Alexandre de Morais Amory (Coordenador-FACIN)
Aurelio Tergolina Salton (Coordenador-FENG)
Renan Guedes Maidana
Jane Liliana Chan

Exploração de Paralelismo em Algoritmo de Localização Indoor para Robôs Móveis

Porto Alegre
Janeiro 2016

Alexandre de Morais Amory (Coordenador-FACIN)
Aurelio Tergolina Salton (Coordenador-FENG)
Renan Guedes Maidana
Jane Liliana Chan

Exploração de Paralelismo em Algoritmo de Localização Indoor para Robôs Móveis

Relatório técnico referente ao projeto de Exploração de Paralelismo em Algoritmo de Localização Indoor para Robôs Móveis, realizado em conjunto com as Faculdades de Informática e Engenharia, apresentado à Pontifícia Universidade Católica do Rio Grande do Sul.

Pontifícia Universidade Católica do Rio Grande do Sul
Faculdade de Informática (FACIN)
Faculdade de Engenharia (FENG)

Porto Alegre
Janeiro 2016

Resumo

A robótica tem sido empregada com sucesso desde a década de 50 em linhas de montagem. Entretanto, os robôs comumente utilizados são braços robóticos, fixos, e instalados em um ambiente criado em função dos robôs. A desvantagem desta abordagem é que a falta de mobilidade do robô reduz muito as possíveis aplicações dos mesmos. Neste sentido tem-se pesquisado em robótica móvel. Porém, robótica móvel apresenta diversos desafios computacionais tais como determinar a localização do robô em um ambiente fechado, que é uma tarefa computacionalmente custosa. O objetivo deste projeto é pesquisar e propor otimizações em algoritmos de localização empregados em robôs móveis como o filtro de partículas. Os resultados obtidos demonstram grandes ganhos de desempenho, comparáveis com o estado da arte. Outro resultado do projeto foi a implementação de um simulador do filtro de partículas, que pode ser úteis para fins educacionais.

Palavras-chaves: Localização; Robótica; Filtro de Partículas. Otimização de desempenho.

Lista de Figuras

Lista de Códigos

Sumário

1 Introdução

A área de robótica atingiu um grande sucesso na indústria moderna, com a introdução de braços robóticos. Estes robôs possuem uma posição específica em uma linha de montagem e são tipicamente compostos por articulações e manipuladores especializados em tarefas tais como, por exemplo, pintura e soldagem. Por outro lado, apesar das vantagens deste tipo de robô, tais como precisão e velocidade, eles possuem a desvantagem relacionada à falta de mobilidade, reduzindo o alcance do robô. Outra desvantagem é que, na realidade, poucos ambientes são desenvolvidos especificamente visando robôs. Ou seja, estes robôs fixos estão restritos a aplicações a ambientes estruturados do tipo 'linha de montagem'. Robótica móvel [1, 2, 3] é a área de pesquisa que trata do controle de veículos autônomos ou semi-autônomos.

Nesta área, um problema que existe desde as primeiras pesquisas é o da localização, definido como a determinação da posição atual de um robô móvel em duas dimensões (e.g. robôs terrestres) ou mais (e.g. drones, robôs submarinos) através de dados de sensores. Há diversas soluções para este problema, podendo-se citar: técnicas de odometria [4], técnicas de fusão de sensores [2] ou aplicações dos algoritmos de Monte-Carlo, conhecidos mais popularmente como Filtros de Partículas [5], que são estudados neste relatório. Uma desvantagem da utilização destes algoritmos é seu alto custo computacional, que é diretamente proporcional à quantidade de partículas utilizadas no filtro. Como a precisão e robustez do filtro estão também em proporção direta com a quantidade de partículas, o objetivo deste trabalho é realizar um estudo das maneiras possíveis de diminuir este custo computacional, incluindo otimizações e diferentes implementações, para que seja possível utilizar o filtro com rapidez e um número de partículas elevado.

1.1 Objetivos

O presente projeto de pesquisa tem por *objetivos gerais*:

a) Estudar otimizações algorítmicas de problemas relacionados à robótica, mais especificamente otimização de algoritmos de localização;

b) Fortalecer a interação entre a Faculdade de Informática e a Faculdade de Engenharia, promovendo o Núcleo de Robótica (ROCAI) da PUCRS;

O presente projeto de pesquisa tem por *objetivos específicos*:

a) Estudar o algoritmo de filtro de partículas, que é utilizado em diversas aplicações na área de robótica, incluindo localização do robô em um mapa;

b) Buscar redução do tempo de execução do algoritmo de filtro de partículas.

2 Implementações

O filtro de partículas [6, 7] é um algoritmo probabilístico (Bayesiano) de localização - isto é, ele calcula a probabilidade de um agente (e.g. robô, carro, etc) estar em vários pontos diferentes em uma região. O algoritmo alcança este objetivo através da análise de dados de sensores e/ou de modelos matemáticos de movimentação, de forma a providenciar, ao longo do tempo, uma estimativa de posição independente de distúrbios de medição ou incertezas de modelo (e.g. terrenos instáveis, ou o efeito de *drift*) [5]. Mais detalhadamente, este algoritmo pode ser descrito em três etapas [5]:

- Predição: Nesta etapa, o filtro realiza uma estimativa de movimentação das partículas através de dados dos sensores (neste caso os sensores são *encoders* dos motores de um robô terrestre) e do modelo de movimentação do robô (neste caso um robô diferencial de duas rodas [1, 2, 3]). Esta movimentação é então aplicada às partículas, sendo teoricamente igual à movimentação real do robô.

- Atualização: o filtro atualiza o *peso* de cada partícula, ou seja, a probabilidade da posição de cada partícula condizer à posição do real do robô. O peso é uma função da diferença entre dados de sensores do robô (neste caso, um laser scanner) e dados de sensores simulados das partículas. Há diversas funções de cálculo do peso, sendo que é usual utilizar uma função de distribuição normal ou bayesiana para cálculo do peso.

- Reamostragem: A última etapa do filtro, é quando são eliminadas as partículas com probabilidades baixas e duplicadas as partículas com probabilidades altas, de forma que o número total de partículas não se altere. Este passo pode ser feito em determinado período (e.g. a cada 15 iterações do filtro), por motivo de otimização de tempo computacional, sendo esta etapa diretamente proporcional à quantidade de partículas computadas. Existem diversas formas de realizar a reamostragem, diferindo em eficácia e desempenho computacional.

