UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE ENGENHARIA DE COMPUTAÇÃO

MARIANE TEIXEIRA GIAMBASTIANI

Sistema de Localização e Mapeamento Simultâneos Embarcado para Robôs Autônomos

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Engenharia da Computação

Orientador: Prof. Dr. Edison Pignaton de Freitas

Porto Alegre
2019

“...mais inteligente é aquele que sabe que não sabe...”

— SOCRATES

AGRADECIMENTOS

Agradeço a minha família por todo apoio e paciência ao longo desses anos de graduação, a minha mãe e meu pai por sempre me incentivar desde a infância a estudar e por me apoiarem na busca pelos meus sonhos, ao meu irmão pela paciência com a minha ausência. Agradeço ao meu namorado Renan por me apoiar, incentivar e me ensinar muito ao longo desse período. Agradeço também ao Professor Edson Prestes e a Professora Mariana Kolberg pelo ensino e suporte dado durante a minha iniciação científica no grupo de pesquisa de ambos e aos meus colegas de laboratório do Phi Group. Agradeço a minha amiga Ana Clara Mativi que conheci na graduação pela parceria durante esses anos.

RESUMO

Robôs verdadeiramente autônomos necessitam realizar o mapeamento do ambiente ao seu redor e estimar a sua localização no mapa. Por isso, Localização e Mapeamento Simultâneos (SLAM) é um problema fundamental da robótica móvel. Uma importante técnica de SLAM proposta na literatura é o RatSLAM: que modela computacionalmente uma rede de atratores competitivos baseado no funcionamento de hipocampos de roedores. O RatSlam é capaz de realizar a localização e o mapeamento em tempo real usando uma única câmera. Esse trabalho de graduação tem como objetivo, o desenvolvimento do sistema RatSLAM embarcado numa placa Jetson TX2 da NVIDIA, a fim de futuramente acoplar a placa em um robô.

Palavras-chave: Embedded SLAM. RatSLAM. SLAM. SLAM CUDA.

Embedded System of Simultaneous Localization and Mapping for Autonomos Robot

ABSTRACT

Truly autonomous robots need to map the environment that surrounds them while locating themselves in this environment. Therefore, Simultaneous Localization and Mapping (SLAM) is a fundamental problem of mobile robotics. An important SLAM technique proposed in the literature is RatSLAM. This proposal computationally models a network of competitive attractors based on the functioning of rodent hippocampus. RatSlam is capable of locating and mapping in real time using a single camera. This work aims to develop an implementation of the RatSLAM system embedded in an NVIDIA Jetson TX2 development kit, in order to couple this board in a robot in the future.

Keywords: Embedded SLAM, RatSLAM, SLAM, SLAM CUDA.

LISTA DE ABREVIATURAS E SIGLAS

SLAM	Simultaneous Localization and Mapping
CAN	Continuous attractor network
GPU	Graphics Processing Unit
CPU	Central Processing Unit
PC	Personal Computer
SRC	Source
ROS	Robot Operating System
CUDA	Compute Unified Device Architecture

LISTA DE FIGURAS

LISTA DE TABELAS

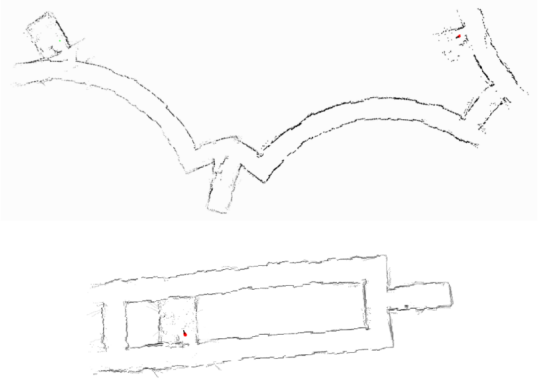
SUMÁRIO

1 INTRODUÇÃO

Nos últimos anos estamos presenciando o avanço científico e tecnológico de robôs, que estão cada vez mais presentes em aplicações científicas e comerciais. Contudo, o controle desses robôs ainda é praticamente manual, embora muitas das situações que necessitam da utilização de robôs são atividades de risco que podem prejudicar a curto e a longo prazo a saúde de quem os controla. Um exemplo é o uso de robôs aéreos para pulverização de agrotóxicos em lavouras, onde o operador do robô tem que ficar próximo da lavoura para poder guiá-lo, deixando o piloto perigosamente próximo da exposição a produtos químicos. Esse tipo de situação de risco pode ser evitada se o controle do robô aéreo for autônomo. Neste caso, a autonomia do robô faria com que ele sozinho pulverizasse uma região da lavoura sem ser guiado por um piloto.

O desenvolvimento de sistemas autônomos vem sendo estudado há décadas na área de robótica móvel. Vários trabalhos nessa área tratam de problemas de localização, mapeamento e navegação autônoma de robôs. Um dos problemas fundamentais da área é o chamado problema de *SLAM* (*Simultâneous Localization and Mapping*), que consiste em estimar simultaneamente o mapeamento de um ambiente previamente desconhecido e a localização precisa do robô neste mapa que está sendo estimado. Uma enorme variedade de técnicas de *SLAM* tem sido apresentada desde o início da década de 90, mas o grande foco nos últimos anos são as técnicas de *SLAM* visual, ou seja, técnicas que utilizam apenas câmeras como fonte de informação (ao invés, por exemplo, de sensores de alcance como sonares ou laser). Na Figura 1.1 temos um exemplo de um mapa, baseado em grade de ocupação, gerado com e sem o uso de um método de *SLAM*.

Figura 1.1: Comparação entre mapa construído sem o uso de *SLAM* e o mesmo mapa construído utilizando um método de *SLAM*.

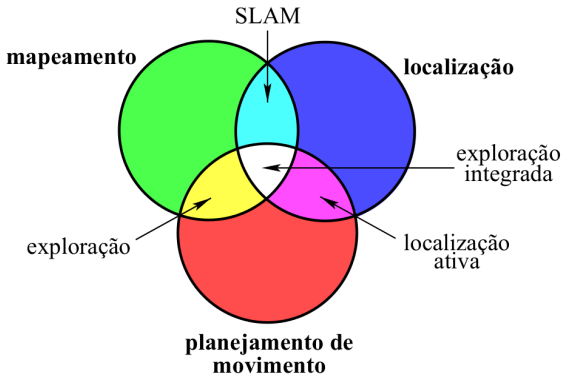


Este trabalho tem como objetivo a implementação de uma técnica de Localização e Mapeamento Simultâneos usando uma única câmera chamada *RatSLAM*. A técnica será embarcada em uma placa de pequeno porte, que futuramente poderá ser acoplada a um robô. O *RatSLAM* pode ser embarcado sem alterar o código original, contudo o desempenho do algoritmo depende da quantidade de dados que será processado e armazenado em memória, por exemplo, se esse algoritmo for compartilhado entre vários robôs, a tendencia é que o desempenho seja prejudicado. Por isso, nesse trabalho será apresentado uma alternativa de paralelização do *RatSLAM*. Um dos motivos para a escolha da técnica *RatSLAM* é que ela apresenta particularidades na estratégia de localização (reconhecimento de lugares usando um rede neural que pode ser compartilhada) a qual acreditamos ter um grande potencial para futuras aplicações com múltiplos robôs. Portanto, esse trabalho visa paralelizar o método *RatSLAM* a fim de que futuramente ele possa ser aplicado a múltiplos robôs com desempenho semelhante a sua execução em um único robô.

2 LOCALIZAÇÃO E MAPEAMENTO SIMULTÂNEO

A robótica móvel é a área que desenvolve métodos com a finalidade de tornar robôs verdadeiramente autônomos. Os três principais problemas estudados nessa área são: mapeamento, localização e planejamento de movimento. Todavia, a solução desses métodos separadamente não é muito usado, em geral são mais utilizadas a combinação desses problemas (MAKARENKO et al., 2002), como se pode observar na Figura 2.1.

Figura 2.1: Combinação dos problemas da robótica móvel.



Fonte: Figura adaptada de (MAKARENKO et al., 2002)

Nesse capítulo serão apresentados os problemas de localização, mapeamento e em seguida, a definição do problema de *SLAM* (*Simultaneous Localization and Mapping*) e a sua importância para a navegação de robôs autônomos.

2.1 Definição de *SLAM* e outros problemas básicos da robótica móvel

Conforme o próprio nome informa, o problema de *SLAM* é a combinação dos problemas de mapeamento e localização.

No problema de **mapeamento**, um robô autônomo precisa construir uma representação precisa dos objetos ou obstáculos presentes no ambiente que o cerca. O objetivo é permitir com que o robô seja capaz de navegar no local desviando de paredes e obstáculos. Para construir um mapa é necessário saber a localização global do robô, ou ao menos ter uma boa estimativa dessa localização, assim é possível gerar um mapa de obstáculos a partir de dados coletados por sensores (e.g. sonares e câmeras).

No problema de **localização**, o robô deve estimar sua posição em relação ao mapa do ambiente. Caso ela seja desconhecida a priori, o problema é chamado de localização

global. Caso a posição inicial seja conhecida, o problema é chamado de localização local ou *position tracking*. Neste caso, o robô deve ser capaz de corrigir erros de posicionamento que se acumulam devido ao ruído nas observações dos sensores. Resolver o problema de localização é muito importante para a navegação de um robô em um ambiente em que se conhece o mapa, pois sem isso o robô não será capaz de navegar eficientemente pelo ambiente. Quando o mapa da região onde se encontra o robô é conhecido, é possível estimar a sua localização usando dados coletados por sensores acoplados no robô.

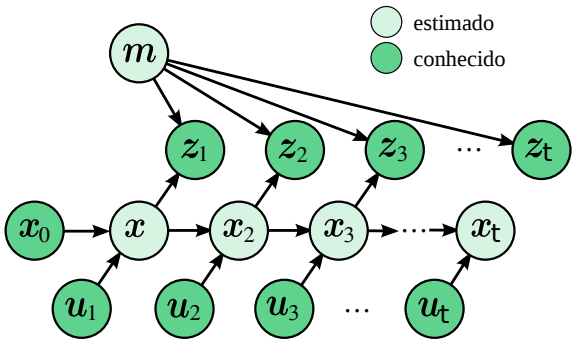
Na prática, esses dois problemas geralmente não podem ser resolvidos separadamente. Logo, um dos problemas fundamentais da robótica móvel é, construir um mapa sem ter a localização precisa do robô e estimar a localização do robô sem ter um mapa preciso. Esse problema, chamado de **Localização e Mapeamento Simultâneos** (comumente abreviado de **SLAM**), é uma tarefa bastante complexa dado que ele soluciona dois problemas interdependentes, como mostra a Figura 2.1.

A Figura 2.2 apresenta a exemplificação do problema. O modelo mostrado na Figura 2.2 apresenta os seguintes estados:

- u_t: representa a ação de navegação tomada pelo robô no instante t. Pode ser descrita por medidas de velocidade linear e angular, ou por medidas de odometria, etc.
- z_t: representa as observações feitas pelo robô no instante t. Em um ambiente descrito por *features*, pode ser as posições relativas de cada *feature* em relação ao robô. Considerando um robô equipado com sensores de alcance (e.g. sonar/laser), pode ser as medidas de distância lidas naquele instante.
- x_t: é o estado do robô (posição e orientação) no instante t. Com exceção da postura inicial x_0, que geralmente se assume como sendo o local (0,0,0) do mapa, deve ser estimado em função das ações $u_{1:t}$ e observações $z_{1:t}$ feitas pelo robô.
- m: é o mapa descrevendo os objetos no ambiente. É construído a partir da localização estimada do robô $x_{0:t}$ e das observações $z_{1:t}$ feitas por ele.

O problema supoem que um robô se desloca num ambiente onde não existe um mapa a priori e a localização dele não é conhecida, contudo, se sabe a posição de marcadores presente no ambiente e também o conjunto de ações de controle de movimento que será executado pelo robô. Assumindo uma posição inicial de partida, é possível estimar iterativamente a localização, a partir dos dados conhecidos, por seguinte se constrói um mapa com base na estimativa da localização.

Figura 2.2: Modelo do problema de *SLAM*.



2.2 Tipos de *SLAM*

A construção de um mapa com *SLAM* pode ser feito de diferentes formas: baseado em grades de ocupação, características (*features*) ou em geometria (STACHNISS, 2009). Os mapas baseados em grades de ocupação discretizam o ambiente em células, sobre as quais é atualizada a probabilidade de estarem livres ou ocupadas. Já os mapas baseados em *features* usam algoritmos de extração de características para detectar regiões conhecidas e usa-las como *landmarks*. Muitos desses mapas são armazenados na forma de nuvem de pontos. Mapas geométricos representam os obstáculos como polígonos, ocupam menos memória do que os mapas baseado em grade, mas são mais difíceis de construir, o que os deixa restritos a tipos específicos de ambientes.

A escolha do tipo de mapa está diretamente associada ao tipo de sensor que o robô dispõe. Geralmente, robôs equipados com sensores de alcance, como laser e sonares, são muito apropriados para a construção de mapas baseados em grades de ocupação. Por outro lado, robôs equipados apenas com câmera são mais apropriados para mapas baseados em *features*.

Existem inúmeras técnicas que resolvem o problema de *SLAM*, contudo elas geralmente podem ser agrupadas em três tipos: soluções baseadas em Filtro de Kalman, soluções baseadas em Filtro de Partículas Rao-Blackwellized (RBPF) e soluções baseadas na otimização de grafos de posição.

Abordagens usando Filtro de Kalman e suas variações, como os Filtros de Kalman Estendido (SMITH; SELF; CHEESEMAN, 1990) e Unscented (JULIER; UHLMANN, 1997), foram as abordagens pioneiras de *SLAM*, dominando a literatura durante a década de 90. Ainda são muito usadas para fusão sensorial em sistemas equipados com múltiplos sensores. Abordagens usando filtro de partículas (MURPHY, 1999) se torna-

ram muito populares no início da década passada após a proposta do método *FastSLAM* (MONTEMERLO et al., 2002) e, especialmente, após sua extensão para uso com grades de ocupação (HAHNEL et al., 2003). Enquanto abordagens baseadas em filtragem de Kalman usam mapas apenas baseados em *features*, abordagens de filtro de partículas permitem tratar diferentes tipos de mapas.

Mais recentemente, a abordagem que passou a dominar a literatura foi a baseada na otimização de grafos (THRUN; MONTEMERLO, 2006; GRISETTI et al., 2010). Este tipo de abordagem constrói um grafo com a sequência de poses ocupadas pelo robô durante sua trajetória e corrige as posições conforme vai reconhecendo lugares por onde passou. Com os avanços nas técnicas de otimização, esse tipo de abordagem mostrou melhoras expressivas nos resultados em mapas de larga escala, que era um dos principais pontos fracos das duas abordagens anteriores. De fato, a grande maioria dos métodos atuais de *SLAM* se baseiam na otimização de grafos, e dentre eles o método *RatSLAM*.

3 *RATSLAM*

Nesse capítulo será apresentado o método *RatSLAM* que se trata de um *SLAM* biologicamente inspirado em hipocampos de roedores. Os roedores são capazes de memorizar a localização de objetos de referência e a partir disso memorizar um mapa virtual onde é possível reconhecer regiões revisitadas (MILFORD; WYETH, 2008).

O método *RatSLAM* simula o hipocampo de memorização de um roedor utilizando uma rede neural de atratividade contínua (*CAN: Continuous attractor network*) para estimar a localização de um robô (MILFORD; WYETH; PRASSER, 2004). Combinando essa estimativa de posição juntamente da odometria é possível construir um mapa da região por onde o robô se desloca.

Assim como os roedores são capazes de memorizar regiões conhecidas de um ambiente, o *RatSLAM* usa imagens capturadas por uma única câmera para reconhecer locais já visitados e corrigir a construção do mapa fazendo fechamento de *loop*.

3.1 Visão Geral

A arquitetura do *RatSLAM* é apresentado na Figura 3.1. O *RatSLAM* é dividido em três grandes blocos *Pose Cells*, *Local View Cells* e *Experience Map*.

