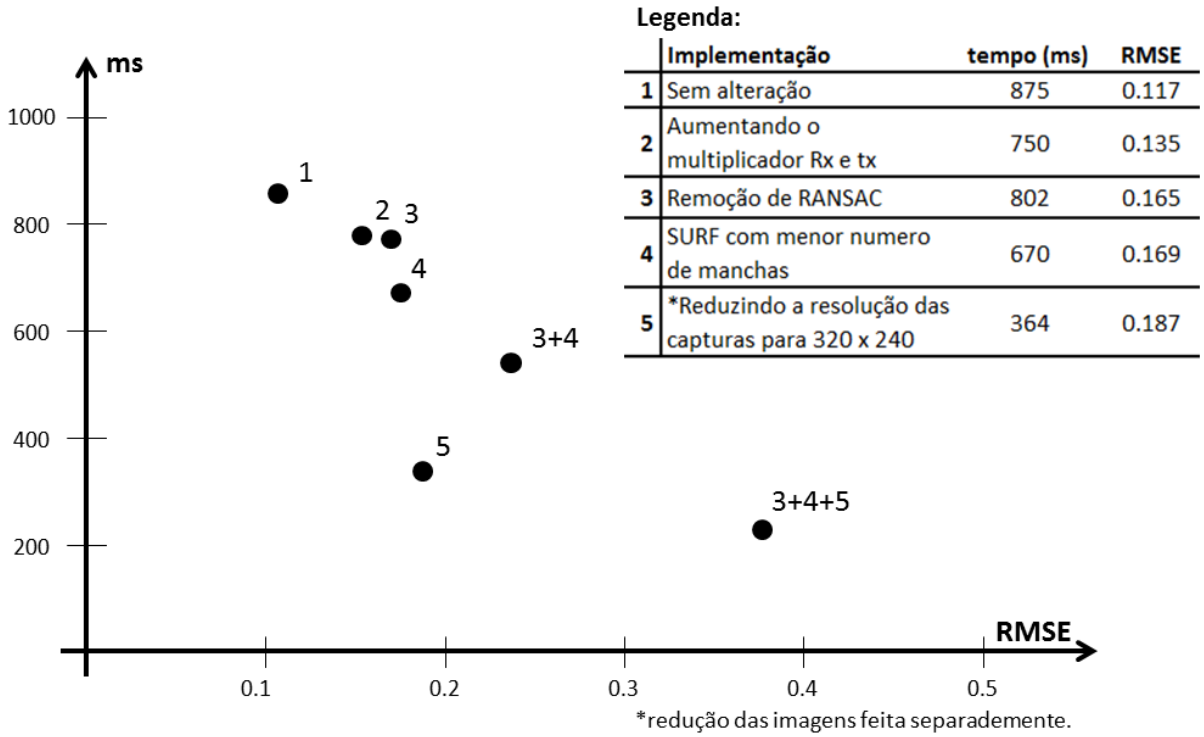
Foram também colhidos tempo e RMSE da soma das alterações de legenda 3 e 4, impressa no gráfico como 3+4, e a soma destas duas com a alteração 5, impressa 3+4+5.

Por concatenação de todos os movimentos individuais, a trajetória total da câmera e do agente a ela associado pode ser estimada. Através da trajetória e dos dados adquiridos pela câmera RGB-D é possível a criação de densas representações 3D de ambientes. Esta seção mostra os resultados obtidos nos ensaios utilizando as sequencias de imagens FR1/teddy e FR1/floor.

Todas as sequências foram adquiridas com um sensor RGB-D Microsoft Kinect 360 de movimentado manualmente com uma resolução de 640×480 *pixels*. A taxa de captura foi de apenas 3 Hz, devido às limitações de *hardware* do laptop utilizado para adquirir os conjuntos de dados. Devido a incapacidade de reprodução em tempo real, como explica a seção anterior.