Neste trabalho, foi estudado um algoritmo de filtro de partículas construído originalmente no software MATLAB, simulando um robô terrestre com encoders e um sensor de distância a laser, em um mapa conhecido. Este algoritmo foi implementado com sucesso em outras linguagens de programação (MEX, C, C++) , para análise de desempenho computacional. As seções a seguir descrevem estas implementações do filtro, que estão disponíveis no repositório `https://github.com/lsa-pucrs/particle-filter`.

2.1 MATLAB™

Esta seção apresenta a implementação original do filtro de autoria do professor Dr. Aurélio Tergolina Salton. Esta implementação serviu de base para a criação das outras versões apresentadas nas seções seguintes.

O software de modelagem e simulação matemática MATLAB™ é muito popular entre engenheiros e cientistas no mundo inteiro, por proporcionar uma interface simples e de fácil utilização, e por possuir uma versatilidade incrível para realizar modelagens e simulações complexas. O programa é muito utilizado para construção, modelagem ou ajuste de sistemas de controle, por exemplo, sendo possível partir da modelagem até a implementação de um sistema de controle em pouquíssimo tempo.

A implementação original do filtro é, portanto, um tanto simplificada - isto é, apesar de utilizar funcionalidades avançadas do software, a implementação segue fielmente as três etapas do algoritmo. O trecho de código responsável pela implementação do filtro pode ser visto no código 1, cujas etapas são estudadas adiante. Ele consiste em uma estrutura de repetição que vai da primeira até a última partícula do filtro, implementando as etapas de predição e atualização para cada partícula. A etapa de reamostragem é efetuada separadamente, em intervalos de quinze iterações, por questão de otimização do desempenho do filtro e do seu tempo computacional. Estes códigos, juntamente com outras funções complementares, foram as partes implementadas nas outras linguagens de programação.

A facilidade de uso do MATLAB, porém, vem com um custo. Para tarefas mais complexas, o MATLAB apresenta grandes tempos computacionais. Em vista disso, o algoritmo do filtro fica computacionalmente custoso com um alto número de partículas, sendo assim reduzido o seu desempenho. Os resultados de tempos computacionais e desempenho para a implementação no MATLAB serão discutidos em comparação com outras implementações no Capítulo 3.

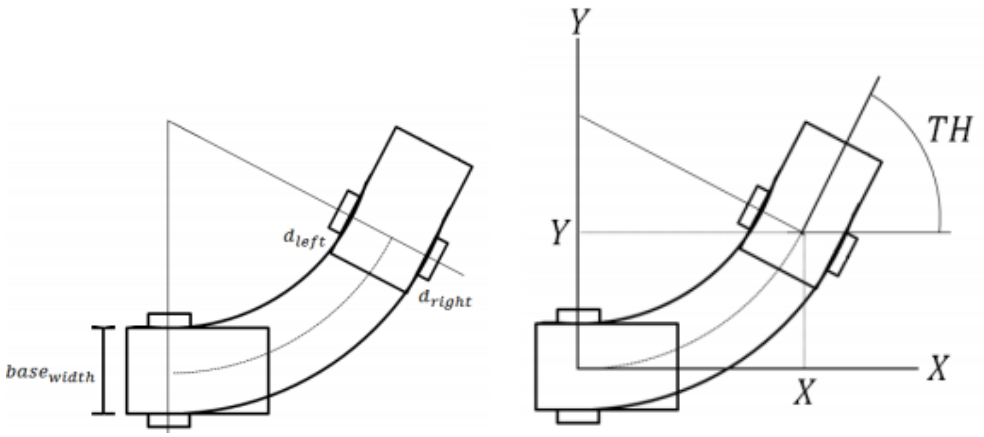
```
%% Particle Filter
for i = 1:Np
    %% Prediction
    % estimate movement according to encoders
    barp    = F_estimate_p(p(:,i),dtick_L,dtick_R,L,N,R);
    % throw in some variation in the RTR model
    p(:,i) = F_sample_odometry(p(:,i),barp,alpha);

    %% Update
    % compute particle probability
    if map(max(1,min(ceil(p(2,i)*mapscale),size(map,1))),max(1,min(ceil(p(1,i)*mapscale),
    size(map,2))))
        w(i) = 0;   % if on obstacle or outside map
    else % else compute the particle probability based on sonars
        % simulate the sonars for the particles
        z = Fast_ray_cast(p(1,i),p(2,i),p(3,i),map,max_range,angles,mapscale,1);%
        for j = 1:length(sonars)
            w(i) = w(i)*F_measurProb(z(j),sonars(j),CovSonars);
```

```
        end
    end
end
```

Código 1 – Código principal do Filtro de Partículas no MATLAB

2.1.1 Modelo de movimentação

Nesta implementação, foi simulado um robô terrestre com duas rodas, utilizando um modelo de movimentação diferencial. Este é um modelo genérico não-omnidirecional (ou seja, o robô só pode mover-se em uma direção), onde as rodas movem-se independentemente e encontram-se fixas no robô em um mesmo eixo, com uma distância constante entre sí. Este modelo é de simples implementação, pois são poucas as equações que regem seu comportamento [8, 2]. A figura 1 mostra uma ilustração deste modelo genérico. A seguir serão discutidas suas equações.



(a) (b)

Figura 1 – Modelo genérico de movimentação diferencial

O modelo descreve o deslocamento linear do robô (Dc, equação 2.3) através da média de movimentação das rodas esquerda e direita, na figura 1a identificadas como d_{left} e d_{right}. Estes deslocamentos são obtidos através da relação entre o perímetro das rodas (R), as leituras (vulgarmente conhecidas como *ticks*) de seus sensores ($ticks_{left}$ e $ticks_{right}$), os *encoders*, e a quantidade de *ticks* por rotação ($N_{ticks\ per\ rotation}$), como visto nas equações 2.1 e 2.2. A orientação angular (th, equação 2.4) é obtida através da diferença entre os deslocamentos individuais das rodas, em relação à distância entre elas (figura 1a, $base_{width}$) [8]. Ou seja:

$$d_{left} = \frac{2 \times \pi \times R \times ticks_{left}}{N_{ticks\ per\ rotation}} \tag{2.1}$$

$$d_{right} = \frac{2 \times \pi \times R \times ticks_{right}}{N_{ticks\ per\ rotation}} \tag{2.2}$$

$$Dc = \frac{d_{left} + d_{right}}{2} \tag{2.3}$$

$$th = \frac{d_{left} - d_{right}}{base_{width}} \tag{2.4}$$

Para determinar, enfim, a posição cartesiana do robô no ambiente juntamente com sua orientação angular atual (considerando estas inicialmente como $[X, Y, TH] = [0, 0, 0]$), utiliza-se as relações 2.5, 2.6 e 2.7 definidas abaixo:

$$TH = TH + th \tag{2.5}$$

$$X = X + Dc \times cos\left(\frac{th + TH}{2}\right) \tag{2.6}$$

$$Y = Y + Dc \times sen\left(\frac{th + TH}{2}\right) \tag{2.7}$$

Essas três coordenadas representam a posição e orientação atuais do robô. Estas equações são utilizadas na implementação para aplicar a mesma movimentação realizada pelo robô às partículas. Sua implementação pode ser vista no código 2, na seção 2.1.2.