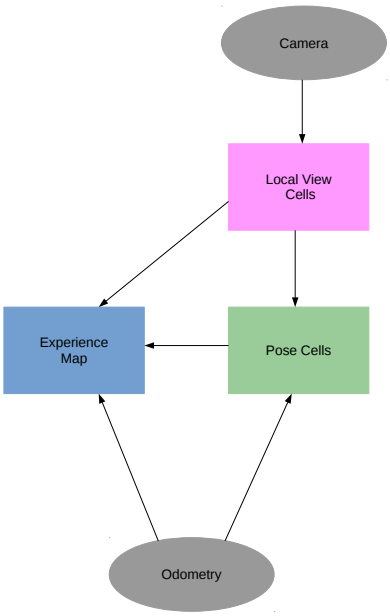
O bloco *Pose Cells* faz a estimativa da localização do robô atualizando em tempo real uma rede neural CAN a partir da odometria e do bloco *Local View Cells*. O bloco *Local View Cells* monta *templates* de imagens observadas pela câmera para serem utilizadas pelos blocos *Pose Cells* e *Experience Map*. O bloco *Experience Map* constrói o mapa do ambiente por onde o robô caminha usando as informações fornecidas pelo bloco *Local View Cells*, *Pose Cells* e também pela odometria.

A odometria usada pode ser tanto visual, extraída de imagens, quanto fornecida por um odômetro. Os blocos serão descritos detalhadamente nas próximas seções.

3.2 *Local View Cells*

Esse módulo do *RatSLAM* é responsavel por coletar as imagens observadas e salvá-las em *templates*. Toda vez que uma nova cena é coletada é feita uma análise de semelhança com todos os *templates* salvos, se essa cena for similar a um já existente a *Lo-*

Figura 3.1: Arquitetura do *RatSLAM*.



Fonte: Figura ataptada de (MILFORD; WYETH, 2008)

cal View Cell é ativada, caso contrário um novo template é gerado e adicionado a coleção, como mostra a Figura 3.2.

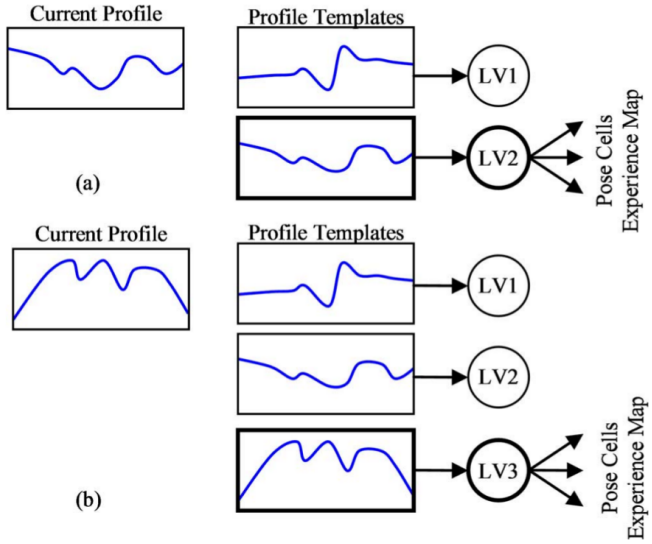
Quando uma célula do bloco *Local View Cells* é ativada essa informação é enviada para os módulos seguintes *Pose Cells* e *Experience Map*.

O casamento de imagens é medido a partir de uma curva de intensidade da cena. Essa curva é gerada da soma das intensidades luminosas de cada coluna da imagem em *grayscale*. A semelhança da curva de uma imagem atual é comparada com as curvas dos *templates* anteriores, se as curvas forem suficientemente semelhantes é feito o casamento, como também pode ser observado na Figura 3.2.

3.3 *Pose Cells*

A rede neural CAN se trata de uma matriz P de três dimensões (x, y, θ) como mostrado na Figura 3.3. As coordenadas x e y reproduzem o ambiente de deslocamento de um roedor, já o ângulo θ simula a rotação da cabeça do roedor, ou nesse caso a rotação da câmera de um robô terrestre.

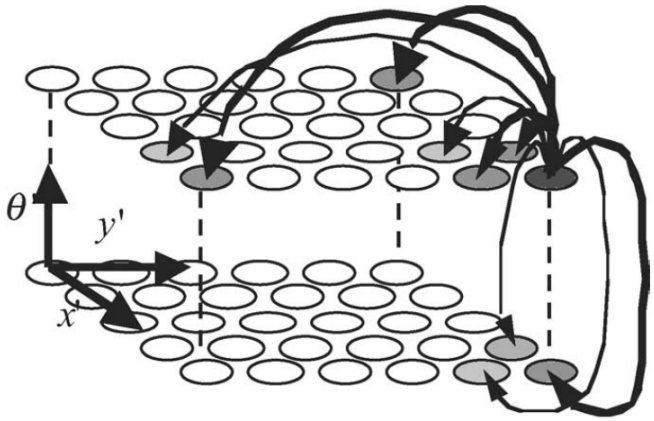
Figura 3.2: Calculando casamento de cenas. (a) Quando a imagem observada é correspondente à um *template* existente um *Local View* é ativado. (b) Contudo, se a cena observada não casa com nenhum *template* existente um novo é adicionado.



Fonte: Figura extraída de (MILFORD; WYETH, 2008)

3.3.1 Dinâmica de Atratores Contínuos (CAN)

Figura 3.3: Matriz de *Pose Cells*.



Fonte: Figura extraída de (MILFORD; WYETH, 2008)

A matriz de *Pose cells* P tem seus pesos de ativação atualizados de acordo com uma distribuição Gaussiana tridimensional representada pela matriz $\varepsilon_{a,b,c}$ e os índices a, b e c são as distâncias entre cada índice (i,j,k) e a célula excitada (x', y', θ'). Os pesos da distribuição Gaussiana são calculados pela fórmula 3.1 onde k_p e k_d são respectivamente

constantes de pose e direção.

$$\varepsilon_{a,b,c} = e^{\frac{-(a^2+b^2)}{kp}} e^{\frac{-(c^2)}{k_d}} \tag{3.1}$$

Quando pesos das Pose Cells são atualizados devido a uma excitação usa-se a fórmula 3.2 onde os índices n representam os tamanhos de cada dimensão da matriz P.

$$\Delta P_{x',y',\theta'} = \sum_{i=0}^{(n_{x'}-1)} \sum_{j=0}^{(n_{y'}-1)} \sum_{k=0}^{(n_{\theta'}-1)} P_{i,j,k}\ \varepsilon_{a,b,c} \tag{3.2}$$

Para calcular os índices a, b e c usados na fórmula 3.1 usa-se as equações 3.3 isso implica que as conexões excitatórias sejam circulares, ou seja, se conectem em faces opostas da matriz tridimensional como mostra a Figura 3.3.

$$\begin{aligned} a &= (x' - i)(\text{mod } n_{x'}) \\ b &= (y' - j)(\text{mod } n_{y'}) \\ c &= (\theta' - k)(\text{mod } n_{\theta'}) \end{aligned} \tag{3.3}$$

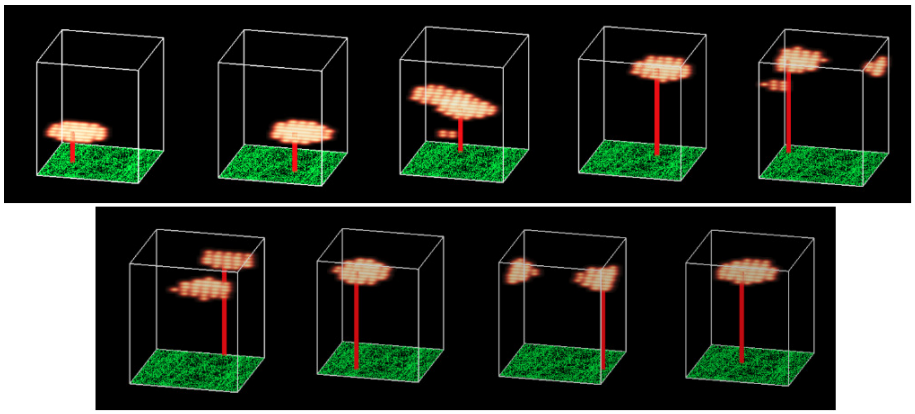
Assim como existe a excitação das células também existe a inibição das *Pose Cells*. A inibição ocorre logo após um excitação e é feita da mesmo forma que a excitação, porém usando pesos negativos, esse processo é feito de maneira sequencial e não simultâneo. Essa inibição é feita no conjunto de células ativas, portanto também é realizado uma inibição global que abrange todas as células da matriz P, ambas são dadas pela equação 3.4, onde $\psi_{a,b,c}$ representa os pesos negativos (que são calculados da mesma forma que o $\varepsilon_{a,b,c}$ usando a fórmula 3.2) e φ a inibição global. Mesmo usando pesos negativos todos as células de P são limitadas a valores não negativos e têm seus valores normalizados.

$$\Delta P_{x',y',\theta'} = \sum_{i=0}^{(n_{x'}-1)} \sum_{j=0}^{(n_{y'}-1)} \sum_{k=0}^{(n_{\theta'}-1)} P_{i,j,k}\ \psi_{a,b,c} - \varphi \tag{3.4}$$

Conforme descrito no início dessa seção o módulo *Pose Cells* recebe como entrada os *templates* da *Local View Cells* e a odometria. Caso esse bloco não receba esses dados de

entrada a matriz P irá convergir para um elipsoide onde seu volume representa a atividade da rede neural. A medida que a *Pose Cells* recebe as informações de entrada o elipsoide, que se trata de um pacote de células ativas, vai se mover pela matriz P, conforme mostra a Figura 3.4.

Figura 3.4: Movimentação do elipisoide na matriz *Pose Cells*.



3.3.2 Integração de Caminho

Enquanto o robô se desloca o pacote de células ativas também tem que se movimentar. A partir dos dados de odometria, seja visual ou por um sensor próprio, o módulo *Pose Cells* usa as informações de translação e rotação recebidas para deslocar o elipsoide pela matriz P. A matriz é tridimensional representando translação nos eixos x e y e rotação no eixo θ, ou seja, a terceira dimensão da matriz representa o ângulo de rotação. O cálculo do deslocamento é dado pela fórmula 3.5, onde k é uma constante e v e ω são respectivamente as velocidades linear e angular do robô.

$$\Delta x = k_x v \cos(\theta), \quad \Delta y = k_y v \sin(\theta), \quad \Delta\theta = k_\theta \omega \tag{3.5}$$

3.3.3 Calibração da *Local View*

O módulo *Local View* armazena um vetor V e cada elemento desse vetor está associado a uma célula ativa da matriz P de *Pose Cells*. Uma célula do vetor V só é ativa quando a cena atual coletada pela câmera é semelhante as armazenadas anteriormente, como foi discutido na seção 3.2.

O aprendizado da matriz CAN é dado pela ligação entre o vetor V, matriz P e o peso do aprendizado que é salvo em uma matriz β.

Após calibrar os pesos de aprendizado essa informação tem que ser adicionada na rede neural *Pose Cells* isso é feito usando a fórmula 3.6. O δ é uma constante que representa a força da calibração visual e o n_{act} é o número de células ativas.

$$\Delta P_{x',y',\theta'} = \frac{\delta}{n_{act}} \sum_{i} \beta_{i,x',y',\theta'} V_i \tag{3.6}$$

O processo de aprendizagem também auxilia na correção do erro acumulado durante o procedimento de integração de caminho (MILFORD; WYETH, 2008).

3.4 *Experience Map*

O mapa de experiência é um mapa topológico composto por várias experiências individuais e_i, e as experiências são conectadas por transações t. Cada experiencia e_i é definida como uma associação entre um código de *Pose Cells* P^i e um código de *Local View* V^i, onde esse código se refere a um comportamento padrão em um conjunto de células. Uma experiencia é definida como uma 3-tupla 3.7 e cada e_i é posicionada na posição $\mathbf{p}_i$ no espaço de experiências.

$$e_i = \{P^i, V^i, \mathbf{p}^i\} \tag{3.7}$$

3.4.1 *Experience Creation*

A primeira experiência é criada em um posição inicial arbitrária, posteriormente as experiências são geradas a partir da primeira. Quando os códigos P^i ou V^i são diferentes dos códigos das experiências armazenadas uma nova e_i é criada. Para medir a diferença entre os códigos de pose e *Local View* atuais com os já existentes usa-se a fórmula 3.8 para calcular um *score*, onde μ_p e μ_v são usados para ponderar a pontuação S.

$$S = \mu_p \left|P^i - P\right| \; + \; \mu_v \left|V^i - V\right| \tag{3.8}$$

Quando a pontuação S ultrapassar um limiar S_{max} para todas as experiências exis-

tentes, então uma nova experiência é criada e uma transação t é associada a ela. A transação t_{ij} 3.9 salva a mudança de posição medida através da odometria.

$$t_{ij} = \{\Delta \mathbf{p}^{ij}\} \tag{3.9}$$

O deslocamento de posição Δp^{ij} representa a movimentação do robô de acordo com a odometria. O t_{ij} é a ligação entre a experiência prévia e_i e a nova experiência e_j, portanto e_j é criado através da equação 3.10. Contudo, é provável que p^j mude quando ocorrer um fechamento de ciclo.

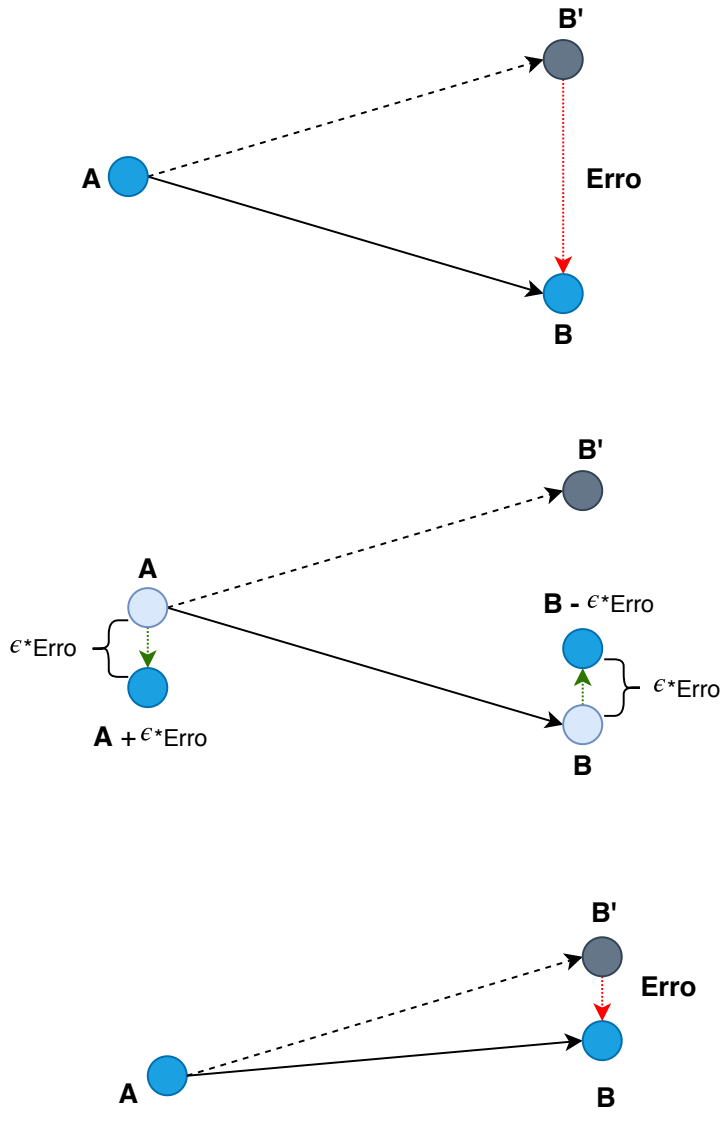
$$e_j = \{P^j\ , V^j\ , \mathbf{p}^i\ + \Delta \mathbf{p}^{ij}\} \tag{3.10}$$

3.4.2 Fechamento de Ciclo

Embora o *RatSLAM* não faça detecção de ciclos explicitamente, o fechamento de ciclo ocorre quando o código de *Pose Cells* P^i e o código de *Local View* V^i, após uma alteração na experiência, correspondem a alguma experiência já armazenada. Contudo, quando isso ocorre é improvável que o deslocamento somado nas transições acarretem no fechamento de um ciclo na mesma posição. Por isso, para realizar um fechamento de ciclo se faz o relaxamentos sucessivos do erro das arestas (conforme mostra a Figura 3.5) usando a fórmula 3.11, onde α é uma taxa constante de correção de valor 0.5, N_f é o número de ligações que saem da experiência e_i e se conectam em outras experiências, e N_t é o número de conexões que chegam na experiência e_i.

$$\Delta \mathbf{p}^i = \alpha \left(\sum_{j=1}^{N_f} (\mathbf{p}^j - \mathbf{p}^i - \Delta \mathbf{p}^{ij}) + \sum_{k=1}^{N_t} (\mathbf{p}^k - \mathbf{p}^i - \Delta \mathbf{p}^{ki}) \right) \tag{3.11}$$

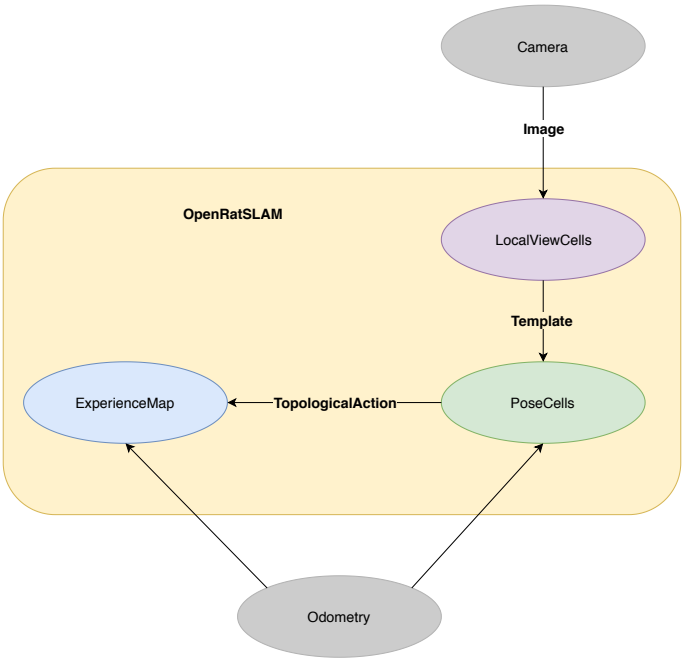
Figura 3.5: Correção do grafo a partir da observação B'.