Os experimentos foram realizados no sistema operacional Windows 10, rodando em um notebook com um processador de 1.7 GHz Intel Core i3 x64 e 4 GB de RAM. O sistema é implementado no Matlab versão R2016a utilizando o pacote *Computer Vision System Toolbox* que foi fundamental para alcançar o desempenho relatado.

As Figuras 23 e 25 ilustram parcialmente as sequências FR1/floor e FR1/teddy,



	Implementação	tempo (ms)	RMSE
1	Sem alteração	875	0.117
2	Aumentando o multiplicador Rx e tx	750	0.135
3	Remoção de RANSAC	802	0.165
4	SURF com menor numero de manchas	670	0.169
5	*Reduzindo a resolução das capturas para 320 x 240	364	0.187

Figura 22 – Gráfico de compromisso Acurácia *versus* tempo de processamento.

respectivamente, utilizadas na seção anterior para avaliação de desempenho. Os mapas gerados pelo algoritmo de SLAM Visual implementado podem ser visualizados nas Figuras 24 e 26.

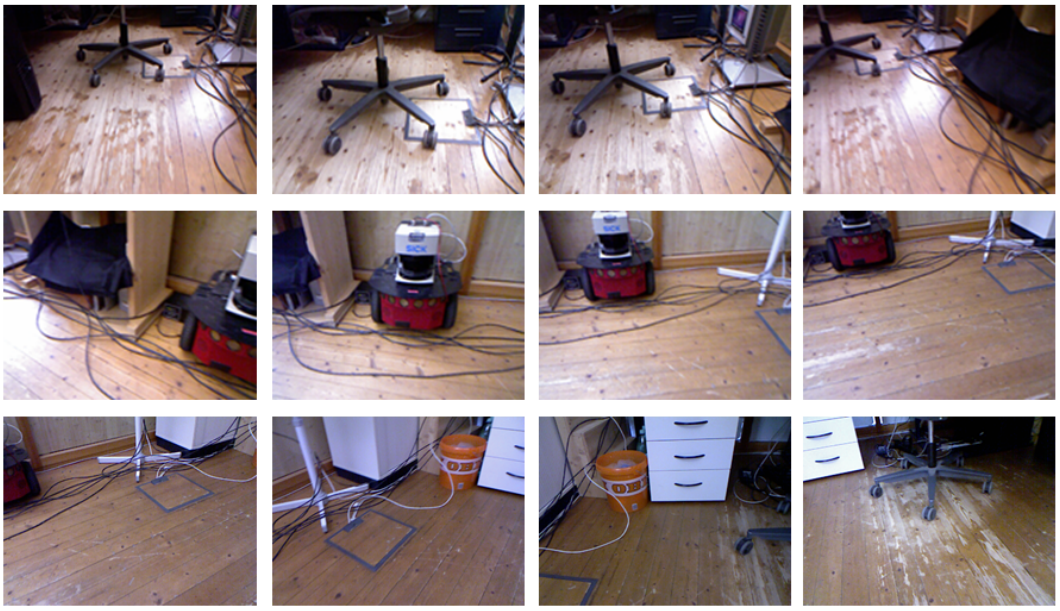


Figura 23 – Sequencia de imagens RGB adquiridas no *dataset* FR1/floor.