2.1.2 Predição

Como discutido anteriormente, a etapa de predição é responsável por aplicar às partículas uma movimentação teoricamente idêntica à realizada pelo robô. Este modelo é aplicado às partículas através da função *F_estimate_p*, que pode ser vista no Código 2.

```
1  function bar_p = F_estimate_p(p,dtick_L,dtick_R,L,N,R)
2  %% Calculate wheel displacement
3  Dl = 2*pi*R*dtick_L/N;
4  Dr = 2*pi*R*dtick_R/N;
5  Dc = (Dl+Dr)/2;
6
7  %% Calculate [X,Y,TH] coordinates
8  bar_p(3) = p(3) + (Dr-Dl)/L;
9  bar_p(1) = p(1) + Dc*cos((p(3)+bar_p(3))/2);
10 bar_p(2) = p(2) + Dc*sin((p(3)+bar_p(3))/2);
11
12 end
```

Código 2 – Código que implementa o modelo de movimentação das partículas

Sendo objetivo principal do filtro de partículas a estimativa precisa da posição, o modelo de movimentação aplicado às partículas determina em grande parte a precisão das etapas seguintes. Porém, como não é possível construir o modelo ideal, são aplicadas variações na translação e rotação aplicadas às partículas, para compensar os distúrbios e

incertezas não inclusos no modelo [5]. A implementação destas variações pode ser vista no algoritmo da figura 3, que descreve a função *F_sample_odometry*, utilizada no código 1, do Filtro.

Esta variação é descrita como um modelo RTR (*Rotação, Translação, Rotação*), onde o movimento é decomposto em três passos: Uma rotação (δ_{rot1}), seguido de uma translação em linha reta (δ_{trans}) e então outra rotação (δ_{rot2}), como pode-se ver na figura 2. Evidentemente, para cada posição de partículas há um conjunto de parâmetros $[\delta_{rot1}\ \delta_{trans}\ \delta_{rot2}]^T$, sendo este considerado corrompido por distúrbios e incertezas do ambiente [9]. Desta forma, calcula-se uma distribuição de probabilidade, que será aplicada na posição das partículas.

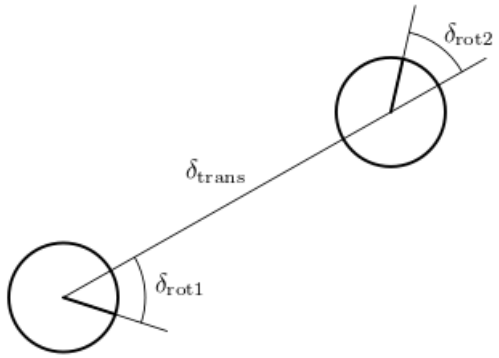


Figura 2 – Modelo de movimentação RTR

1: **Algorithm sample_motion_model_odometry**(u_t, x_{t-1})**:**

2: $\delta_{\text{rot}1} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$

3: $\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^s}$

4: $\delta_{\text{rot}2} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot}1}$

5: $\hat{\delta}_{\text{rot}1} = \delta_{\text{rot}1} - \textbf{sample}(\alpha_1\delta_{\text{rot}1} + \alpha_2\delta_{\text{trans}})$

6: $\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3\ \delta_{\text{trans}} + \alpha_4(\delta_{\text{rot}1} + \delta_{\text{rot}2}))$

7: $\hat{\delta}_{\text{rot}2} = \delta_{\text{rot}2} - \textbf{sample}(\alpha_1\delta_{\text{rot}2} + \alpha_2\delta_{\text{trans}})$

8: $x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot}1})$

9: $y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot}1})$

10: $\theta' = \theta + \hat{\delta}_{\text{rot}1} + \hat{\delta}_{\text{rot}2}$

11: *return* $x_t = (x', y', \theta')^T$

Figura 3 – Algoritmo que implementa variação RTR nas partículas

Olhando novamente ao algoritmo na figura 3, pode-se ver as diferentes partes deste algoritmo [9]:

1. As linhas 2 a 4 estimam o primeiro conjunto de rotações e translação que o robô teria feito.

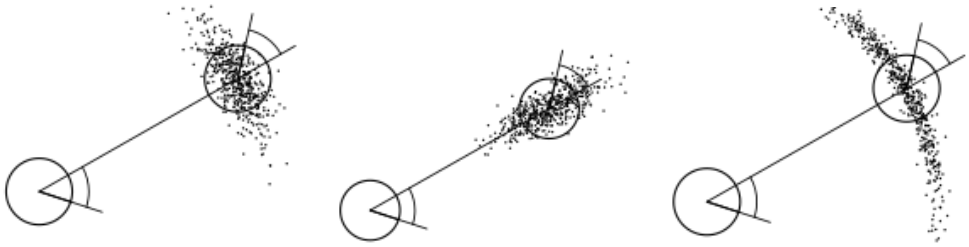
2. Nas linhas 5 a 7 são calculados os parâmetros de movimentação relativa, referente às poses anterior e atual. Juntamente, é inserida uma perturbação nestes parâmetros, dada por uma distribuição normal, na função *sample*.

3. Finalmente, nas linhas 8 a 10 são calculadas as probabilidades de erro para o conjunto de coordenadas $[X\ Y\ TH]$, que são somadas à posição atual, resultando assim na nova posição variada da partícula.