4 ANÁLISE DO CÓDIGO

Neste capítulo será apresentado a análise do código do *RatSLAM*. A implementação analisada nesse capítulo é a versão *Open Source Open RatSLAM* (BALL, 2018). O *Open RatSLAM* usa o sistema ROS *(Robot Operating System)* que se trata de pseudo sistema operacional que disponibiliza bibliotecas, controle de sensores e permite comunicação entre processos. O *RatSLAM* usa o ROS para realizar troca de mensagens entre os processos que o compõem como apresentam as Figuras 4.1 e 4.2.

Figura 4.1: Módulos do *Open RatSLAM*

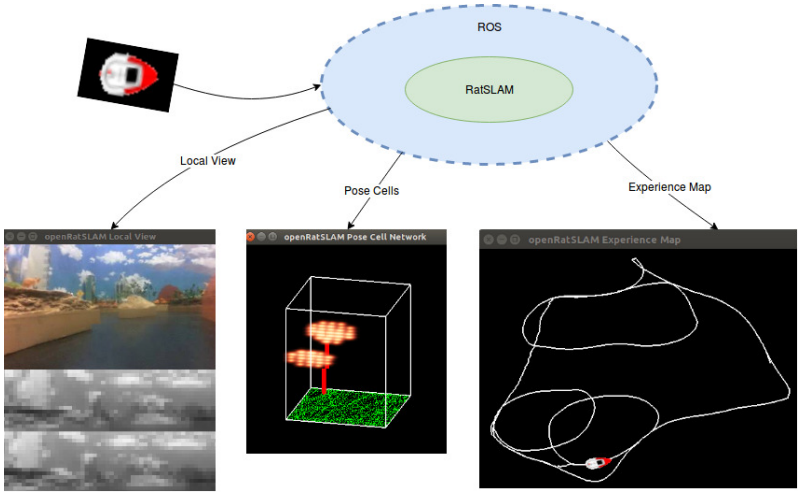


A análise do código primeiramente se deu através da medição do tempo médio de execução de cada um dos métodos das classes que compõem o *RatSLAM*. Cada processo principal representa um módulo (descritos nos arquivos *main_pc*, *main_lv* e *main_em*) e para cada módulo existe uma classe que contém todos os métodos referentes a *Pose Cells*, *Local View* e *Experience Map* respectivamente. O três processos principais são executados simultaneamente.

Nas seções 4.1, 4.2 e 4.3 será apresentado os testes medidos de cada módulo em uma tabela que segue a ordem de hierarquia de chamada das funções correspondentes a cada classe.

Os testes foram realizados em dois ambientes um computador pessoal (PC) e na placa Jetson TX2, cujas configurações desses ambientes estão descritas na Seção 6.1. Os

Figura 4.2: *Open RatSLAM* com o ROS



dados coletados nos testes foram o tempo médio de execução e o número de iterações executadas de cada método, com esses dois valores foi calculado o tempo total estimado de execução de cada função.

Para cada ambiente de execução foi construído uma tabela com as informações coletadas. Essas tabelas foram utilizadas para escolher quais métodos tinham a maior necessidade de serem paralelizados a fim de melhorar o desempenho do algoritmo *RatSLAM*. Os números maiores de cada coluna foram destacados em negrito e os nomes das funções cujos nomes estão colorizados representam os métodos que têm um custo alto de desempenho e se fossem modificadas para funcionarem em paralelo poderiam melhorar o desempenho geral do *RatSLAM*. Os nomes em azul mostram as funções que são paralelizáveis e os nomes em vermelho as funções com desempenho afetado por estas (isto é, as funções que as invocam).

4.1 *Local View Cells*

As Tabelas 4.1 e 4.2 mostram um teste do módulo *Local View* executado no PC e na placa Jetson. Como podemos observar a função *compare* é executada muitas vezes e é a função mais lenta do módulo *Local View*, afetando o desempenho das funções que a utilizam, por exemplo as funções *image_callback* e *lv->on_image*.

Tabela 4.1: Teste ***Local View*** usando a base de dados irataus no **PC**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_lv.cpp	–	–	–
+ image_callback	**18.0552**	16657	**300745.47**
+ lv->on_image	**17.555**	16657	**292413.64**
+ convert_view_to_template	1.1403	16657	18993.98
+ compare	**16.3046**	16656	**271569.42**
+ create_template	0.0373	2064	76.99

Tabela 4.2: Teste ***Local View*** usando a base de dados irataus na placa **Jetson**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_lv.cpp	–	–	–
+ image_callback	**24.2336**	16655	**403610.608**
+ lv->on_image	**24.1633**	16655	**402439.7615**
+ convert_view_to_template	1.4136	16655	23543.508
+ compare	**22.588**	16654	**376180.552**
+ create_template	0.0426	2064	87.9264

4.2 *Pose Cells Network*

As Tabelas 4.3 e 4.4 apresentam um teste do módulo *Pose Cells Network*. O método *excite_helper* faz uma convolução para excitar as células da matriz *Pose Cells Network*, enquanto que o *inhibit_helper* faz uma convolução para inibir a energia das células da *Pose Cells Network*. Podemos observar nessas tabelas que as funções *excite* e *inhibit* são as mais custosas do módulo *Pose Cells* e afetam no desempenho dos métodos *odo_callback* e *pc->on_odo*.

Tabela 4.3: Teste ***Pose Cells*** usando a base de dados irataus no **PC**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_pc.cpp	–	–	–
+ PosecellNetwork	1.344	1	1.34
+ pose_cell_builder	0.486	1	0.49
+ generate_wrap	0.0008	6	0.0
+ odo_callback	**28.4156**	16658	**473347.06**
+ pc->on_odo	**27.4588**	16657	**457381.23**
+ excite	5.6037	16657	93340.83
+ excite_helper	0.0085	**3980736**	33836.26
+ inhibit	**20.5566**	16657	**342411.29**
+ inhibit_helper	0.0025	**27281272**	68203.18
+ global_inhibit	0.0142	16657	236.53
+ normalise	0.0281	16657	468.06
+ path_integration	1.1622	16657	19358.77
+ rot90_squar	0.005	**1499130**	7495.65
+ circshift2d	0.0003	599652	179.90
+ find_best	0.0259	16657	431.42
+ pc->get_action	0.0178	16440	292.63
+ create_experience	0.0051	2826	14.41
+ get_delta_pc	0.0001	19764	1.98
+ template_callback	11.8819	16657	197916.81
+ pc->on_view_template	0.0207	16657	344.80
+ create_view_template	0.0062	2064	12.80
+ inject	0.0002	2776	0.56

4.3 *Experience Map*

As Tabelas 4.5 e 4.6 apresentam um teste do módulo *Experience Map*. Como podemos observar a função *iterate* possui um custo elevado de execução que interfere no método que a utiliza chamado *action_callback*.

Tabela 4.4: Teste ***Pose Cells*** usando a base de dados irataus na placa **Jetson**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_pc.cpp	–	–	–
+ PosecellNetwork	1.256	1	1.256
+ pose_cell_builder	0.425	1	0.425
+ generate_wrap	0.0007	6	0.0042
+ odo_callback	**52.4504**	16658	**873718.7632**
+ pc->on_odo	**52.3488**	16657	**871973.9616**
+ excite	8.1932	16657	136474.1324
+ excite_helper	0.0104	**3965439**	41240.5656
+ inhibit	**40.3076**	16657	**671403.6932**
+ inhibit_helper	0.0038	**27079179**	102900.8802
+ global_inhibit	0.0391	16657	651.2887
+ normalise	0.0619	16657	1031.0683
+ path_integration	3.5085	16657	58441.0845
+ rot90_squar	0.0154	**1499130**	23086.602
+ circshift2d	0.0009	599652	539.6868
+ find_best	0.0691	16657	1150.9987
+ pc->get_action	0.0451	16245	732.6495
+ create_experience	0.0157	2822	44.3054
+ get_delta_pc	0.0001	19472	1.9472
+ template_callback	0.0878	16655	1462.309
+ pc->on_view_template	0.0483	16655	804.4365
+ create_view_template	0.0148	2064	30.5472
+ inject	0.0002	2776	0.5552

Tabela 4.5: Teste ***Exp. Map*** usando a base de dados irataus no **PC**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_em.cpp	–	–	–
+ odo_callback	0.084	**16658**	1399.27
+ en->on_odo	0.0029	**16657**	48.31
+ action_callback	**4.8866**	4416	**21579.23**
+ em->on_create_experience	0.0516	2826	145.82
+ em->on_set_experience	0.0003	4415	1.32
+ em->on_create_link	0.0079	3234	25.55
+ em->iterate	**4.1802**	4416	**18459.76**

Tabela 4.6: Teste ***Exp. Map*** usando a base de dados irataus na placa **Jetson**.

	irataus		
Método	**Tempo (ms)**	**# Iterações**	**Total (ms)**
main_em.cpp	–	–	–
+ odo_callback	0.0737	**16658**	1227.6946
+ en->on_odo	0.0029	**16657**	48.3053
+ action_callback	**11.811**	4390	**51850.29**
+ em->on_create_experience	0.0851	2822	240.1522
+ em->on_set_experience	0.0005	4389	2.1945
+ em->on_create_link	0.0132	3224	42.5568
+ em->iterate	**11.3548**	4390	**49847.572**

5 PARALELIZAÇÃO DO *RATSLAM*

5.1 Paralelização na *Local View Cells*

Conforme mostrado na Seção 4.1 no módulo *Local View*, *compare* é uma função que poderia ser paralelizada pois seu desempenho impacta bastante no desempenho de todo o módulo. Esse módulo possui um vetor de *templates* de imagens reconhecidas pelo algoritmo. Toda vez que uma nova imagem é coletada, o algoritmo deve verificar se o *template* correspondente a ela é igual ou semelhante aos *templates* de imagens armazenados, para então decidir se deve adicionar ou não esse novo *template* ao vetor.

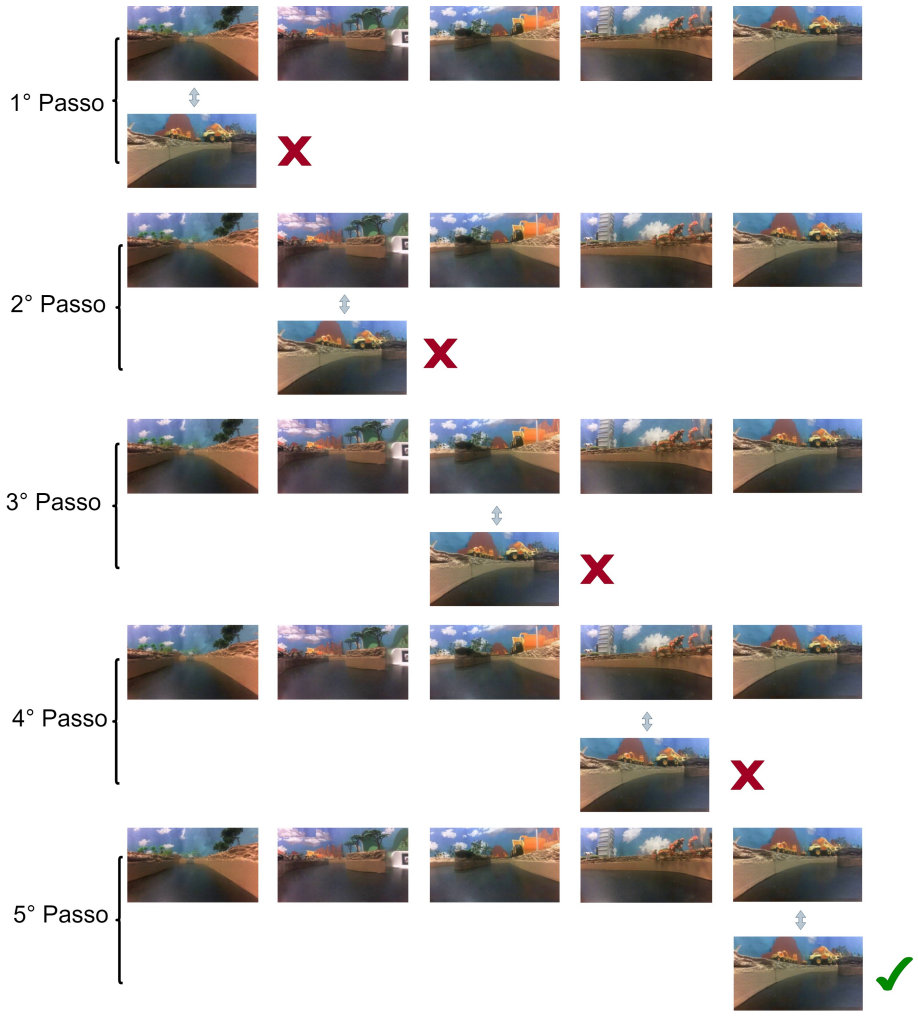
5.1.1 Método *compare*

O método *compare* compara o *template* do instante atual com os *templates* do passado, um a um, como pode ser visto no exemplo da Figura 5.1 e no Algoritmo 1. Além disso, cada comparação entre dois *templates* considera uma janela deslizante de *offsets*. A função calcula dois valores que são usados na decisão da criação de um novo *template*.

Algorithm 1 Função ***compare*** do *RatSLAM* original.

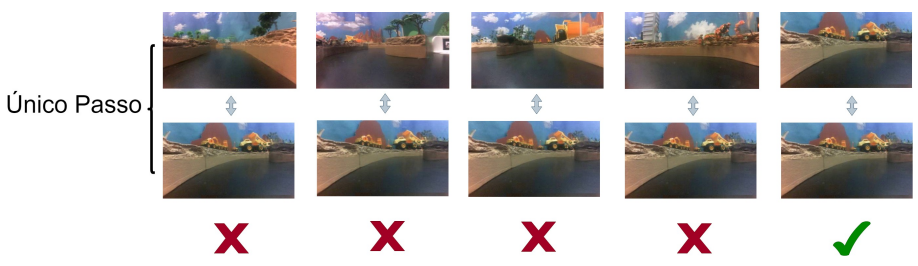
```
1:  procedure COMPARE(currentTemplate, templates)
2:      // Verifica cada template
3:      for all vt in templates do
4:          // Para cada template testado aplica diferentes offsets
5:          for o in 1..num_offsets do
6:              // Mede a semelhança do template atual com vt aplicado o offset
7:              diff ← calc_diff(currentTemplate,vt,o)
8:              // Encontra melhor resultado (menor diferença entre templates)
9:              if diff < min_diff then
10:                 min_diff ← diff
11:                 min_template ← vt.id
12:             end if
13:         end for
14:     end for
15:     vt_matchId ← min_template
16:     vt_error ← min_diff
17:     // Com as informações de vt_matchId e vt_error a função on_image de-
        cide se há um novo template diferente dos demais.
18: end procedure
```

Figura 5.1: Comparação sequencial



Da forma que é feito no *RatSLAM* a medida que o algoritmo executa, o número de *templates* aumenta, tornando a comparação mais custosa com o passar do tempo. Contudo, esse tipo de problema pode ser solucionado mudando a estratégia de comparação. Se a comparação for feita de maneira paralela ela terá um custo fixo por execução. Um exemplo de uma comparação paralela pode ser visto na Figura 5.2.