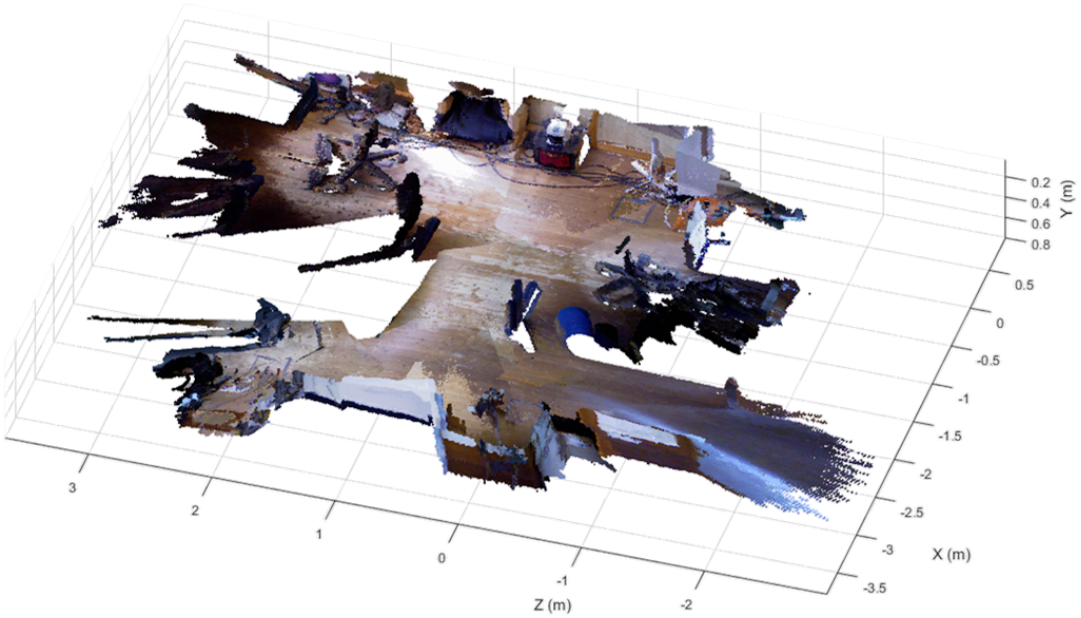


Figura 24 – Vista perspectiva da representação 3D do ambiente navegado na sequência FR1/floor.

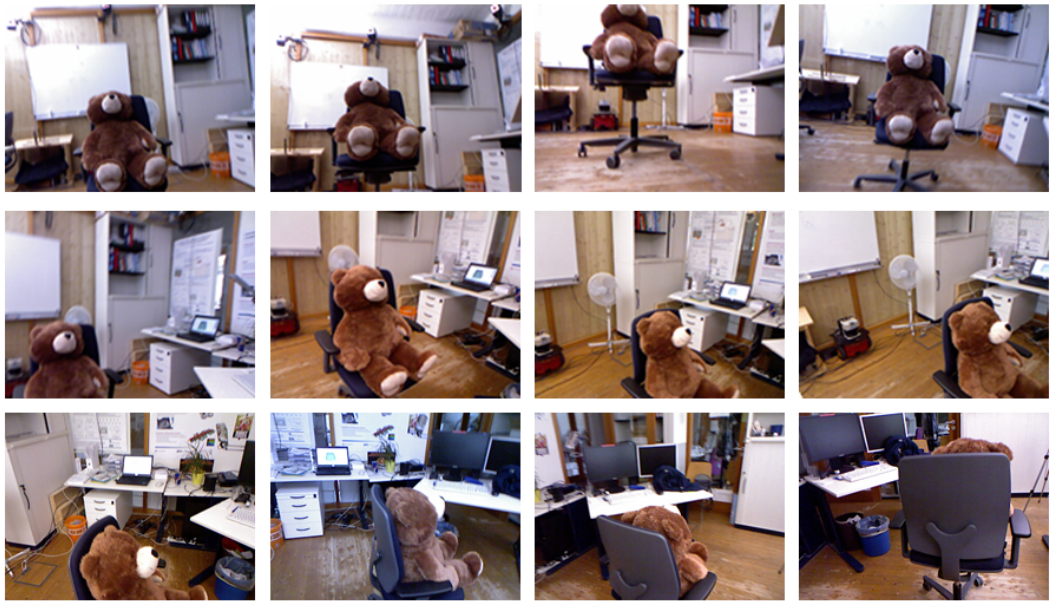


Figura 25 – Sequencia de imagens RGB adquiridas no *dataset* FR1/teddy.