Ainda na mesma figura, pode-se ver os parâmetros $[\alpha_1\ ..\ \alpha_4]$. Estes são parâmetros de variação específicos do robô utilizado na simulação. Eles descrevem o efeito dos tipos de movimentações em relação a outros, ou a eles mesmos, como segue:

1. α_1: Efeito da rotação nas rotações.
2. α_2: Efeito da translação nas rotações.
3. α_3: Efeito da translação nas translações.
4. α_4: Efeito da rotação nas translações.

Infelizmente, não há uma metodologia precisa para a determinação destes parâmetros. Eles devem ser otimizados empiricamente, dependendo muito de diversas variáveis (e.g. tipo de robô, parâmetros construtivos, etc), sendo esta uma das grandes resalvas do algoritmo do Filtro de Partículas. Na figura 4, pode-se ver os efeitos da variação na posição. **renan, seria bom explicar estas 3 figuras abaixo o que cada uma representa. ficou bom assim?**



(a) Efeito das translações e rotações (α_2 e α_4)

(b) Efeito das translações (α_3)

(c) Efeito das rotações (α_1)

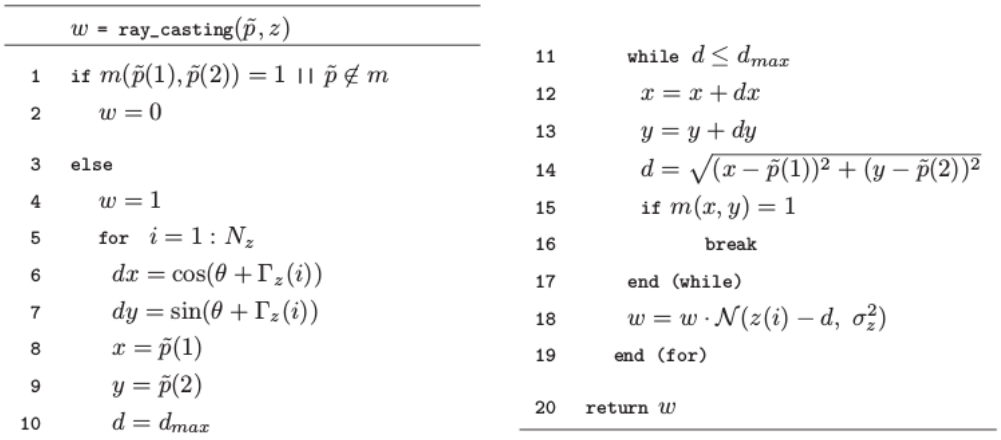
Figura 4 – Efeito das variações na posição das partículas

2.1.3 Atualização

Nesta etapa, o algoritmo atualiza o "peso" (probabilidade) das partículas de acordo, no caso, com uma distribuição gaussiana. Para tanto, é feita uma comparação entre os dados do sensor de distância do robô e das partículas, denominado *sonars*. Como trata-se de um ambiente de simulação, este sensor também é simulado, imitando um sensor

de distância a laser através do algoritmo de *Ray Casting*. Ray casting é uma técnica muito utilizada em computação gráfica, que calcula a distância entre dois pontos traçando um "raio"entre o ponto de origem até uma intersecção (no caso, uma parede) [10]. Uma limitação evidente deste, porém, é que é necessário haver um mapa conhecido do ambiente a ser navegado, sendo impossível utilizar esta técnica em ambientes dinâmicos (que mudam constantemente). O algoritmo de *Ray Casting* é implementado pela função *Fast_ray_cast*, utilizada no código do Filtro.

O algoritmo implementado nesta função pode ser visto na figura 5. Ele recebe como entrada a posição de uma partícula, já com a variação da etapa de predição conforme discutido previamente, e um vetor de medições de distância do robô em sí. Na primeira linha, a posição da partícula é comparada com as coordenadas do mapa. Se a partícula encontra-se fora do mapa ou em uma parede, esta recebe a probabilidade zero ($\omega = 0$). Caso contrário, o algoritmo simula cada sensor (N_z sendo o número de sensores) através do *Ray Casting* nas linhas 5 a 16. A variável Γ_z corresponde a um vetor com as orientações angulares de cada sensor, em referência ao plano ordenado central do robô. A condição de parada do raio encontra-se na linha 15, que dita se o raio chegou a algum obstáculo, e na linha 11, que dita se foi alcançada a distância máxima permitida pelo sensor. Na linha 18, é feita uma comparação entre a distância calculada pelo *Ray Casting* e o valor real de distância medido pelo sensor do robô *z(i)*, através de uma função de distribuição gaussiana de média d e covariância σ_z. Finalmente, na linha 20, o algoritmo retorna uma medida de probabilidade ou peso associada a cada partícula.



(a) (b)

Figura 5 – Algoritmo de *Ray Casting*

A função que implementa este algoritmo é utilizada em dois trechos de código. O primeiro uso está antes do algoritmo do filtro, para calcular os dados de distâncias do robô até as paredes do ambiente onde ele circula, e o segundo uso está no código principal do filtro (1). Esta função retorna cinco medidas de distância, simulando um sensor com campo de visão de 180 graus, com medidas espaçadas em 45 graus, como demonstrado

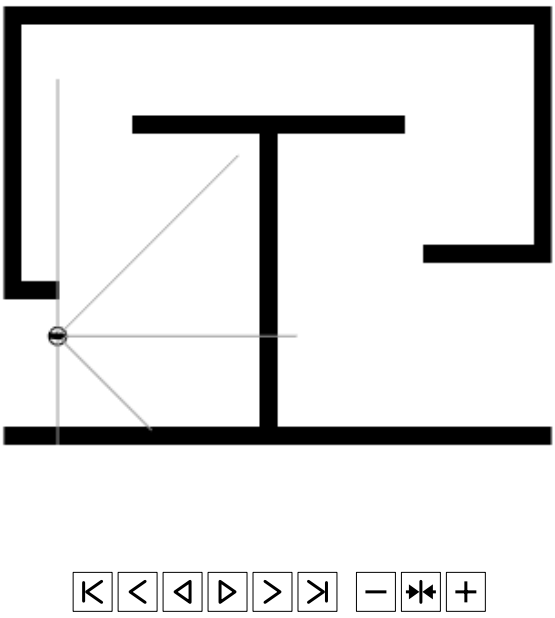


Figura 6 – Animação que demonstra a núvem de partículas convergindo para a localização real do modelo do robô.

na animação da figura 6.