Figura 5.2: Comparação paralela



5.1.2 Modificações na função *compare*

A fim de melhorar o desempenho do algoritmo *RatSLAM* propomos fazer a comparação de imagens usando uma GPU. Portanto, a função *compare* do algoritmo foi reescrita na linguagem CUDA conforme mostra o Algoritmo 2, deixando a comparação com um custo computacional fixo no decorrer da execução do *RatSLAM*. Para isso funcionar é preciso que a informação dos *templates* já esteja copiada na memória da GPU (vetor `templates_cuda`). Essa é cópia é feita fora da função compare, durante a criação de cada *template*, tendo custo proporcional ao tamanho de um *template* (ou seja, constante)[1].

Algorithm 2 Função ***compare_cuda*** do *RatSLAM_CUDA*.

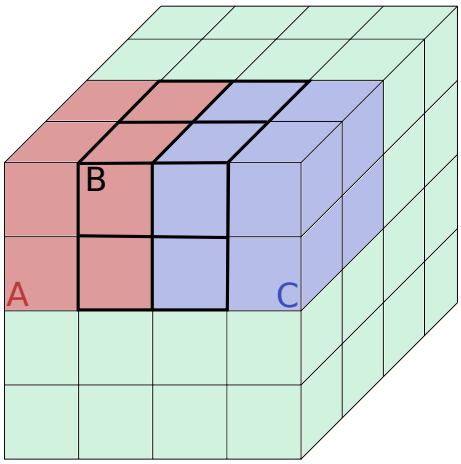
```
1:  procedure COMPARE_CUDA(currentTemplate, templates_cuda)
2:      // Copia o template atual para memória da GPU
3:      ct_cuda ← currentTemplate
4:      // Seta o tamanho da matriz de blocos na memória da GPU
5:      dim3 G(num_templates) // dimensão do grid de blocos
6:      dim3 B(num_offsets) // dimensão de bloco de threads

7:      // Mede em paralelo a semelhança do template atual com todos os templates
        considerando todos os offsets
8:      r[] ← calc_diff_cuda<<< G, B >>>(ct_cuda, templates_cuda)

9:      // Copia o vetor de resultados para memória da CPU
10:     result[] ← r[]
11:     // Encontra melhor resultado no vetor
12:     for tId in 1..num_templates do
13:         for o in 1..num_offsets do
14:             diff ← result[tId * num_offsets + o]
15:             if diff < min_diff_cuda then
16:                 min_diff_cuda ← diff
17:                 min_template_cuda ← tId
18:             end if
19:         end for
20:     end for
21:     vt_matchId ← min_template_cuda
22:     vt_error ← min_diff_cuda
23: end procedure
```

[1] Assim como é feito no código original do *RatSLAM*, o vetor de *templates* é inicialmente alocado com um tamanho bem grande para evitar custos de realocação toda vez que um novo *template* é criado.

5.2 Paralelização na *Pose Cells Network*

Conforme discutido na Seção 3.3 o módulo *Pose Cells* tem um conjunto de células ativas cujo formato é um elipsoide que se move por uma matriz tridimensional. Dois métodos são aplicados constantemente nessa matriz para excitar e inibir o conjunto de células ativas, são eles respectivamente *excite* e *inhibit*. Porém, esses métodos não são facilmente paralelizáveis, pois fazem convoluções na matriz onde muitas células são variáveis dependentes e caso fossem paralelizadas provocariam problemas de concorrência. Isso é exemplificado na Figura 5.3 onde a matriz B possui células em comum com as matrizes A e C. Entretanto, a aplicação de um filtro 3D local nas submatrizes, modificando uma variável por vez, pode ser feito em paralelo. A aplicação do filtro 3D é semelhante a aplicação de um filtro 2D e consiste na multiplicação de duas matrizes (a original e um kernel) célula a célula, como pode ser visto no exemplo da Figura 5.4, portanto é algo que pode ser feito em paralelo sem problemas de concorrência.

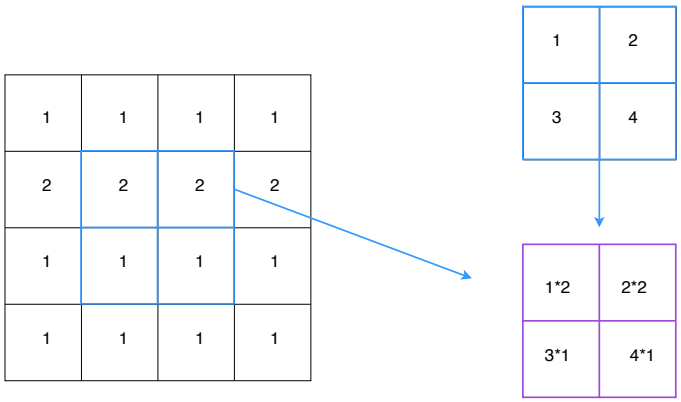
Figura 5.3: Sobreposição de células na atualização de matrizes adjacentes causa problemas de concorrência.



5.2.1 Métodos *excite_helper* e *inhibit_helper*

O método *excite_helper* aplica um *kernel* que excita as células da *Pose Cells Network*. A fim de tentar melhorar o desempenho do *RatSLAM* propomos aplicar esse filtro de forma paralela usando uma GPU, visto que esse método é executado inúmeras vezes, conforme mostra a Tabela 4.3. O mesmo trabalho de paralelização foi feito de

Figura 5.4: Aplicação de um filtro em uma matriz bidimensional.



forma análoga para a função *inhibit_helper*, que inibe as células da *Pose Cells Network*.

5.2.2 Modificações nas funções *excite_helper* e *inhibit_helper*

Algorithm 3 Função ***excite*** do *RatSLAM* original.

```
1:  procedure EXCITE(poseCells)
2:      Cria matriz auxiliar newPoseCells vazia
3:      for i in 1..dim_X do
4:          for j in 1..dim_Y do
5:              for k in 1..dim_θ do
6:                  if Célula é ativa then
7:                      // Chama a função excite_helper
8:                      excite_helper(i, j, k, poseCells, newPoseCells)
9:                  end if
10:             end for
11:         end for
12:     end for
13:     poseCells ← newPoseCells
14: end procedure
```

O Algoritmo 3 mostra que para cada célula ativa da matriz *Pose Cells* (i.e. uma célula com peso diferente de zero) é executada a chamada da função *excite_helper* que aplica um filtro gaussiano nas células ao redor da célula ativa, como pode ser visto no Algoritmo 4. O método *excite_helper* aplica um filtro cúbico de tamanho sete ao redor da célula ativa.

Para melhorar o desempenho da função *excite* uma abordagem seria executar a aplicação do método *excite_helper* em paralelo, a fim de diminuir o custo de execução

Algorithm 4 Função ***excite_helper*** do *RatSLAM* original.

```
1:  procedure EXCITE_HELPER(i, j, k, poseCells, newPoseCells)
2:      // Para todas as células em um cubo de lado dim_excite centrado em (i, j, k)
3:      int dim = dim_excite // igual a 7
4:      for x in 1..dim do
5:          for y in 1..dim do
6:              for z in 1..dim do
7:                  a, b, c = indices_correspondentes_na_matriz(i,j,k,x,y,z)
8:                  // Atualiza a energia da célula
9:                  newPoseCells[a,b,c] ← poseCells[a,b,c] * kernel[x,y,z]
10:             end for
11:         end for
12:     end for
13:     return newPoseCells
14: end procedure
```

Algorithm 5 Função ***excite_cuda*** do *RatSLAM_CUDA*.

```
1:  procedure EXCITE_CUDA(poseCells)
2:      Cria matriz auxiliar newPC_cuda vazia na GPU
3:      // Copia a matriz poseCells para memória da GPU
4:      pc_cuda ← poseCells
5:      for i in 1..dim_X do
6:          for j in 1..dim_Y do
7:              for k in 1..dim_θ do
8:                  if Célula é ativa then
9:                      // Chama a função excite_helper_cuda
10:                     excite_helper_cuda(i, j, k, pc_cuda, newPC_cuda)
11:                 end if
12:             end for
13:         end for
14:     end for
15:     // Copia a matriz resultante para memória da CPU
16:     poseCells ← newPC_cuda
17: end procedure
```

total da *excite*. Portanto, os Algoritmos 5 e 6 apresentam as modificações realizadas nas respectivas funções. A mesma abordagem foi feita para as funções *inhibit* e *inhibit_helper* a única diferença em relação as funções de *excite* é que os pesos de inibição são negativos e a dimensão do cubo de inibição é menor (no dataset testado, a inibição é feita em um cubo de tamanho 5 ao invés de 7), porem a implementação dos métodos são idênticas.

Algorithm 6 Função ***excite_helper_CUDA*** do *RatSLAM_CUDA*.

```
1: procedure EXCITE_HELPER_CUDA(i, j, k, pc_cuda, newPC_cuda)
2:     // Seta o tamanho da matriz de blocos na memória da GPU
3:     int dim = dim_excite
4:     dim3 G(1) // dimensão do grid de blocos
5:     dim3 B(dim, dim, dim) // dimensão de bloco de threads

6:     // Chama a função excite_cell_cuda que atualiza em paralelo a energia de cada
       célula do bloco sendo excitado
7:     excite_cell_cuda<<< G, B>>>(i, j, k, pc_cuda, newPC_cuda)
8: end procedure
```

5.3 Paralelização na *Experience Map*

O módulo *Experience Map* constrói um mapa de experiências baseado nas observações do robô, conforme mostrado na Seção 3.4. O mapa é representado por uma grafo onde cada nodo é uma experiência. O método *iterate* desse módulo faz o relaxamento do grafo que corrige parcialmente os erros de construção do mapa. A função *iterate* do *RatSLAM* não pode ser diretamente paralelizada, pois a correção é feita através da atualização de todos os pares de nodos associados a arestas do grafo. Portanto, se a comparação entre nodos que possuem vizinhos em comum fosse feita em paralelo várias comparações poderiam corrigir a posição de um mesmo nodo simultaneamente gerando um problema de concorrência de variáveis.

5.3.1 Método *iterate*

A implementação original da função *iterate* não é diretamente paralelizável, contudo uma modificação na implementação foi adicionada para que o método pudesse ser executado em paralelo. Como a correção de múltiplas arestas associadas a um mesmo nodo implica em múltiplas atualizações na posição deste nodo, é preciso executar a atualização destas arestas de forma sequencial, como mostra o Algoritmo 7. No entanto, é possível corrigir simultaneamente arestas que não possuem nodos em comum. Para tal é preciso descobrir quais são essas arestas, o que é conhecido como o problema de "coloração de arestas".

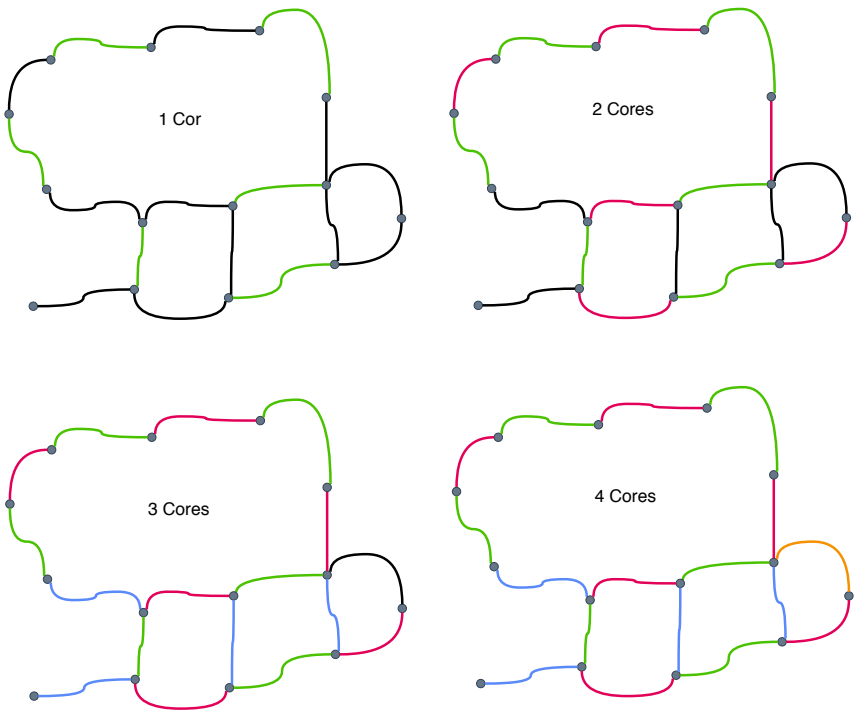
Algorithm 7 Função ***iterate*** do RatSLAM original.

```
1:  procedure ITERATE(experiences)
2:      int exp_loops = 10 //Número de iterações do relaxamento
3:      for i in 1...exp_loops do
4:          for e in experiences do
5:              for l in e→links_from do
6:                  Relaxa uma vez a aresta l
7:              end for
8:          end for
9:      end for
10: end procedure
```

5.3.2 Modificações na função *iterate*

Para desenvolver uma abordagem paralelizada do método *iterate*, primeiramente usamos um algoritmo trivial de coloração de arestas, mostrado em Algoritmo 8. Esse algoritmo atribui cores diferentes entre arestas vizinhas não permitindo que um nodo tenha duas ou mais arestas de mesma cor, conforme mostra o exemplo da Figura 5.5. A cada nova aresta criada o algoritmo verifica se há alguma cor não usada nas arestas adjacentes (i.e. incidentes aos nós da nova aresta). Se for o caso, ele escolhe a primeira que encontrar para colorir a nova aresta. Senão, incrementa o número de cores no grafo.

Figura 5.5: Colorindo as arestas do grafo.



Algorithm 8 Função de coloração de uma nova aresta no *Experience Map*.

```
1:  procedure COLOR_LINK(experiences, new_link, num_colors)
2:      a = experiences[new_link→id_from] // nodo que a aresta sai
3:      b = experiences[new_link→id_to] // nodo que a aresta entra
4:      Adj_links = { a->links_from, a->links_to,
                      b->links_from, b->links_to }

5:      Cria um vetor booleano colorVec de tamanho num_colors
6:      for link in Adj_links do
7:          Marca em colorVec que link->color foi usada
8:      end for

9:      // busca primeira cor de colorVec não marcada
10:     c = 0
11:     while colorVec[c] marcada || c < num_colors do
12:         c++
13:     end while

14:     // se não achou cor válida, aumenta num_colors
15:     if c == num_colors then
16:         num_colors++
17:     end if

18:     new_link→color = c
19: end procedure
```

Depois de ter o grafo colorizado a função *iterate* é executada n vezes onde n é o número de cores usados para colorir o grafo, assim nodos ligados por arestas de mesma cor podem ser corrigidos em paralelo sem problema de concorrência, conforme apresenta o Algoritmo 9.

Algorithm 9 Função ***iterate*** do *RatSLAM_CUDA*.

```
1:  procedure ITERATE_CUDA(exp_cuda, links_cuda)
2:      //Quando um novo nodo é criado também é criado a sua cópia na memória da GPU
3:      int exp_loops = 10
4:      for i in 1...exp_loops do
5:          for c in 1...num_colors do
6:              dim3 G(num_links)
7:              dim3 B(1)
8:              //Chama a função que faz o relaxamento de todas as arestas de mesma cor em paralelo
9:              iterateLink_cuda<<< G, B>>>(links_cuda, exp_cuda, c)
10:         end for
11:     end for
12:     if num_experiences % 100 == 99 then
13:         for e in 1...num_experiences do
14:             //Copia o dados de experiências da GPU para a CPU para a visualização
15:             experiences[e] ← exp_cuda[e]
16:         end for
17:     end if
18: end procedure
```

6 RESULTADOS

Os algoritmos *RatSLAM* e *RatSLAM_CUDA* foram testados em duas máquinas com configurações diferentes, um computador pessoal com uma placa de vídeo GeForce GTX 1050 e em uma placa Jetson TX2 ambas da fabricante NVidia. Trinta e três testes de cada algoritmo foram executados para cada um dos ambientes. Nos testes foram medidos as médias de tempos de execução dos métodos de cada módulo dos dois algoritmos. Os testes foram feitos usando o *dataset* irataus (MILFORD, 2019).