3.3 DISCUSSÃO

Nas seções anteriores foram realizados uma avaliação dos sistemas de Odometria e SLAM implementados utilizando dados publicamente disponíveis e a comparação dos resultados alcançados com outras contribuições, quando possível e com as ressalvas devidas. Os ensaios realizados mostram que, apesar de mais lento que os outros trabalhos comparados, o sistema implementado gera resultados globalmente consistentes pois possui acurácia dentro da média.

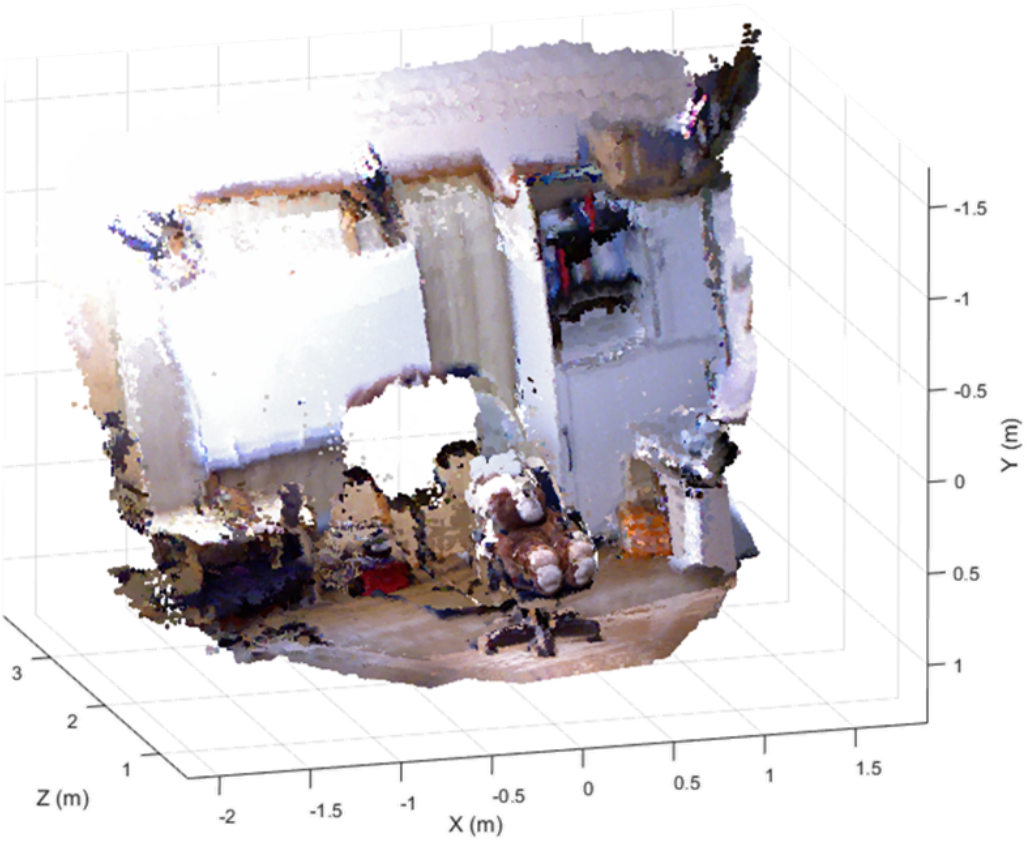


Figura 26 – Vista perspectiva da representação 3D do ambiente navegado na sequência FR1/teddy.

Embora tenham sido mostrados os tempos de processamento, este não foi o foco deste trabalho, A rotina implementada funcionaria bem na taxa de 1 *frame* por segundo. Na implementação foram utilizados apenas os algoritmos componentes padrões (*standards*). Entretanto, quase todos eles já possuem versões mais eficientes resultantes de investigações de outros pesquisadores. Por limitações de tempo para esta dissertação foram avaliados apenas os algoritmos *standards*.