2.1.4 Reamostragem

Um dos problemas que surge para o algoritmo do filtro de partículas é que com a variância incluida na movimentação das partículas, as partículas dispersam, e após algumas iterações elas estão tão separadas que já estão fora do mapa do ambiente ou que suas probabilidades são tão pequenas que não contribuem para a determinação da posição [5]. Para evitar isto, realiza-se a etapa da reamostragem, para ajustar a população de partículas baseando-se em suas probabilidades. Existem diversas formas de realizar esta etapa - eliminando as partículas com probabilidades abaixo de um certo nível e duplicando as partículas com probabilidades acima [5]. Por exemplo, neste trabalho foi utilizado o método da *roda de reamostragem*, cuja implementação pode ser vista no código 3.

```
1  % sampling wheel
2  index = round(rand(1,1)*(Np-1)+1);  % index starts randomly
3  beta = 0;
4  mw = max(w);
5  for i = 1:Np
6      beta = beta+rand(1,1)*mw*2;
7      while beta > w(index)
8          beta = beta-w(index);
9          index = index+1;
10         if index > Np
11             index = 1;
12         end
13     end
14     p2(:,i) = p(:,index);
15 end
16 p = p2;
17 w(:) = 1;
```

Código 3 – Código que implementa o algoritmo da *roda de reamostragem*

O algoritmo da *roda de reamostragem* funciona da seguinte maneira [11]:

- Primeiramente, considera-se uma roda, onde cada partícula possui uma parcela proporcional a seu peso, como pode ser visto na figura 7.
- Em seguida, determina-se um índice através de uma distribuição uniforme - por exemplo, determina-se um índice aleatório (no código, chamado de *index*) de 1 até o número de partículas, no caso, N_p, como pode ser visto na linha 2 do código 3.
- De posse do índice aleatório, é inicializada uma variável chamada β como zero (linha 3), obtém-se o maior peso entre as partículas (linha 4) e constrói-se o laço de repetição que percorre toda a população de partículas. Neste laço, é definido um novo valor de β, em função do valor deste somado uma distribuição aleatória de 0 até duas vezes o maior valor de peso entre as partículas (linha 6).
- A seguir no laço, é verificado se o peso da partícula atual não é suficiente para cobrir o valor de β - ou seja, se $Peso\ atual < \beta$, o valor do peso é subtraido de β e o índice *index* é incrementado em 1 (linhas 7 a 13).
- No caso contrário, onde $Peso\ atual \geq \beta$, a partícula é selecionada (linha 14).

Desta forma, o algoritmo seleciona as partículas proporcionalmente a seu peso. Se uma partícula ocupa uma parcela maior da roda, ela será selecionada mais vezes. Não obstante, ainda é possível para este algoritmo selecionar partículas com probabilidade mediana, o que evita que o filtro de partículas torne-se tendencioso, isto é, que as partículas

tendam a uma probabilidade alta, efetivamente eliminando o tão necessário efeito da variância no movimento das mesmas.

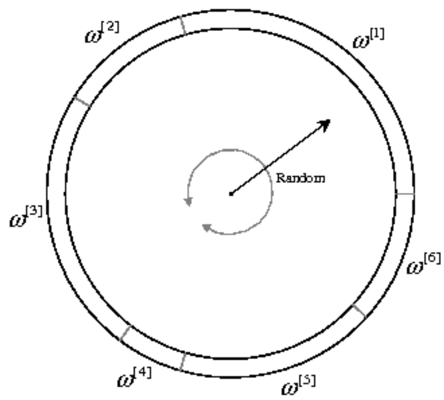


Figura 7 – Roda de reamostragem

2.1.5 Outras funções

Para o funcionamento do filtro, ao menos nesta implementação, são necessárias ainda outras funções que não são diretamente relacionadas ao algoritmo. Por exemplo, um pré-requisito para a utilização do filtro é que haja um mapa conhecido do ambiente onde o robô irá trafegar (métodos de navegação sem mapa são conhecidos como *Simultaneous Localization And Mapping*, SLAM). Este mapa é construído no ambiente de simulação como uma matriz binária, onde o valor 1 representa uma área com parede. A função que realiza a inicialização do mapa no MATLAB pode ser vista na figura 8.

```
function WGrid = F_load_map( scale )
%F_LOAD_MAP(scale) Returns a binary matrix containing the map.
%
% Each item of the matrix is either 1, for occupied position, or 0 for
% free position. Scale is the accuracy of the matrix. Each obstacle is
% represented a number of points given by 'scale'.

% Fixed world for preliminary tests
Wggrid = [1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          1 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 1 1 1 1 1 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
          1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1];
WGrid = ones(size(Wggrid)*scale);
for i = 1:size(WGrid,1)
    for j=1:size(WGrid,2) %scaling the World Grid (MAP)
        WGrid(i,j)=Wggrid(ceil(i/scale),ceil(j/scale));
    end
end
end
```

Figura 8 – Função de inicialização do mapa

Além desta função, há as funções que realizam a simulação de movimento do robô. Uma delas é a função nativa do MATLAB, *ode23*, que soluciona uma equação diferencial baseada em uma função definida pelo usuário, vista no código 4.

```
function dx = F_dif_drive_car(t,x,vr,vl,R,L)
% ------------------------------------------------------------------------
%%F_dif_drive_car(t,x,vr,vl,R,L) Simulate a differential drive car
%
%   Receives the initial pose p0 = [x0 y0 th0], the velocities of the
% wheels and the simulation time "T". Returns, the car pose p = [x y th].
% ------------------------------------------------------------------------
dx = zeros(3,1);
    dx(1) = R/2*(vr+vl)*cos(x(3));
    dx(2) = R/2*(vr+vl)*sin(x(3));
    dx(3) = R/L*(vr-vl);
end
```

Código 4 – Função de simulação de movimentação do robô.

2.2 MATLAB™+ MEX

Apesar dos custos computacionais elevados do MATLAB, ele traz consigo muitas vantagens para verificação e validação do algoritmo do filtro, como por exemplo a sua interface gráfica que permite a visualização do ambiente de simulação, como visto na figura 6. Em vista disso, decidiu-se primeiramente realizar a implementação em uma linguagem intermediária, unindo as funcionalidades do MATLAB com uma linguagem que permitisse maior desempenho nas tarefas computacionalmente mais intensivas. Assim, foi realizada a implementação no framework MEX do MATLAB. MEX (*MATLAB Executable*) permite estender o Matlab criando funções em C/C++ e acessá-las como se estas funções fossem próprias do Matlab. Outra vantagem é que a função desenvolvida em MEX é compilada ao invés de interpretada como o MATLAB, trazendo ganhos de desempenho.