6.1 Configurações das Máquinas de Teste

6.1.1 Computador Pessoal (PC)

Especificações técnicas do PC utilizado:

- **GPU:** NVIDIA GeForce GTX 1050 3GB GDDR5 96Bit
- **CPU:** Intel Core i5-8600 Coffee Lake LGA 1151 3.1Ghz 9MB Cache
- **Memória:** 8GB 2400MHz DDR4
- **Armazenamento:** 1TB 7200 RPM 64MB Cache

A Figura 6.1 apresenta as configurações da placa GeForce utilizada nos testes.

6.1.2 Jetson

Especificações técnicas da placa Jetson TX2:

- **GPU:** NVIDIA Pasca Architecture GPU with 256 CUDA cores
- **CPU:** Dual-core NVIDIA Denver 2 64-bit CPU and quad-core ARM A57 Complex
- **Memória:** 4GB 128-bit LPDDR4
- **Armazenamento:** 16GB eMMC 5.1 Flash

A Figura 6.2 apresenta as configurações da placa Jetson utilizada nos testes.

Figura 6.1: Configurações da placa de vídeo GeForce GTX 1050.

```
mariane@bruce:/usr/local/cuda-10.1/samples/1_Utilities/deviceQuery$ ./deviceQuery
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "GeForce GTX 1050"
  CUDA Driver Version / Runtime Version          10.1 / 10.1
  CUDA Capability Major/Minor version number:    6.1
  Total amount of global memory:                 3019 MBytes (3165323264 bytes)
  ( 6) Multiprocessors, (128) CUDA Cores/MP:     768 CUDA Cores
  GPU Max Clock rate:                            1518 MHz (1.52 GHz)
  Memory Clock rate:                             3504 Mhz
  Memory Bus Width:                              96-bit
  L2 Cache Size:                                 786432 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 16384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 65536
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 2 copy engine(s)
  Run time limit on kernels:                     Yes
  Integrated GPU sharing Host Memory:            No
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Device supports Compute Preemption:            Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 1 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 10.1, CUDA Runtime Version = 10.1, NumDevs = 1
Result = PASS
mariane@bruce:/usr/local/cuda-10.1/samples/1_Utilities/deviceQuery$
```

Figura 6.2: Configurações da placa Jetson TX2.

```
mariane@gnode34:/home/nvidia/NVIDIA_CUDA-9.0_Samples/1_Utilities/deviceQuery$ sudo ./deviceQuery
[sudo] password for mariane:
./deviceQuery Starting...

 CUDA Device Query (Runtime API) version (CUDART static linking)

Detected 1 CUDA Capable device(s)

Device 0: "NVIDIA Tegra X2"
  CUDA Driver Version / Runtime Version          9.0 / 9.0
  CUDA Capability Major/Minor version number:    6.2
  Total amount of global memory:                 7854 MBytes (8235802624 bytes)
  ( 2) Multiprocessors, (128) CUDA Cores/MP:     256 CUDA Cores
  GPU Max Clock rate:                            1301 MHz (1.30 GHz)
  Memory Clock rate:                             1600 Mhz
  Memory Bus Width:                              128-bit
  L2 Cache Size:                                 524288 bytes
  Maximum Texture Dimension Size (x,y,z)         1D=(131072), 2D=(131072, 65536), 3D=(16384, 16384, 1
6384)
  Maximum Layered 1D Texture Size, (num) layers  1D=(32768), 2048 layers
  Maximum Layered 2D Texture Size, (num) layers  2D=(32768, 32768), 2048 layers
  Total amount of constant memory:               65536 bytes
  Total amount of shared memory per block:       49152 bytes
  Total number of registers available per block: 32768
  Warp size:                                     32
  Maximum number of threads per multiprocessor:  2048
  Maximum number of threads per block:           1024
  Max dimension size of a thread block (x,y,z): (1024, 1024, 64)
  Max dimension size of a grid size    (x,y,z): (2147483647, 65535, 65535)
  Maximum memory pitch:                          2147483647 bytes
  Texture alignment:                             512 bytes
  Concurrent copy and kernel execution:          Yes with 1 copy engine(s)
  Run time limit on kernels:                     No
  Integrated GPU sharing Host Memory:            Yes
  Support host page-locked memory mapping:       Yes
  Alignment requirement for Surfaces:            Yes
  Device has ECC support:                        Disabled
  Device supports Unified Addressing (UVA):      Yes
  Supports Cooperative Kernel Launch:            Yes
  Supports MultiDevice Co-op Kernel Launch:      Yes
  Device PCI Domain ID / Bus ID / location ID:   0 / 0 / 0
  Compute Mode:
     < Default (multiple host threads can use ::cudaSetDevice() with device simultaneously) >

deviceQuery, CUDA Driver = CUDART, CUDA Driver Version = 9.0, CUDA Runtime Version = 9.0, NumDevs = 1
Result = PASS
```

6.2 Testes

6.2.1 Medidas *Local View*

A Tabela 6.1 apresenta os testes do *RatSLAM* original, onde as médias de tempo executadas no PC são menores do que as médias executadas na Jetson, visto que as configurações de CPU e memória do PC são melhores do que a CPU e memória da Jetson. Os testes do *RatSLAM_CUDA* exibido na Tabela 6.2 mostram que a versão paralela do algoritmo é mais rápida do que a original, embora a média de tempo de execução do *RatSLAM_CUDA* na Jetson não seja tão eficiente quando a média de execução no PC, devido a menor capacidade de memória e CPU da placa Jetson.

Tabela 6.1: Média e desvio padrão do módulo ***Local View*** usando o *RatSLAM*.

	Tempo (ms)			
	PC		Jetson	
Método	Média	Des. Padrão	Média	Des. Padrão
main_lv.cpp	-	-	-	-
+ lv->on_image	14.582	2.064	24.235	0.134
+ compare	13.639	1.857	22.662	0.129

Tabela 6.2: Média e desvio padrão do módulo ***Local View*** usando o *RatSLAM_CUDA*.

	Tempo (ms)			
	PC		Jetson	
Método	Média	Des. Padrão	Média	Des. Padrão
main_lv.cpp	-	-	-	-
+ lv->on_image	2.034	0.293	17.052	0.112
+ compare_cuda	1.253	0.051	15.560	0.113

6.2.2 Medidas *Pose Cells*

A Tabela 6.3 apresenta os testes do módulo *Pose Cells* do *RatSLAM* original. Nesse teste podemos ver a escalabilidade da deficiência de memória da Jetson em relação ao PC, pois a média de tempo de execução na Jetson é quase o dobro do tempo de execução no PC. Já na Tabela 6.4 é exibido os dados do teste do *RatSLAM_CUDA*. Podemos observar que não houve ganho na paralelização das funções *inhibit_helper* e *excite_helper*, visto que o tempo de execução desses métodos eram baixos no *RatSLAM*

original, além disso existe um *overhead* ao fazer cópias de dados da CPU para a GPU, nesse caso o *overhead* de passar o dados do *host* para o *device* prejudicou o tempo execução fazendo com que a paralelização desses métodos não tivessem uma média de tempo menor do que os mesmos métodos não paralelizados.

Tabela 6.3: Média e desvio padrão do módulo ***Pose Cells*** usando o *RatSLAM*.

	Tempo (ms)			
	PC		**Jetson**	
Método	**Média**	**Des. Padrão**	**Média**	**Des. Padrão**
main_pc.cpp	–	–	–	–
+ pc->on_odo	23.425	2.739	52.365	0.184
+ inhibit	17.731	1.831	40.408	0.163
+ inhibit_helper	0.002	0.0002	0.004	0.00005
+ excite	4.339	4.339	8.104	0.084
+ excite_helper	0.006	0.00142	0.010	0.000333

Tabela 6.4: Média e desvio padrão do módulo ***Pose Cells*** usando o *RatSLAM_CUDA*.

	Tempo (ms)			
	PC		**Jetson**	
Método	**Média**	**Des. Padrão**	**Média**	**Des. Padrão**
main_pc.cpp	–	–	–	–
+ pc->on_odo	27.432	2.971	79.400	0.496
+ inhibit	21.789	1.800	64.641	0.429
+ inhibit_helper_cuda	0.005	0.00039	0.017	0.00017
+ excite	4.614	1.191	10.967	0.068
+ excite_helper_cuda	0.007	0.002	0.018	0.00015

6.2.3 Medidas *Experience Map*

A Tabela 6.5 exibe os testes realizados no módulo *Experience Map* com o algoritmo *RatSLAM*. Nessa tabela também podemos observar que o tempo médio de execução na placa Jetson é maior do que o tempo de execução no PC, pelos mesmos motivos relatados nas Seções 6.2.1 e 6.2.2. Os testes feitos com o *RatSLAM_CUDA* são apresentados na Tabela 6.6, onde podemos verificar que o tempo de execução nas duas configurações foram menores do que os tempos do algoritmo original mesmo com o método incluindo o processo de coloração do grafo.

Tabela 6.5: Média e desvio padrão do módulo ***Exp. Map*** usando o *RatSLAM*.

	Tempo (ms)			
	PC		Jetson	
Método	**Média**	**Des. Padrão**	**Média**	**Des. Padrão**
main_em.cpp	–	–	–	–
+ action_callback	5.030	0.067	11.828	0.064
+ iterate	4.316	0.060	11.372	0.062

Tabela 6.6: Média e desvio padrão do módulo ***Exp. Map*** usando o *RatSLAM_CUDA*.

	Tempo (ms)			
	PC		Jetson	
Método	**Média**	**Des. Padrão**	**Média**	**Des. Padrão**
main_em.cpp	–	–	–	–
+ action_callback	1.178	0.050	7.965	1.395
+ iterate_cuda	0.508	0.048	6.160	1.372

6.2.4 Análise Estatística

Com o resultado dos testes feitos, foram aplicados testes estatísticos para comprovar se houve melhora (ou piora) significativa nos tempos de execução do método após a paralelização.

Nem todos os tempos coletados nos testes aparentaram uma distribuição normal por exemplo, em alguns casos como os testes da Seção 6.2.2 temos valores de desvio padrão bastante elevados, como visto no desvio padrão da função *excite* na Tabela 6.3. Considerando experimentos independentes e aleatórios, ou seja não-pareados, e distribuições não Gaussianas, foi escolhido o teste U de Mann-Whitney (MANN; WHITNEY, 1947), também conhecido como o teste de Wilcoxon de Soma dos Postos, considerado um dos mais robustos para as condições comentadas acima. O teste U valida se os valores de uma amostra de dados A é diferente, maior ou menor que os valores de uma amostra de dados B. A comparação entre as amostras é feita da seguinte forma: primeiro se computa o posto (rank) de cada valor em relação ao total de valores das duas amostras (onde o maior valor assume o posto 1, o segundo maior valor assume o posto 2, e assim por diante); depois, se compara a soma dos postos de cada amostra. Se as duas somas forem suficientemente diferentes o teste indica que as amostras são diferentes.

A partir de uma hipótese nula (e.g.: amostra A é igual a amostra B) e com os dados das duas amostras o teste U de *Mann-Whitney* calcula com precisão se a hipótese nula pode ser rejeitada, o que implica na aceitação da hipótese alternativa, isto é que as

amostras A e B são: menores, maiores ou diferentes. O cálculo do teste U foi realizado usando a implementação em Python da biblioteca Scipy (COMMUNITY, 2019).

A hipótese nula usada é dada pela fórmula 6.1, onde A é a amostra de médias de tempo de execução no *RatSLAM* original e B é a amostra do *RatSLAM_CUDA*, já a hipótese alternativa testada é dada pela fórmula 6.2.

$$H_0 : A = B \tag{6.1}$$

$$H_1 : A > B \tag{6.2}$$

As Tabelas 6.7 e 6.8 apresentam os resultados dos testes estatísticos, onde podemos observar que nos módulos *Local View* e *Experience Map* O *RatSLAM_CUDA* obteve melhora significativa (hipótese alternativa confirmada na Tabela 6.7). Todavia, na *Pose Cells* conforme observado anteriormente o *RatSLAM_CUDA* não conseguiu superar o *RatSLAM* original, pelo contrário (hipótese nula confirmada). Entretanto, note que na função *excite_helper*, onde a matriz paralelizada é maior do que no *inhibit_helper* (7x7x7 ao invés de 5x5x5), o efeito da paralelização evita com que haja perda significativa na GeForce (não é possível provar nem melhora, nem piora conforme mostra a Tabela 6.8).

Tabela 6.7: Teste U de *Mann-Whitney* com hip. alternativa $A > B$ aplicado nas amostras de tempo do *RatSLAM* original e do *RatSLAM_CUDA*.

$H_1 = A > B$		**p-value**	
Módulo	**Método**	**GeForce**	**Jetson**
Local View	on_image	0.0000000001513%	0.0000000001513%
	compare	0.00000000015098%	0.0000000001513%
Pose Cells	on_odo	99.9982808173974%	99.9999999998618%
	excite_helper	28.1326646378393%	99.9999999999546%
	inhibit_helper	99.9999999999549%	99.9999999999741%
Experience Map	action_callback	0.00000000015106%	0.0000000001513%
	iterate	0.0000000001513%	0.0000000001513%

6.3 Mapas Construídos

A Figura 6.3 mostra uma foto do topo do ambiente mapeado no *dataset* **irataus**. O mapa da Figura 6.4a foi gerado a partir da execução do algoritmo *RatSLAM*, já o mapa da figura 6.4b foi construído a partir do algoritmo *RatSLAM_CUDA*.

O mapa do *RatSLAM_CUDA* não é idêntico ao mapa do *RatSLAM*, uma vez que

Tabela 6.8: Teste U de *Mann-Whitney* com hip. alternativa $A < B$ aplicado nas amostras de tempo do *RatSLAM* original e do *RatSLAM_CUDA*.

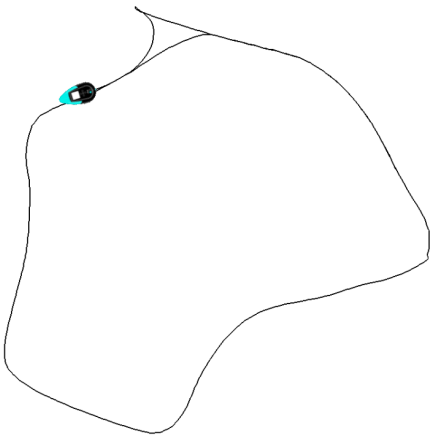
$H_1 = A < B$		**p-value**	
Módulo	**Método**	**GeForce**	**Jetson**
Local View	on_image	99.9999999998618 %	99.9999999998618%
	compare	99.9999999998621%	99.9999999998618%
Pose Cells	on_odo	0.00181793919372494%	0.0000000001513%
	excite_helper	72.3146062872974%	0.0000000000499%
	inhibit_helper	0.0000000000496%	0.0000000000285%
Experience Map	action_callback	99.9999999998621%	99.9999999998618%
	iterate	99.9999999998618%	99.9999999998618%

no *RatSLAM* todos os passos andados pelo robô são desenhados no mapa e no *RatSLAM_CUDA* o mapa é atualizado a cada 100 passos para que o *overhead* de cópias de dados da CPU para GPU e vice-versa não prejudique o desempenho da atualização do mapa, conforme apresentado no Algoritmo 9. Contudo, os dados da *Experience Map* são armazenados igualmente nos dois algoritmos, o *RatSLAM_CUDA* apenas exibe o mapa de forma diferente do *RatSLAM* original.