O gráfico da Figura 21 pode ser útil a um implementador pois mostra um compromisso entre acurácia, representada pelo RMSE, e tempo de processamento. Nesse caso, como todos os pontos do gráfico foram obtidos na mesma máquina os dados são válidos como comparação. Se o implementador utilizar uma máquina mais potente isso muda a escala dos eixos mas não muda a posição relativa.

Dois mecanismos adicionais presentes em algumas aplicações de SLAM não foram estudadas neste trabalho: detecção de *loop* e recuperação da trajetória em caso de falha de rastreamento. A detecção de *loop* detecta sobreposição da trajetória e é comprovado que melhora substancialmente o resultado do SLAM.

4 CONCLUSÃO

O trabalho dessa dissertação consistiu na implementação e avaliação de um SLAM Visual Geométrico programado em Matlab. A implementação é organizada em três blocos principais: Odometria, Previsão e Atualização. As contribuições deste trabalho podem ser assim resumidas:

- O trabalho oferece um roteiro de implementação de SLAM Visual RGB-D compreensível, altamente estruturada e de fácil reprodução;
- Apresenta um gráfico de compromisso no plano acurácia x tempo de processamento útil para um implentador que deseje prever o peso relativo de cada algoritmo componente.

Há um número grande de melhorias já disponíveis na literatura que um implementador pode utilizar a partir deste trabalho. Elas podem ser classificadas em dois tipos. Primeiro, algoritmos de SLAM podem se utilizar de detecção de loop para recalcular a trajetória e também de recuperação do rastreamento em caso de falha. Segundo, cada um dos algoritmos componentes já possue versões aperfeiçoadas apresentadas na literatura. Essas melhorias não foram abordadas neste trabalho mas são apontadas como trabalhos futuros.

REFERÊNCIAS

AGRAWAL, M.; KONOLIGE, K.; BLAS, M. R. Censure: Center surround extremas for realtime feature detection and matching. In: SPRINGER. **European Conference on Computer Vision**. [S.l.], 2008. p. 102–115.

AHN, S.; CHOI, J.; DOH, N. L.; CHUNG, W. K. A practical approach for ekf-slam in an indoor environment: fusing ultrasonic sensors and stereo camera. **Autonomous robots**, Springer, v. 24, n. 3, p. 315–335, 2008.

BAKER, S.; MATTHEWS, I. Lucas-kanade 20 years on: A unifying framework. **International journal of computer vision**, Springer, v. 56, n. 3, p. 221–255, 2004.

BAY, H.; TUYTELAARS, T.; GOOL, L. V. Surf: Speeded up robust features. In: **Computer vision–ECCV 2006**. [S.l.]: Springer, 2006. p. 404–417.

BENSEDDIK, H. E.; DJEKOUNE, O.; BLHOCINE, M. Sift and surf performance evaluation for mobile robot-monocular visual odometry. **Journal of Image and Graphics**, v. 2, n. 1, p. 70–76, 2014.

BESL, P. J.; MCKAY, N. D. Method for registration of 3-d shapes. In: INTERNATIONAL SOCIETY FOR OPTICS AND PHOTONICS. **Robotics-DL tentative**. [S.l.], 1992. p. 586–606.

CHETVERIKOV, D.; SVIRKO, D.; STEPANOV, D.; KRSEK, P. The trimmed iterative closest point algorithm. In: IEEE. **Pattern Recognition, 2002. Proceedings. 16th International Conference on**. [S.l.], 2002. v. 3, p. 545–548.

CHOSET, H.; NAGATANI, K.; LAZAR, N. A. The arc-transversal median algorithm: a geometric approach to increasing ultrasonic sensor azimuth accuracy. **IEEE Transactions on Robotics and Automation**, IEEE, v. 19, n. 3, p. 513–521, 2003.