Para utilização do MEX, existe uma estrutura pré-definida para qualquer programa. Esta consiste em uma função principal, chamada de função de entrada (*mexFunction*), onde são definidas as variáveis utilizadas no procedimento que se deseja implementar, além das variáveis de saída, que serão repassadas ao programa do MATLAB, visto que os ambientes de execução do MATLAB e das funções MEX são distintos. Nesta função também é chamada a rotina principal que deseja-se executar, que também é definida no mesmo arquivo MEX. Esta leva como parâmetros as variáveis de entrada (que recebem os valores passados na chamada de função no MATLAB), e as variáveis de saída.

Neste trabalho foi implementado inteiramente o laço principal do filtro, juntamente com as funções auxiliares necessárias ao laço em um único arquivo MEX, que encontra-se disponível no arquivo *mexFP.c*, diretório *src/matlab_mex* do repositório do projeto no GitHub. O uso de MEX causou um grande ganho de desempenho, o que será discutido no Capítulo 3.

2.3 Linguagem C

Esta implementação do programa, disponível no diretório *src/c*, é um passo adicional no sentido em desvincular o filtro do ambiente MATLAB. A versão MATLAB + MEX executa o programa no ambiente do MATLAB, mas com a maior parte do filtro implementada em C. A versão apresentada nesta seção executa o filtro totalmente fora do ambiente MATLAB. Entretando, para compilar este programa ainda é necessário ter o MATLAB instalado pois ele utiliza a biblioteca *mat.h* que permite ler arquivos *.mat*.

Estes arquivos *.mat* são usados para ler os inputs do filtro, como dados de odometria do carro, mapa, sonares, entre outros parâmetros. Esta leitura é realizada nas funções setDynamicVariables(), setStaticVariables(), setCarPosition(). Ao final do programa outro arquivo *.mat* é gerado com a posição das partículas calculadas pelo filtro ao longo do tempo. Este arquivo é então carregado no MATLAB simplesmente para fins de visualização dos resultados em formato gráfico, usando o rescurso de animação da versão original do filtro.

A parte principal do código está representada no código 5.

```
1  ode23(k);   // estimate the actual car position
2
3  fastRayCasting(x[k+1],y[k+1],th[k+1], 1, k, temp);  // estimate the sonar
       position
4
5  F_encoders(x[k+1]-x[k],y[k+1]-y[k],th[k+1]-th[k], k); // encoders
6  double odo[3] = {x[k], y[k], th[k]};
7  F_estimate_p(odo, k);
8
9  particleFilter(k);  // execute the k iteration of the filter
10
11 memcpy((void *)(mxGetPr(p_temp)), (void *)p, p_M*p_N*sizeof(p)); // save
       particle position into .mat
```

Código 5 – Parte principal da implementação em C do filtro.

2.3.1 Otimização na representação do mapa

Uma das otimizações realizadas nesta versão do filtro foi na representação do mapa do ambiente simulado. Em MATLAB o mapa é representado por uma matriz onde cada campo da matriz representa um tipo de dados de tipo *float*, que ocupa 8 bytes em memória. Sendo uma matriz binária, isto representa uma desperdício de memória significativo, especialmente para mapas grandes. A solução adotada foi codificar o mapa em uma matriz onde cada posição do mapa ocupasse somente um bit em memória. Isto

siginifica que cada byte de memória representa 8 posições do mapa. Em relação ao mapa em MATLAB, isto representa uma redução de 64 vezes de uso de memória para o mapa.

O ganho desta nova representação não se resumo somente em redução de uso de memória. Como os computadores possuem hierarquia de memória e memórias cache, isto significa que com uma representação mais compacta do mapa, a quantidade de *cache miss* vai reduzir drasticamente, aumentando o desempenho geral do programa.

Foi desenvolvida uma função de conversão da representação original do mapa em MATLAB chamada de *resize_map*. Ela recebe como entrada um ponteiro para o mapa desotimizado (map), para o mapa otimizado (map_opt), e as dimensões do mapa (row e col).

```
int resize_map(double *map, char *map_opt, int row, int col)
```

Como a linguagem C não acessa diretamente tipo bit de dados, foi desenvolvida uma macro auxiliar que converte a forma original de acessar o mapa (assumindo *row* linhas e *col* colunas), mas usando a nova representação compacta do mapa. O trecho do código da macro GETPIXEL(i,j) está apresentado abaixo. Basicamente ela converte i e j para acessar o bit correto da matriz que representa o mapa. O retorno é '1' se a posição está ocupada ou '0' caso contrário.

```
#define GETPIXEL(i,j) (((map_opt[i*col_opt+(j/8)] & (1<<(7-(j%8)))) != 0) ? 1:0)
```

Futuramente, ao invés de ler o mapa em MATLAB, este programa pode ler uma mapa em bitmap binário codificado para representar 1 pixel por bit. Isto simplificaria a descrição de mapas diferentes pois bastaria editar o novo mapa em um software tipo GIMP. Pensando nisto foi desenvolvido um programa chamado *bmp2hex*, disponível no diretório *test*, que converte o arquivo bmp para um formato compacto e mais simplificado. Alternativamente o filtro também poderia ler diretamente o bitmap, sem passar pela conversão realizada pelo programa *bmp2hex*.

2.4 Simulador STAGE

Ainda pensando em uma forma de facilitar a visualização e validação de resultados, optou-se por investigar a implementação do algoritmo do filtro no simulador robótico STAGE [12], que já é consolidado no ramo da robótica. Ele funciona utilizando um formato de arquivos específico, sendo o arquivo principal o "mundo". Este arquivo define o mapa, chamado *floorplan*, os atores (robôs) e os arquivos que irão ditar o comportamento dos atores. O *floorplan*, por sua vez, é definido por uma imagem em escala de cinza, o que é extremamente interessante pois permite facilmente a mudança de ambientes de navegação, sem a necessidade da tarefa trabalhosa de montar manualmente uma matriz binária como