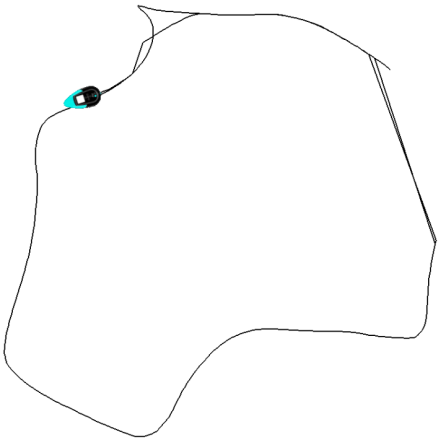
Figura 6.3: Foto do mapa.

Figura 6.4: Exemplos de mapas construídos.

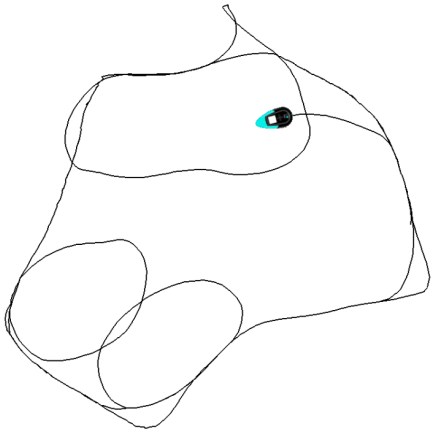
(a) *RatSLAM* Original passo 100.



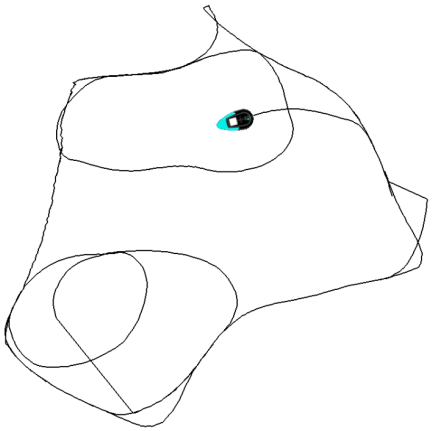
(b) *RatSLAM_CUDA* passo 100.



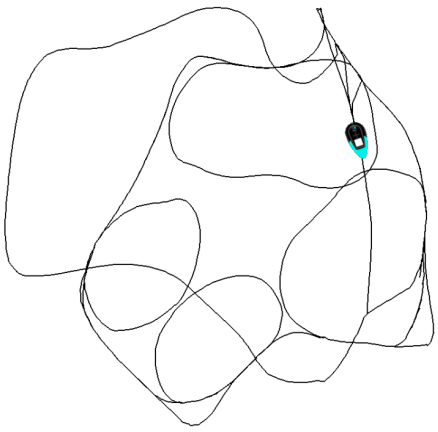
(a) *RatSLAM* Original passo 250.



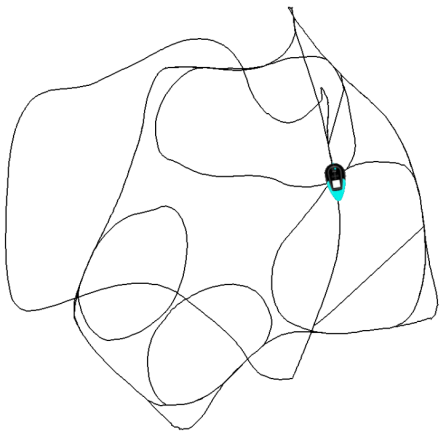
(b) *RatSLAM_CUDA* passo 250.



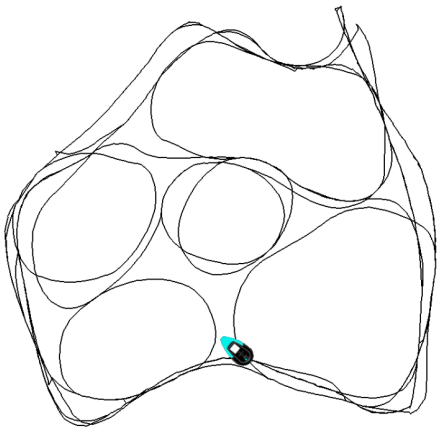
(a) *RatSLAM* Original passo 415.



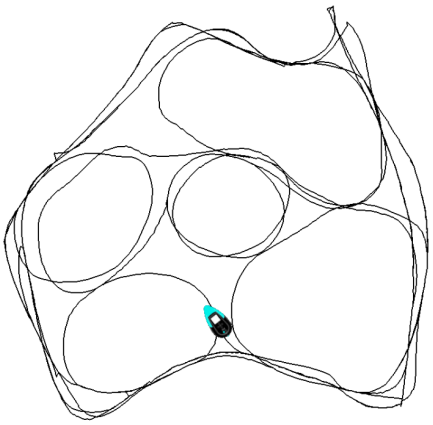
(b) *RatSLAM_CUDA* passo 415.



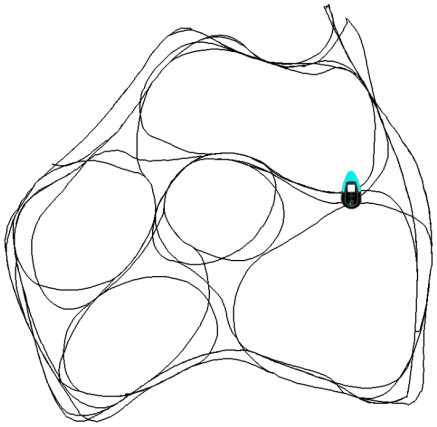
(a) *RatSLAM* Original passo 700.



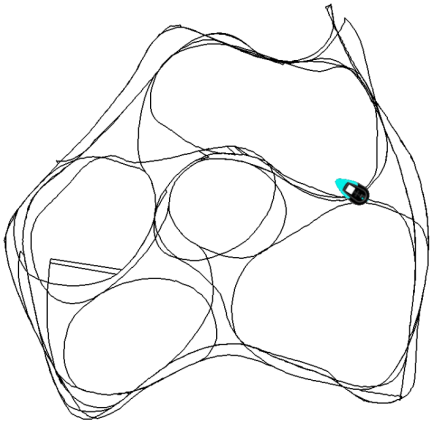
(b) *RatSLAM_CUDA* passo 700.



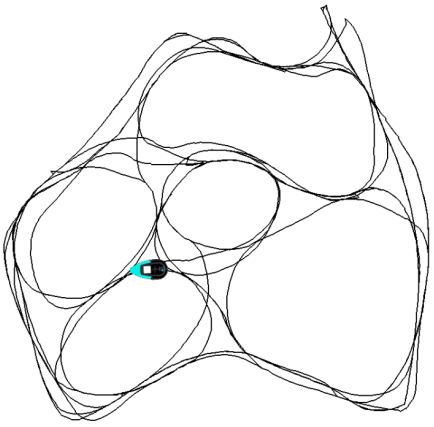
(a) *RatSLAM* Original passo 800.



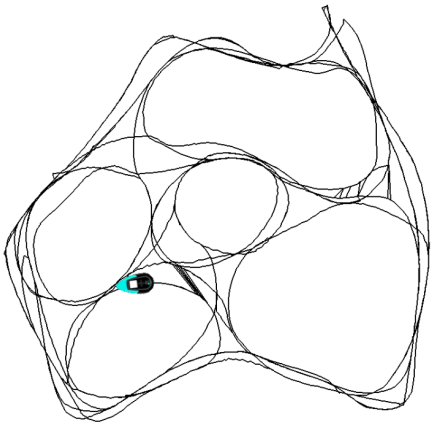
(b) *RatSLAM_CUDA* passo 800.



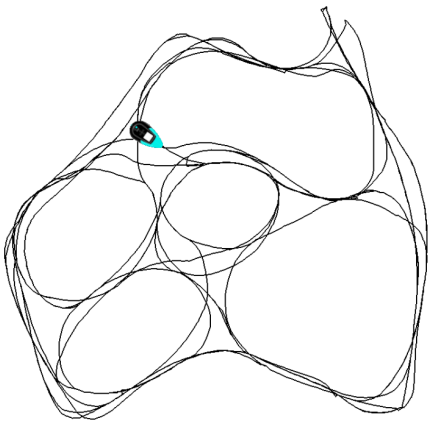
(a) *RatSLAM* Original passo 900.



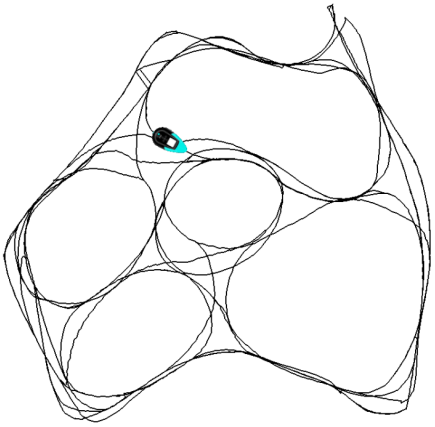
(b) *RatSLAM_CUDA* passo 900.



(a) *RatSLAM* Original passo 940.



(b) *RatSLAM_CUDA* passo 940.



7 CONCLUSÃO

Este trabalho tinha como objetivo embarcar a técnica de Localização e Mapeamento Simultâneos usando uma única câmera chamada *RatSLAM* em uma placa de pequeno porte Jetson TX2, que no futuro poderá ser acoplada a um veículo aéreo. O *RatSLAM* foi embarcado na placa e alguns métodos específicos foram modificados para serem executados na GPU que a placa possui, uma vez que o resultado da execução em CPU na placa Jetson TX2 não é eficiente. Portanto, a nova proposta de versão desse algoritmo chamada de *RatSLAM_CUDA* foi embarcada, testada e comparada com a versão original do *RatSLAM*.

A partir das médias de tempos dos dois algoritmos foi feita uma análise estatística a fim de medir se a nova proposta do algoritmo erá mais eficiente do que a versão anterior. Nessa análise foi verificado que em dois dos três módulos do *RatSLAM* tiveram melhorias enquanto que em um módulo não foi possível melhorar o desempenho devido ao *overhead* de cópias de dados da CPU para a GPU. Contudo, no módulo onde não houve ganho de tempo de execução os métodos originais já executavam a tarefa rapidamente e por isso a paralelização não foi eficiente.

Em suma, um dos grandes problemas de desempenho do *RatSLAM* é a comparação de *templates* de imagens observadas com as imagens armazenadas em memória. Esse problema é contornado no *RatSLAM_CUDA*, uma vez que a comparação é feita na GPU em paralelo e não mais de maneira sequencial como era feita na versão original. Dessa maneira, a nova versão do algoritmo poderá ser usada como uma alternativa ao *RatSLAM* original.

Como trabalhos futuros, há possibilidade de investigar mais profundamente formas de aprimorar a paralelização da atualização das *Pose Cells*. No caso, é preciso tratar o problema de concorrência na atualização das matrizes sobrepostas, e uma forma de abordar o problema é seguir a ideia aplicada na otimização de grafos no *Experience Map*. Por fim, pretende-se aplicar o novo método em um robô real aéreo, no entanto, isso demandará modificações nos módulos do ROS para aquisição de dados, obtenção de odometria, entre outros.

REFERÊNCIAS

BALL, D. **Open RatSLAM**. 2018. Available from Internet: <https://github.com/davidmball/ratslam>.

COMMUNITY, T. S. **scipy.stats.mannwhitneyu**. 2019. Available from Internet: <https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html#scipy.stats.mannwhitneyu>.

GRISETTI, G. et al. A tutorial on graph-based slam. **IEEE Intelligent Transportation Systems Magazine**, IEEE Press, Piscataway, NJ, USA, v. 2, n. 4, p. 31–43, winter 2010. ISSN 1939-1390.

HAHNEL, D. et al. An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In: **Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. Piscataway, NJ, USA: IEEE Press, 2003. v. 1, p. 206–211.

JULIER, S. J.; UHLMANN, J. K. A new extension of the kalman filter to nonlinear systems. In: **Proceedings of the 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls**. Bellingham, WA, USA: SPIE, 1997.

MAKARENKO, A. A. et al. An experiment in integrated exploration. In: **Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)**. Piscataway, NJ, USA: IEEE Press, 2002. v. 1, p. 534–539.

MANN, H. B.; WHITNEY, D. R. On a test of whether one of two random variables is stochastically larger than the other. **Ann. Math. Statist.**, The Institute of Mathematical Statistics, v. 18, n. 1, p. 50–60, 03 1947. Available from Internet: <https://doi.org/10.1214/aoms/1177730491>.

MILFORD, M. J. **OpenRatSLAM datasets**. 2019. Available from Internet: <https://wiki.qut.edu.au/display/cyphy/openRatSLAM+datasets>.

MILFORD, M. J.; WYETH, G. F. Single camera vision-only slam on a suburban road network. In: **2008 IEEE International Conference on Robotics and Automation**. [S.l.: s.n.], 2008. p. 3684–3689. ISSN 1050-4729.

MILFORD, M. J.; WYETH, G. F.; PRASSER, D. Ratslam: a hippocampal model for simultaneous localization and mapping. In: **Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on**. [S.l.: s.n.], 2004. v. 1, p. 403–408 Vol.1. ISSN 1050-4729.

MONTEMERLO, M. et al. Fastslam: A factored solution to the simultaneous localization and mapping problem. In: **Proceedings of the AAAI National Conference on Artificial Intelligence**. Edmonton, Canada: AAAI, 2002.

MURPHY, K. P. Bayesian map learning in dynamic environments. In: **Proceedings of the 12th Advances in Neural Information Processing Systems (NIPS)**. Cambridge, MA, USA: MIT Press, 1999. p. 1015–1021.

SMITH, R.; SELF, M.; CHEESEMAN, P. Estimating uncertain spatial relationships in robotics. In: COX, I. J.; WILFONG, G. T. (Ed.). **Autonomous Robot Vehicles**. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 1990. v. 8, chp. Autonomous robot vehicles, p. 167–193. ISBN 0-387-97240-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=93002.93291>.

STACHNISS, C. **Robotic Mapping and Exploration**. 1st. ed. Berlin Heidelberg: Springer Publishing Company, Incorporated, 2009. (Springer Tracts in Advanced Robotics). ISSN 1610-7438. ISBN 978-3-642-01096-5. Available from Internet: <http://www.springerlink.com/content/978-3-642-01096-5/\#section=72328&\#38;page=1>.

THRUN, S.; MONTEMERLO, M. The graph slam algorithm with applications to large-scale mapping of urban structures. **The International Journal of Robotics Research**, Sage Publications, Inc., Thousand Oaks, CA, USA, v. 25, n. 5-6, p. 403–429, 2006. Available from Internet: <http://ijr.sagepub.com/content/25/5-6/403.abstract>.

Sistema de Localização e Mapeamento Simultâneos Embarcado para Robôs Autônomos

Mariane Teixeira Giambastiani[1], **Edison Pignaton de Freitas (Orientador)**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)

mtgiambastiani@inf.ufrgs.br, edison.pignaton@inf.ufrgs.br

Abstract. *Truly autonomous robots need to map the environment that surrounds them while locating themselves in this environment. Therefore, Simultaneous Localization and Mapping (SLAM) is a fundamental problem of mobile robotics. An important SLAM technique proposed in the literature is RatSLAM. This proposal computationally models a network of competitive attractors based on the functioning of rodent hippocampus. RatSlam is capable of locating and mapping in real time using a single camera. This work aims to develop an implementation of the RatSLAM system embedded in an NVIDIA Jetson TX2 development kit, in order to couple this board in an aerial vehicle in the future.*

Resumo. *Robôs verdadeiramente autônomos necessitam realizar o mapeamento do ambiente ao seu redor e estimar a sua localização no mapa. Por isso, Localização e Mapeamento Simultâneos (SLAM) é um problema fundamental da robótica móvel. Uma importante técnica de SLAM proposta na literatura é o RatSLAM: que modela computacionalmente uma rede de atratores competitivos baseado no funcionamento de hipocampos de roedores. O RatSlam é capaz de realizar a localização e o mapeamento em tempo real usando uma única câmera. Esse trabalho de graduação tem como objetivo, o desenvolvimento do sistema RatSLAM embarcado numa placa Jetson TX2 da NVIDIA, a fim de futuramente acoplar a placa em um robô aéreo.*

1. Introdução

Nos últimos anos estamos presenciando o avanço científico e tecnológico de robôs aéreos, que estão cada vez mais presentes em aplicações científicas e comerciais. Contudo, o controle desses robôs ainda é praticamente manual, embora muitas das situações que necessitam da utilização de drones são atividades de risco que podem prejudicar a curto e a longo prazo a saúde de quem os controla. Um exemplo é o uso de veículos aéreos para pulverização de agrotóxicos em lavouras, onde o operador do robô tem que ficar próximo da lavoura para poder guiá-lo, deixando o piloto perigosamente próximo da exposição a produtos químicos. Esse tipo de situação de risco pode ser evitada se o controle do robô aéreo for autônomo. Neste caso, a autonomia do robô faria com que ele sozinho pulverizasse uma região da lavoura sem ser guiado por um piloto.