DAVISON, A. J. Real-time simultaneous localisation and mapping with a single camera. In: IEEE. **Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on**. [S.l.], 2003. p. 1403–1410.

DELLAERT, F.; FOX, D.; BURGARD, W.; THRUN, S. Monte carlo localization for mobile robots. In: IEEE. **Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on**. [S.l.], 1999. v. 2, p. 1322–1328.

DOUCET, A.; FREITAS, N. D.; MURPHY, K.; RUSSELL, S. Rao-blackwellised particle filtering for dynamic bayesian networks. In: MORGAN KAUFMANN PUBLISHERS INC. **Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence**. [S.l.], 2000. p. 176–183.

DURRANT-WHYTE, H.; BAILEY, T. Simultaneous localization and mapping: part i. **IEEE robotics & automation magazine**, IEEE, v. 13, n. 2, p. 99–110, 2006.

FISCHLER, M. A.; BOLLES, R. C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. **Communications of the ACM**, ACM, v. 24, n. 6, p. 381–395, 1981.

FOLEY, J. D.; DAM, A. V. et al. **Fundamentals of interactive computer graphics**. [S.l.]: Addison-Wesley Reading, MA, 1982. v. 2.

FRAUNDORFER, F.; SCARAMUZZA, D. Visual odometry: Part i: The first 30 years and fundamentals. **IEEE Robotics and Automation Magazine**, v. 18, n. 4, p. 80–92, 2011.

FRAUNDORFER, F.; SCARAMUZZA, D. Visual odometry: Part ii: Matching, robustness, optimization, and applications. **IEEE Robotics & Automation Magazine**, IEEE, v. 19, n. 2, p. 78–90, 2012.

GUTIÉRREZ-GÓMEZ, D.; MAYOL-CUEVAS, W.; GUERRERO, J. Inverse depth for accurate photometric and geometric error minimisation in rgb-d dense visual odometry. In: IEEE. **2015 IEEE International Conference on Robotics and Automation (ICRA)**. [S.l.], 2015. p. 83–89.

HARRIS, C. G.; PIKE, J. 3d positional integration from image sequences. **Image and Vision Computing**, Elsevier, v. 6, n. 2, p. 87–90, 1988.

HENRY, P.; KRAININ, M.; HERBST, E.; REN, X.; FOX, D. Rgb-d mapping: Using kinect-style depth cameras for dense 3d modeling of indoor environments. **The International Journal of Robotics Research**, SAGE Publications, v. 31, n. 5, p. 647–663, 2012.

HOCHDORFER, S.; SCHLEGEL, C. 6 dof slam using a tof camera: The challenge of a continuously growing number of landmarks. In: IEEE. **Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on**. [S.l.], 2010. p. 3981–3986.

HOPPE, C.; PIRKER, K.; RÜTHER, M.; BISCHOF, H. **Large-scale robotic SLAM through visual mapping**. [S.l.]: na, 2010.

KERL, C.; STURM, J.; CREMERS, D. Dense visual slam for rgb-d cameras. In: IEEE. **2013 IEEE/RSJ International Conference on Intelligent Robots and Systems**. [S.l.], 2013. p. 2100–2106.

KIM, S.; OH, S.-Y. Slam in indoor environments using omni-directional vertical and horizontal line features. **Journal of Intelligent and Robotic Systems**, Springer, v. 51, n. 1, p. 31–43, 2008.

LI, S.; LEE, D. Fast visual odometry using intensity-assisted iterative closest point. **IEEE Robotics and Automation Letters**, IEEE, v. 1, n. 2, p. 992–999, 2016.

LOURAKIS, M. I.; ARGYROS, A. A. Sba: A software package for generic sparse bundle adjustment. **ACM Transactions on Mathematical Software (TOMS)**, ACM, v. 36, n. 1, p. 2, 2009.

LOWE, D. G. Distinctive image features from scale-invariant keypoints. **International journal of computer vision**, Springer, v. 60, n. 2, p. 91–110, 2004.