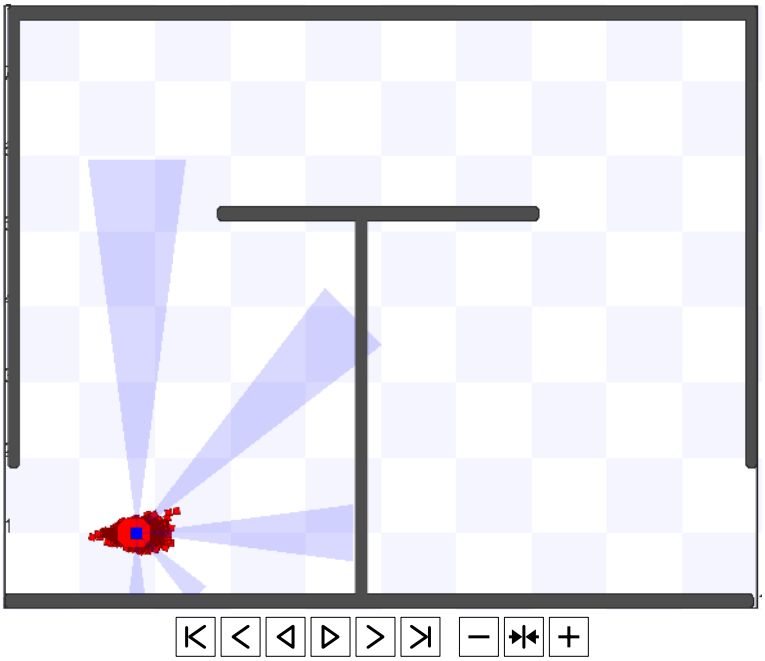


Figura 9 – Animação do ambiente de simulação reproduzido no STAGE

acontecia no MATLAB. Um exemplo do ambiente de navegação do MATLAB (figura 6), reproduzido no STAGE, pode ser visto na animação da figura 9.

Os outros arquivos, que ditam o comportamento dos atores no mundo, são escritos na linguagem C++. Nestes arquivos, foram implementados o algoritmo do filtro e o comportamento de movimentação do robô no ambiente de navegação. Uma versão resumida do código principal de implementação do filtro, que encontra-se no arquivo que dita o comportamento do robô, pode ser visto no código 6.

```
for (i=0; i<robot->number_particles; i++){
  // Get particle name
  sprintf(particle_name, "%s_p%03d", robot->robot_name.c_str(), i);
  p = pos->GetWorld()->GetModel(particle_name);

  // Particle pose estimation (returns bar_p)
  Pose bar_p;
  Pose particle_pose = p->GetGlobalPose();
  odom_estimation(&particle_pose, &bar_p, displacement, rot);
  // Particle pose RTR variation (returns pose)
  odom_variation(&particle_pose, &bar_p, alpha);
  // Set particle pose
  p->SetGlobalPose(particle_pose);

  // Check if outside map
  if ((p->GetPose().x >= (robot->floorplan_size_x)) || (p->GetPose().x < 0) ||
    (p->GetPose().y >= (robot->floorplan_size_y)) || (p->GetPose().y <= 0))
    p->SetGlobalPose(robot_mod->GetGlobalPose());
  else{
    // Raytracing for each sensor
    uint8_t sample_count = rob_laser->GetSensors().size();
    const double sample_incr = dtor(45.0); // 45 deg (dtor() to radians) increment
    const double start_angle = dtor(-90.0);// -90 deg (dtor() radians) as starting angle
    particle_pose.a += start_angle;
```

```
25      // Assuming all sensors have same range
26      double max_range = rob_laser->GetSensors()[0].range.max;
27      Ray particle_ray(rob_laser,particle_pose,max_range,ranger_match,NULL,true);
28
29      for(int n = 0; n<sample_count; n++)
30      {
31        const RaytraceResult rayresult = rob_laser->GetWorld()->Raytrace(particle_ray);
32        meters_t laser_particle = rayresult.range;
33        double laser_robot = rob_laser->GetSensors()[n].ranges[0];
34        particle_ray.origin.a += sample_incr;
35        // Calculate weight
36        weights[i] *= exp(-((laser_robot - laser_particle)*(laser_robot - laser_particle))
      / (2*(CovSonars*CovSonars))) / sqrt(2*M_PI*(CovSonars*CovSonars));
37        total_weight += weights[i];
38      }
39    }
40
41  }
42
43  // Normalize weights
44  for (i=0; i<robot->number_particles; i++){
45    weights[i] /= total_weight;
46  }
47
48  //Resampling
49  if((k % 15) == 0){
50    vector<Pose> p_vect;
51    p_vect.resize(robot->number_particles);
52    double mw, beta = 0;
53    int index, i;
54    char particle_name[50];
55    vector<double>::iterator point;
56    point = max_element(weights.begin(), weights.end());
57    mw = *point;
58    index = rand() % robot->number_particles; //Random from 0 to number_particles - 1
59
60    for(i = 0; i<robot->number_particles; i++){
61      beta += r2()*mw*2;
62      while(beta > weights[index]){
63        beta -= weights[index];
64        index++;
65        if(index > robot->number_particles-1){
66          index = 0;
67        }
68      }
69      sprintf(particle_name, "%s_p%03d", robot->robot_name.c_str(), index);
70      Pose part_pose = pos->GetWorld()->GetModel(particle_name)->GetGlobalPose();
71      p_vect[i] = part_pose;
72    }
73
74    //Rearrange particles
75    for(i = 0; i<robot->number_particles; i++){
76      sprintf(particle_name, "%s_p%03d", robot->robot_name.c_str(), i);
77      pos->GetWorld()->GetModel(particle_name)->SetGlobalPose(p_vect[i]);
78      weights[i] = 1.0;
79    }
80  }
```

Código 6 – Código resumido do algoritmo do Filtro de Partículas no STAGE

Neste código, pode-se ver claramente as etapas do filtro, sendo a etapa de predição realizadas nas linhas 6 a 13, a etapa de atualização nas linhas 15 a 39 e a etapa de reamostragem nas linhas 48 a 80. As etapas foram fielmente transcritas tal como elas são na implementação em MATLAB, com algumas peculiaridades que são exclusivas ao STAGE, como a linha 4 por exemplo, que recupera o modelo da partícula do mundo simulado pelo STAGE. Esta partícula tem diversas propriedades, sendo a mais interessante a sua pose (posição 2d + orientação angular), que é recuperada na linha 8. Uma grande facilidade deste simulador é que não é preciso modelar os sensores, sendo que o próprio STAGE já nos proporciona os dados dos robôs e sensores simulados. O código completo pode ser encontrado no repositório do projeto, no arquivo *random_particles.cc* da pasta */stage/examples/ctrl*.