O desenvolvimento de sistemas autônomos vem sendo estudado há décadas na área de robótica móvel. Vários trabalhos nessa área tratam de problemas de localização, mapeamento e navegação autônoma de robôs. Um dos problemas fundamentais da área é o chamado problema de SLAM (*Simultâneous Localization and Mapping*), que consiste em estimar simultaneamente o mapeamento de um ambiente previamente desconhecido e

a localização precisa do robô neste mapa que está sendo estimado. Uma enorme variedade de técnicas de SLAM tem sido apresentada desde o início da década de 90, mas o grande foco nos últimos anos são as técnicas de SLAM visual, ou seja, técnicas que utilizam apenas câmeras como fonte de informação (ao invés, por exemplo, de sensores de alcance como sonares ou laser). Na Figura 1 temos um exemplo de um mapa, baseado em grade de ocupação, gerado com e sem o uso de um método de SLAM.

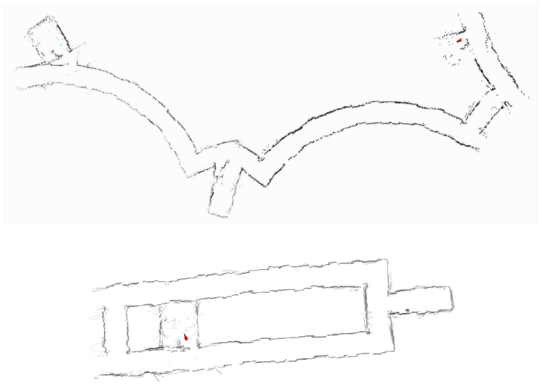


Figura 1. No topo existe mapa construído sem o uso de SLAM e em baixo o mesmo mapa construído utilizando um método de SLAM.

Este trabalho tem como objetivo a implementação de uma técnica de Localização e Mapeamento Simultâneos usando uma única câmera chamada RatSLAM. A técnica será embarcada em uma placa de pequeno porte, que futuramente poderá ser acoplada a um veículo aéreo. Um dos motivos para a escolha da técnica RatSLAM é que ela apresenta particularidades na estratégia de localização (reconhecimento de lugares) a qual acreditamos ter um grande potencial para futuras aplicações com múltiplos robôs. O RatSLAM foi originalmente desenvolvido para robôs terrestres, por isso este trabalho fará primeiro o desenvolvimento também para esses robôs, todavia, pretende-se posteriormente fazer a extensão do sistema para robôs aéreos.

Este trabalho esta organizado da seguinte forma. A seção 2 apresentará o que é o problema de SLAM e como ele é solucionado. Na seção 3 será mostrado o método RatSLAM, qual o seu objetivo, e como foi o desenvolvimento pelo seus autores. A seção 4 discutirá sobre os trabalhos existentes na área de *Visual SLAM* e suas diferenças em relação ao RatSLAM. Na seção 5 será apresentada a metodologia que será usada para realizar o estudo e desenvolvimento desse trabalho. A seção 6 fará uma breve conclusão das pesquisas feitas para o desenvolvimento do trabalho de conclusão. Por fim, a seção 7 mostrará o cronograma de atividades previstas para a conclusão deste trabalho de graduação.

2. Localização e Mapeamento Simultâneos (SLAM)

A robótica móvel é a área que desenvolve métodos com a finalidade de tornar robôs verdadeiramente autônomos. Portanto, os três principais problemas estudados nessa área são: mapeamento, localização e planejamento de movimento. Todavia, a solução desses métodos separadamente é praticamente inviável, por isso em geral são mais utilizadas a combinação desses problemas [Makarenko et al. 2002], como se pode observar na Figura 2.

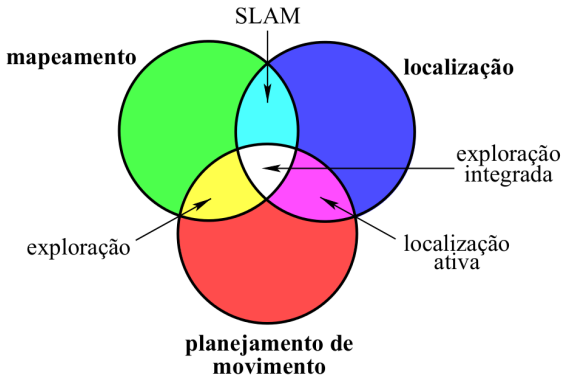


Figura 2. Combinação dos problemas da robótica móvel. Figura adaptada de [Makarenko et al. 2002]

Nesse capítulo serão apresentados os problemas de localização, mapeamento e em seguida, a definição do problema de SLAM (*Simultaneous Localization and Mapping*) e a sua importância para a navegação de robôs autônomos.

2.1. Definição de SLAM e outros problemas básicos da robótica móvel

Conforme o próprio nome informa, o problema de SLAM é a combinação dos problemas de mapeamento e localização.

No problema de **mapeamento**, um robô autônomo precisa construir uma representação precisa dos objetos ou obstáculos presentes no ambiente que o cerca. O objetivo é permitir com que o robô seja capaz de navegar no local desviando de paredes e obstáculos. Para construir um mapa é necessário saber a localização global do robô, ou ao menos ter uma boa estimativa dessa localização, assim é possível gerar um mapa de obstáculos a partir de dados coletados por sensores (e.g. sonares e câmeras).

No problema de **localização**, o robô deve estimar sua posição em relação ao mapa do ambiente. Caso ela seja desconhecida a priori, o problema é chamado de localização global. Caso a posição inicial seja conhecida, o problema é chamado de localização local ou *position tracking*. Neste caso, o robô deve ser capaz de corrigir erros de posicionamento que se acumulam devido ao ruído nas observações dos sensores. Resolver o problema de localização é muito importante para a navegação de um robô em um ambiente em que se conhece o mapa, pois sem isso o robô pode facilmente colidir com obstáculos ao seu redor. Quando o mapa da região onde se encontra o robô é conhecido, é possível estimar a sua localização usando dados coletados por sensores acoplados no robô.

Na prática, esses dois problemas geralmente não podem ser resolvidos separadamente. Logo, um dos problemas fundamentais da robótica móvel é, construir um mapa sem ter a localização precisa do robô e estimar a localização do robô sem ter um mapa preciso. Esse problema, chamado de **Localização e Mapeamento Simultâneos** (comumente abreviado de **SLAM**), é uma tarefa bastante complexa dado que ele soluciona dois problemas ao mesmo tempo, como mostra a Figura 2.

A Figura 3 apresenta a exemplificação do problema. Supondo que um robô se desloca num ambiente onde não existe um mapa a priori e a localização dele não é conhecida. Contudo, se sabe a posição de marcadores presente no ambiente e também o conjunto de

ações de controle de movimento que será executado pelo robô. Assumindo uma posição inicial de partida, é possível estimar iterativamente a localização, a partir dos dados conhecidos, por seguinte se constrói um mapa com base na estimativa da localização.

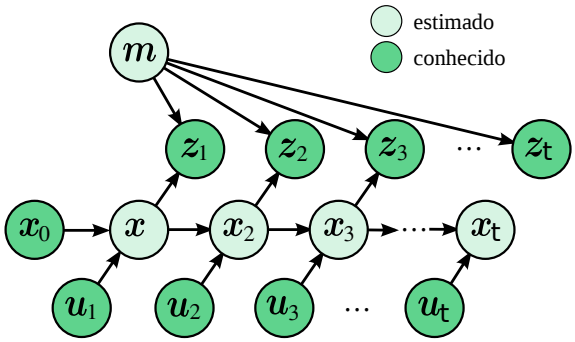


Figura 3. Modelo do problema de SLAM.

O modelo mostrado na Figura 3 apresenta os seguintes estados:

- u_t: representa a ação de navegação tomada pelo robô no instante t. Pode ser descrita por medidas de velocidade linear e angular, ou por medidas de odometria, etc.
- z_t: representa as observações feitas pelo robô no instante t. Em um ambiente descrito por *features*, pode ser as posições relativas de cada *feature* em relação ao robô. Considerando um robô equipado com sensores de alcance (e.g. sonar/laser), pode ser as medidas de distância lidas naquele instante.
- x_t: é o estado do robô (posição e orientação) no instante t. Com exceção da postura inicial x_0, que geralmente se assume como sendo o local (0,0,0) do mapa, deve ser estimado em função das ações $u_{1:t}$ e observações $z_{1:t}$ feitas pelo robô.
- m: é o mapa descrevendo os objetos no ambiente. É construído a partir da localização estimada do robô $x_{0:t}$ e das observações $z_{1:t}$ feitas por ele.

2.2. Tipos de SLAM

A construção de um mapa com SLAM pode ser feito de diferentes formas: baseado em grades de ocupação, características (*features*) ou em geometria [Stachniss 2009]. Os mapas baseados em grades de ocupação discretizam o ambiente em células, sobre as quais é atualizada a probabilidade de estarem livres ou ocupadas. Já os mapas baseados em *features* usam algoritmos de extração de características para detectar regiões conhecidas e usa-las como *landmarks*. Muitos desses mapas são armazenados na forma de nuvem de pontos. Mapas geométricos representam os obstáculos como polígonos, ocupam menos memória do que os mapas baseado em grade, mas são mais difíceis de construir, o que os deixa restritos a tipos específicos de ambientes.

A escolha do tipo de mapa está diretamente associada ao tipo de sensor que o robô dispõe. Geralmente, robôs equipados com sensores de alcance, como laser e sonares, são muito apropriados para a construção de mapas baseados em grades de ocupação. Por outro lado, robôs equipados apenas com câmera são mais apropriados para mapas baseados em *features*.

Existem inúmeras técnicas que resolvem o problema de SLAM, contudo elas geralmente podem ser agrupadas em três tipos: soluções baseadas em Filtro de Kalman,

soluções baseadas em Filtro de Partículas Rao-Blackwellized (RBPF) e soluções baseadas na otimização de grafos de posição.

Abordagens usando Filtro de Kalman e suas variações, como os Filtros de Kalman Estendido [Smith et al. 1990] e Unscented [Julier and Uhlmann 1997], foram as abordagens pioneiras de SLAM, dominando a literatura durante a década de 90. Ainda são muito usadas para fusão sensorial em sistemas equipados com múltiplos sensores. Abordagens usando filtro de partículas [Murphy 1999] se tornaram muito populares no início da década passada após a proposta do método FastSLAM [Montemerlo et al. 2002] e, especialmente, após sua extensão para uso com grades de ocupação [Hahnel et al. 2003]. Enquanto abordagens baseadas em filtragem de Kalman usam mapas apenas baseados em *features*, abordagens de filtro de partículas permitem tratar diferentes tipos de mapas.

Mais recentemente, a abordagem que passou a dominar a literatura foi a baseada na otimização de grafos [Thrun and Montemerlo 2006, Grisetti et al. 2010]. Este tipo de abordagem constrói um grafo com a sequência de poses ocupadas pelo robô durante sua trajetória e corrige as posições conforme vai reconhecendo lugares por onde passou. Com os avanços nas técnicas de otimização, esse tipo de abordagem mostrou melhoras expressivas nos resultados em mapas de larga escala, que era um dos principais pontos fracos das duas abordagens anteriores. De fato, a grande maioria dos métodos atuais de SLAM se baseiam na otimização de grafos, e dentre eles o método RatSLAM.

3. RatSLAM

Nesse capítulo será apresentado o método RatSLAM que usa uma única câmera para realizar o mapeamento e a localização de um robô em tempo real. O RatSLAM é um sistema biologicamente inspirado em um modelo computacional de hipocampo de roedores. O método usa uma rede neural de atratividade competitiva para estimar a localização de um robô [Milford et al. 2004].

3.1. Arquitetura do RatSLAM

A arquitetura do RatSLAM funciona da seguinte forma: a estimativa das células do posicionamento do robô é atualizada usando as informações de dois processos, o de visualização local e o de integração de caminho – um dado pela informação da câmera e o outro pela odometria do robô – respectivamente, como apresenta a Figura 4. O método usa esses dois tipos de informações para realizar a estimativa da posição global de um robô sem a utilização de uma mapa inicial.

A estimativa do posicionamento do robô e feito pela rede neural atrativa a partir dos dados da visualização local observada e pela posição do robô de acordo com a odometria, respectivamente os retângulos magenta e azul da Figura 4. O treinamento da rede acontece a medida que robô se desloca, ou seja, não é necessário nenhum aprendizado prévio para uso do RatSLAM, visto que a rede aprende durante a execução do sistema.

3.2. Dinâmica do RatSLAM

Essa subseção apresenta como o RatSLAM estima a posição global de um robô, usando uma rede neural, a partir visualização da câmera e da odometria. Para obter a estimativa da localização global do robô o RatSLAM faz três processos: a associação visual, o caminho integrado e a rede atratora competitiva.

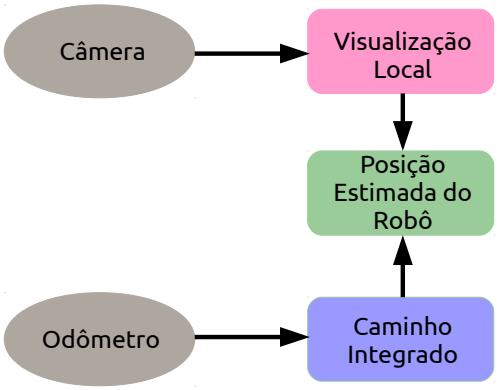


Figura 4. Arquitetura do RatSLAM. Adaptado de [Milford et al. 2004]

A associação visual é processada usando um rede neural de aprendizado de Hebbian, essa rede é empregada em sistemas biologicamente inspirados, assim como o RatSLAM. O aprendizado é usado para gravar a informação visualizada pela câmera juntamente das posições de odometria do robô ao longo do seu deslocamento.

A informação aprendida a partir da rede de Hebbian e dada pela formula 1, onde η e uma constante ($\eta = 0,05$), P é a matriz da posição do robô em relação a odometria e V é a matriz da visualização local dada pela imagem vista da câmera.

$$\beta^{t+1}_{(ijk)(lmn)} = \beta^{t}_{(ijk)(lmn)} + \eta P_{lmn} V_{ijk} \tag{1}$$

As informações da posição do robô e da e da visualização local, de acordo com a formula 1, são usadas para calcular os pesos para o treinamento da rede, conforme apresentado na Figura 5 .

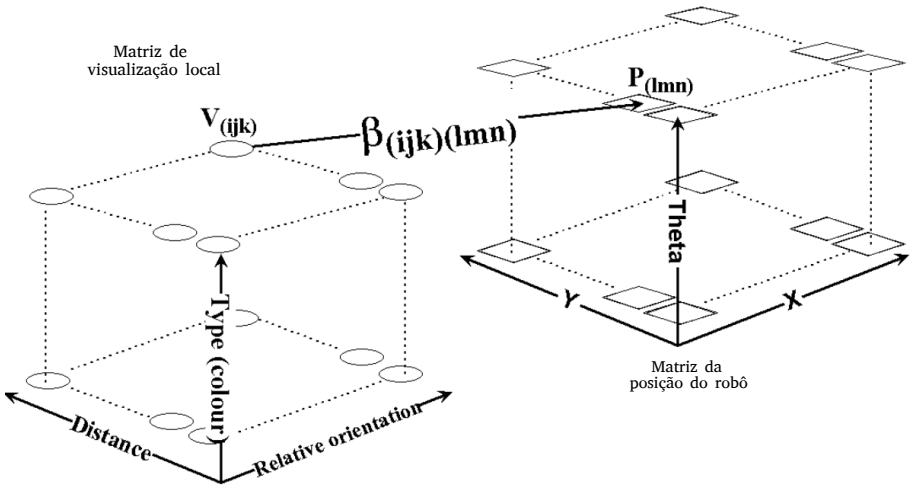


Figura 5. Associação de matrizes para realizar o aprendizado da rede. Figura extraída de [Milford et al. 2004]

A integração do caminho percorrido pelo robô é atualizado na matriz P_{lmn}. A matriz é tridimensional, porém o deslocamento do robô é quantizado numa grade de duas

dimensões, translação nos eixos x e y e rotação no eixo θ, ou seja, a terceira dimensão da matriz representa o ângulo de rotação na grade. O cálculo do deslocamento é dado pela fórmula 2, onde k é uma constante e v é a velocidade do robô.

$$\Delta x = k_x v \cos(\theta), \quad \Delta y = k_y v \sin(\theta), \quad \Delta\theta = k_\theta \omega \tag{2}$$

Depois de armazenar as informações da visualização e da caminho percorrido pelo robô, o último passo é realizar a dinâmica da rede atratora competitiva. Usando métodos probabilísticos a rede interpreta as posições estimadas do robô próximas umas das outras, como probabilisticamente semelhantes, e então as agrupa num único grupo de células dando origem a uma nova estimativa da posição do robô [Milford et al. 2004]. A dinâmica da rede atratora é realiza quatro etapas:

1. Atualização probabilística de cada camada em x e y.
2. Atualização probabilística entre as camadas em x e y, como mostra a Figura 5.
3. Inibição global: que impedi que os valores de ativação da rede sejam negativos.
4. Normalização dos valores de ativação, para valores entre zero e um, visto que são probabilidades.