LUCAS, B. D.; KANADE, T. et al. An iterative image registration technique with an application to stereo vision. In: **IJCAI**. [S.l.: s.n.], 1981. v. 81, n. 1, p. 674–679.

MA, Y.; SOATTO, S.; KOSECKA, J.; SASTRY, S. **An Invitation to 3D Vision: From Images to Models Springer Verlag**. [S.l.]: December, 2003.

MAHON, I.; WILLIAMS, S. B.; PIZARRO, O.; JOHNSON-ROBERSON, M. Efficient view-based slam using visual loop closures. **IEEE Transactions on Robotics**, IEEE, v. 24, n. 5, p. 1002–1014, 2008.

MATHWORKS. **vision.PointTracker System object**. 2016. Disponível em: <https://se.mathworks.com/help/vision/ref/vision.pointtracker-class.html>.

MILELLA, A.; SIEGWART, R. Stereo-based ego-motion estimation using pixel tracking and iterative closest point. In: IEEE. **Computer Vision Systems, 2006 ICVS'06. IEEE International Conference on**. [S.l.], 2006. p. 21–21.

MONTEMERLO, M.; THRUN, S.; KOLLER, D.; WEGBREIT, B. et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In: **Aaai/iaai**. [S.l.: s.n.], 2002. p. 593–598.

MORAVEC, H. P. **Obstacle avoidance and navigation in the real world by a seeing robot rover.** [S.l.], 1980.

MUJA, M.; LOWE, D. G. Fast matching of binary features. In: IEEE. **Computer and Robot Vision (CRV), 2012 Ninth Conference on**. [S.l.], 2012. p. 404–410.

NISTÉR, D.; NARODITSKY, O.; BERGEN, J. Visual odometry. In: IEEE. **Computer Vision and Pattern Recognition, 2004. CVPR 2004. Proceedings of the 2004 IEEE Computer Society Conference on**. [S.l.], 2004. v. 1, p. I–652.

PEREIRA, N. S.; CARVALHO, C. R.; THÉ, G. A. Point cloud partitioning approach for icp improvement. In: IEEE. **Automation and Computing (ICAC), 2015 21st International Conference on**. [S.l.], 2015. p. 1–5.

ROSTEN, E.; DRUMMOND, T. Machine learning for high-speed corner detection. In: SPRINGER. **European conference on computer vision**. [S.l.], 2006. p. 430–443.

SHI, J.; TOMASI, C. Good features to track. In: IEEE. **Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on**. [S.l.], 1994. p. 593–600.

SIEGWART, R.; NOURBAKHSH, I. R.; SCARAMUZZA, D. **Introduction to autonomous mobile robots**. [S.l.]: MIT press, 2011.

SMITH, R.; SELF, M.; CHEESEMAN, P. Estimating uncertain spatial relationships in robotics. In: **Autonomous robot vehicles**. [S.l.]: Springer, 1990. p. 167–193.

SORKINE, O. Least-squares rigid motion using svd. **Technical notes**, v. 120, p. 3, 2009.

STURM, J.; ENGELHARD, N.; ENDRES, F.; BURGARD, W.; CREMERS, D. A benchmark for the evaluation of rgb-d slam systems. In: IEEE. **Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on**. [S.l.], 2012. p. 573–580.

WHELAN, T.; JOHANNSSON, H.; KAESS, M.; LEONARD, J. J.; MCDONALD, J. Robust real-time visual odometry for dense rgb-d mapping. In: IEEE. **Robotics and Automation (ICRA), 2013 IEEE International Conference on**. [S.l.], 2013. p. 5724–5731.

YOUSIF, K.; BAB-HADIASHAR, A.; HOSEINNEZHAD, R. An overview to visual odometry and visual slam: Applications to mobile robotics. **Intelligent Industrial Systems**, Springer, v. 1, n. 4, p. 289–311, 2015.