4. Trabalhos Relacionados

O SLAM baseado em visão tem sido extensivamente estudado nos últimos anos. A principal razão para a sua popularidade é o uso de câmeras, que são sensores de baixo custo, possuem tamanho pequeno e baixo consumo de energia. Além disso, as câmeras fornecem muitas informações que podem ser usadas não apenas para executar o SLAM, mas também para aplicativos que envolvem processamento de informações semânticas, análise de situação de risco, etc. Esta seção resume as principais abordagens que, assim como o RatSLAM, utilizam Visual SLAM.

4.1. RatSLAM2

Os mesmos autores do RatSLAM posteriormente, em outro trabalho, usaram o método em um carro com uma câmera acoplada [Milford and Wyeth 2008]. Utilizando a câmera para estimar visualmente as velocidades angular e translacional do veículo sem qualquer interpretação geométrica do ambiente. Sendo assim, tendo uma boa estimativa de odometria visual e fazendo aprendizagem e reconhecimento de cena com o RatSLAM, foi possível fazer a localização e o mapeamento de um carro navegando por um bairro de uma cidade. Neste caso, aplicando um SLAM passivo.

4.2. SeqSLAM

Métodos de localização baseados em visão, que usam reconhecimento de características, são ineficientes em mudanças de ambientes extremas, como por exemplo, uma navegação durante o dia e a noite ou até mesmo no verão e no inverno, por causa das mudanças nas cenas. Devido à esse ponto, o SeqSLAM propôs uma abordagem de visão que não é baseada em *features* e reconhece sequencias de locais com mudanças extremas de ambiente[Milford and Wyeth 2012]. Essa abordagem escolhe a melhor localização dentro de sequências locais de navegação ao invés de simplesmente realizar o casamento de cenas. A Figura 6 apresenta duas sequencia de imagens do SeqSLAM uma de dia e outra a noite.

Figura 6. À direita temos uma sequência de imagens durante o dia e á esquerda temos uma sequência de cena durante a noite, ambas na mesma rua. Figura extraída de [Milford and Wyeth 2012]

4.3. ORB-SLAM

Outro sistema de SLAM monocular de tempo real, asssim como o RatSLAM, é o ORB-SLAM que é baseado em *features* e usa o descritor visual ORB [Mur-Artal et al. 2015]. Esse sistema funciona tanto em ambientes externos quanto internos com mudanças de iluminação. É baseado em *keyframes* que eliminam a redundância de cenas e permite a extração de *features* sem uso de multithreads ou GPUs. O ORB-SLAM executa três etapas principais para realizar o mapeamento e a localização em tempo real, o rastreamento, o mapeamento e o fechamento de loop. Na Figura 7 é mostrado uma exemplo do método.

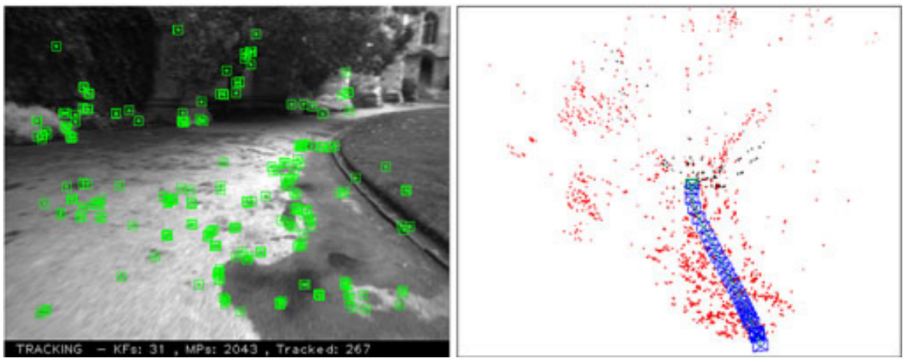


Figura 7. À direita estão as *features* extraídas e á esquerda está a nuvem de pontos das *features* em vermelho e os *keyframes* em azul. Figura retirada de [Mur-Artal et al. 2015]

4.4. ORB-SLAM2

O SLAM visual também poder ser realizado com mais de uma câmera. O método ORB-SLAM2, que é uma extensão do sistema descrito anteriormente, permite fazer o ma-

peamento e a localização em tempo real usando câmeras monoculares, estéreos e RGB-D [Mur-Artal and Tardós 2017]. O ORB-SLAM2 permite também a reutilização de mapa e a capacidade de realizar relocalização. A Figura 8 apresenta um exemplo de reconstrução de dois ambientes usando o ORB-SLAM2.

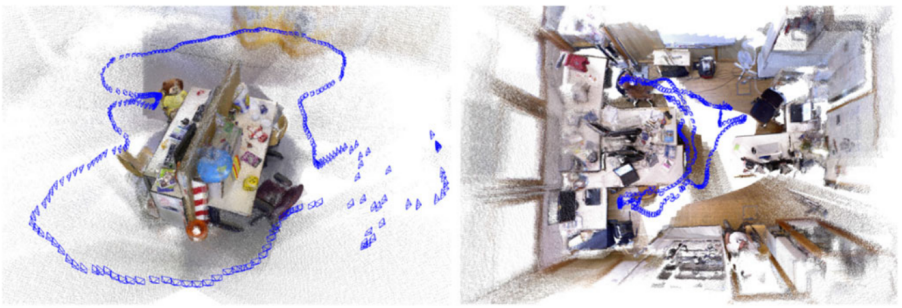


Figura 8. Nuvem de pontos densa reconstruída a partir de *keyframes* em azul usando o método ORB-SLAM2. Figura extraída de [Mur-Artal and Tardós 2017]

4.5. PL-SLAM

Uma das limitações do ORB-SLAM é a detecção de keyframes em ambientes internos com baixa texturização (i.e. pouca presença de arestas nas imagens) e o PL-SLAM [Pumarola et al. 2017] baseado na solução do ORB tenta melhorar essa deficiência adicionando nos keyframes não só marcadores pontuais, mas também linhas de marcação, mantendo a eficiência do ORB-SLAM. A Figura 9 mostra um exemplo de comparação entre os métodos ORB-SLAM e PL-SLAM.

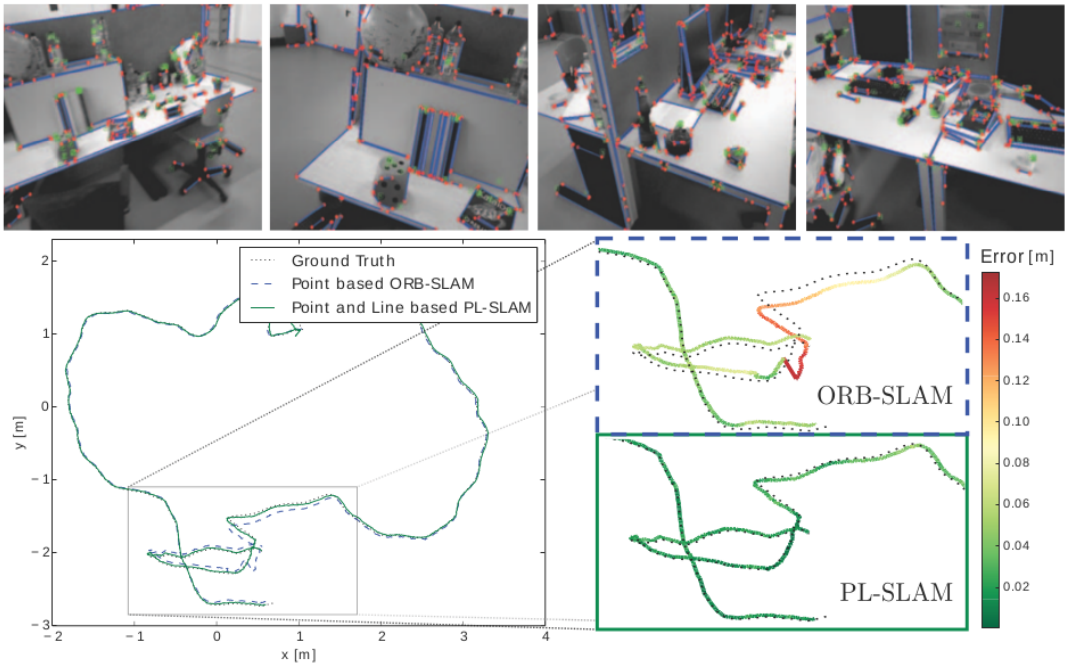


Figura 9. Comparação entre o PL-SLAM e o ORB-SLAM. Figura retirada de [Pumarola et al. 2017]

5. Metodologia

Nesta seção será apresentada todas as etapas de desenvolvimento deste trabalho de conclusão de graduação. As etapas descritas a seguir são: estudo, análise do método, implementação, resultados, análise dos resultados, escrita da monografia e da apresentação em slides.

5.1. Estudo do sistema OpenRatSLAM

O desenvolvimento desse trabalho de graduação começará com o estudo do sistema OpenRatSLAM disponibilizado na plataforma GitHub [Ball 2013]. Nessa etapa será feito o estudo, configuração e a execução do OpenRatSLAM, em um computador convencional, e será realizado os testes do sistema usando os *datasets* disponíveis no repositório.

5.2. Análise do OpenRatSLAM

Nesta etapa o sistema do OpenRatSLAM será analisado a fim de identificar as seções do código que podem ser paralelizadas, visto que, o objetivo do trabalho é implementar o RatSLAM embarcado em uma placa da NVIDIA que possui uma GPU integrada. Sabendo quais são as frações do código que são paralelizavéis o próximo passo é embarcar o sistema na placa.

5.3. Implementação do RatSLAM embarcado

Primeiramente, será feito um estudo da linguagem CUDA e das bibliotecas de redes neurais e OpenCV disponíveis na API da placa Jetson TX2 da NVIDIA. Em seguida, será realizado o desenvolvimento do sistema RatSLAM em CUDA embarcado na placa Jetson TX2.

5.4. Resultados

Nesta etapa os testes serão realizados com os mesmos *datasets* da etapa 5.1. A partir desses resultados será feito uma comparação dos testes rodados na placa e no computador convencional. Em seguida, será feito novos testes acoplando a placa em um robô terrestre.

5.5. Análise dos resultados

Depois de fazer os testes, uma análise sobre os resultados será feita. Nessa análise será comparado os novos testes executados na placa e o mesmos executados em um computador convencional. Em seguida, os dados dessa análise serão documentados na monografia de trabalho de conclusão.

5.6. Escrita da monografia

Nesta etapa será realizada a escrita da monografia, onde será detalhado todo o trabalho realizado anteriormente nas outras etapas. A monografia consistirá na expansão desse artigo juntamente das análises descritas nas seções 5.1 à 5.5.

5.7. Apresentação

Por fim, será construída uma apresentação de slides sobre a monografia. Esses slides serão usados para apresentar o trabalho desenvolvido para uma banca avaliadora no final do semestre.

6. Cronograma

Essa seção descreve a previsão do cronograma de atividades a serem realizadas para desenvolver o trabalho final de graduação. A tabela 1 apresenta o período de execução das etapas descritas na seção 5.

Tabela 1. Cronograma de atividades

	Jul	Ago	Set	Out	Nov	Dez
Estudo	X					
Análise	X	X				
Implementação		X	X			
Resultados			X			
Análise dos resultados			X	X		
Escrita				X	X	X
Apresentação						X

7. Conclusão

Este trabalho apresenta uma visão geral de um dos desafios fundamentais da robótica móvel, chamado de problema de Localização e Mapeamento Simultâneos (SLAM). Descreve um método especifico que resolve esse tipo de problema o RatSLAM e discute brevemente sobre outros métodos de Visual SLAM. O objetivo desse trabalho é servir como base inicial para a escrita de uma monografia de trabalho de conclusão de graduação em Engenharia de Computação. A proposta é desenvolver o sistema RatSLAM embarcado em uma placa de pequeno porte a fim de acoplá-la em um robô móvel.

Referências

Ball, D. (2013). Openratslam. `https://github.com/davidmball/ratslam`.

Grisetti, G., Kummerle, R., Stachniss, C., and Burgard, W. (2010). A tutorial on graph-based slam. *IEEE Intelligent Transportation Systems Magazine*, 2(4):31–43.

Hahnel, D., Burgard, W., Fox, D., and Thrun, S. (2003). An efficient fastslam algorithm for generating maps of large-scale cyclic environments from raw laser range measurements. In *Proceedings of the 2003 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 206–211, Piscataway, NJ, USA. IEEE Press.

Julier, S. J. and Uhlmann, J. K. (1997). A new extension of the kalman filter to nonlinear systems. In *Proceedings of the 11th International Symposium on Aerospace/Defense Sensing, Simulation and Controls*, Bellingham, WA, USA. SPIE.

Makarenko, A. A., Williams, S. B., Bourgault, F., and Durrant-Whyte, H. F. (2002). An experiment in integrated exploration. In *Proceedings of the 2002 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, volume 1, pages 534–539, Piscataway, NJ, USA. IEEE Press.

Milford, M. J. and Wyeth, G. F. (2008). Single camera vision-only slam on a suburban road network. In *2008 IEEE International Conference on Robotics and Automation*, pages 3684–3689.

Milford, M. J. and Wyeth, G. F. (2012). Seqslam: Visual route-based navigation for sunny summer days and stormy winter nights. In *2012 IEEE International Conference on Robotics and Automation*, pages 1643–1649.

Milford, M. J., Wyeth, G. F., and Prasser, D. (2004). Ratslam: a hippocampal model for simultaneous localization and mapping. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, volume 1, pages 403–408 Vol.1.

Montemerlo, M., Thrun, S., Koller, D., and Wegbreit, B. (2002). Fastslam: A factored solution to the simultaneous localization and mapping problem. In *Proceedings of the AAAI National Conference on Artificial Intelligence*, Edmonton, Canada. AAAI.

Mur-Artal, R., Montiel, J. M. M., and Tardós, J. D. (2015). Orb-slam: A versatile and accurate monocular slam system. *IEEE Transactions on Robotics*, 31(5):1147–1163.

Mur-Artal, R. and Tardós, J. D. (2017). Orb-slam2: An open-source slam system for monocular, stereo, and rgb-d cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262.

Murphy, K. P. (1999). Bayesian map learning in dynamic environments. In *Proceedings of the 12th Advances in Neural Information Processing Systems (NIPS)*, pages 1015–1021, Cambridge, MA, USA. MIT Press.

Pumarola, A., Vakhitov, A., Agudo, A., Sanfeliu, A., and Moreno-Noguer, F. (2017). Pl-slam: Real-time monocular visual slam with points and lines. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 4503–4508.

Smith, R., Self, M., and Cheeseman, P. (1990). Estimating uncertain spatial relationships in robotics. In Cox, I. J. and Wilfong, G. T., editors, *Autonomous Robot Vehicles*, volume 8, chapter Autonomous robot vehicles, pages 167–193. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Stachniss, C. (2009). *Robotic Mapping and Exploration*. Springer Tracts in Advanced Robotics. Springer Publishing Company, Incorporated, Berlin Heidelberg, 1st edition.

Thrun, S. and Montemerlo, M. (2006). The graph slam algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, 25(5-6):403–429.