O arquivo de mundo, por sua vez, determina os parâmetros do ambiente de simulação, como qual imagem (em escala de cinza) será utilizada para construir o ambiente, os atores que popularão o mundo (e.g. pessoas, robôs), suas geometrias e comportamentos, entre outros. O código 7 apresenta, também de forma reduzida, o arquivo de mundo utilizado com a implementação do filtro. O código completo, novamente, encontra-se no repositório do projeto, na pasta */stage/worlds* sob o nome de *random_particles.world*.

```
1 # ...
2
3 # load an environment bitmap
4 floorplan
5 (
6   name "cave"
7   size [10.000 8.000 0.800]
8   pose [5 4 0 0]
9   bitmap "bitmaps/salton_world.png"
10 )
11
12 # the robot model
13 saltonBot
14 (
15   # can refer to the robot by this name
16   name "r0"
17   pose [ 1 1 0 0 ]
18
19   # pioneer2dx's sonars will be ranger:0 and the laser will be ranger:1
20   saltonLaser( pose [ 0 0 0 0 ] )
21
22   # control file
23   ctrl "wander_drive"
24
25   # report error-ful position in world coordinates
26   localization "odom"
27   localization_origin [ 0 0 0 0 ]
28 )
29
30 # dummy model used only to create the particles with 'random_particles'
31 define puck model(
32   size [ 0.080 0.080 0.100 ]
33   gui_move 1    # the model can be moved by the mouse in the GUI window
```

```
34   gui_nose 1    # draw a nose on the model showing its heading (positive X axis)
35   gui_outline 0        # draw a bounding box around the model, indicating its size
36   obstacle_return 0   # this model can collide with other models that have this property
       set
37   ranger_return -1     # If negative, this model is invisible to ranger sensors, and does
       not block propogation of range-sensing rays.
38   fiducial_return 0
39   gripper_return 0
40   laser_return 0
41   blob_return 0
42 )
43
44 # the dummy model used to generate particles for a robot
45 puck
46 (
47   block(
48   points 3
49     point[0] [0 0]
50     point[1] [0 1]
51     point[2] [2 0.5]
52     z [0 0.1]
53   )
54   name "r0_dummy"
55   pose [-3 -6 0 0 ]
56   #particle color
57   color "red"
58
59   ctrl "random_particles 1000 r0 cave"
60 )
```

Código 7 – Arquivo de mundo resumido

As linhas 3 a 10, no código 7, realizam a parmetrização do *floorplan*, que é o ambiente onde irá trafegar o robô. Escolhe-se nome, tamanho, orientação e a imagem no qual o ambiente será baseado. A seguir, nas linhas 13 a 28, define-se o robô, ditando seu nome, pose inicial, comportamento (linha 23) e qual o tipo de modelo de movimentação ele utilizará (linha 26). Este no caso é odometria, por ser um modelo ruidoso no STAGE, sendo similar ao modelo em MATLAB e à vida real. Na linha 20, porém, há a descrição do sensor de distância, aqui descrito como uma generalização do sensor de distância a laser SICK[1]. A seguir, tem-se a definição do modelo base de onde serão criadas as partículas, nas linhas 31 a 42. Sua geometria é definida na função *puck*, nas linhas 45 a 60, que entre outros define os parâmetros de execução do mundo em sí (número de partículas, nome do robô e floorplan a ser carregado), na linha 59.

Porém, a curva de aprendizado do STAGE pode vir a ser um tanto elevada. Por sorte, há diversos exemplos de simples aplicações e instruções de uso no repositório do simulador[2]. É incluído como anexo neste relatório um passo-a-passo de compilação e instalação do STAGE.

[1] `https://www.sick.com/de/en/detection-and-ranging-solutions/2d-laser-scanners/tim3xx/c/g205751`

[2] `https://github.com/rtv/Stage`

3 Resultados

3.1 Estudos iniciais

Inicialmente, foi realizado um estudo detalhado da implementação original do professor Salton, para que fosse possível entender seu funcionamento e assim determinar quais eram os pontos críticos em tempo computacional da implementação. Em outras palavras, era de interesse descobrir quais funções ocupavam uma maior porcentagem do tempo de execução total. Para tanto, foi utilizada a ferramenta de *profiling* do próprio MATLAB, que mede o tempo de execução total e individual de cada função. Na figura 10 pode-se ver um exemplo de execução do código com esta ferramenta, de forma reduzida. Analisando os tempos, encontrou-se que o ponto crítico da implementação é a função de *Ray Casting*, que ocupa 42,99 segundos dos 51,63 segundos de execução total do algoritmo (83,26 %). Logo em seguida segue a função *ode23*, que é responsável pela simulação do robô diferencial, ocupando 4,12 % do tempo de execução. Os 12,62 % restantes são compostos de todas as outras funções utilizadas, algumas referentes à implementação em sí (e.g. *F_measurProb*, *F_sample_odometry*) e outras referentes a funções internas do MATLAB (e.g. *imshow*, *movegui*).

Function Name	Calls	**Total Time**
main	1	51.632 s
Fast_ray_cast	27540	42.997 s
ode23	540	2.137 s
F_measurProb	135000	1.317 s
F_anima_map	1	0.624 s
imshow	1	0.537 s
F_sample_odometry	27000	0.527 s
initSize	1	0.429 s
movegui	1	0.410 s

Figura 10 – Tempos de execução do código em MATLAB

Assim, fica evidente que a otimização toma prioridade no algoritmo de *Ray Casting*. Após estudos e experimentações deste algoritmo com diferentes parâmetros de Filtro, descobriu-se que as duas variáveis mais influentes em seu tempo computacional são o número de partículas do Filtro e o número de sensores que o *Ray Casting* precisa simular para cada partícula. Desta forma, é possível estabelecer uma métrica de carga computa-

cional Q, cuja unidade é a quantidade de vezes que o algoritmo de *Ray Casting* precisará ser executado. Definindo-se que o número de partículas é N e o número de sensores, que neste caso é constante, é $M = 5$, tem-se a relação $Q = N \times M = 5 \times N$. Evidentemente, as variáveis N e M possuem fatores de impacto diferentes. Tomando como exemplo os resultados da figura 10, que foram obtidos com $N = 50$, tem-se que o número de execuções fica em $Q = 5 \times 50 = 250$ execuções para apenas uma iteração do Filtro. Se N aumentar em 5 unidades, tem-se $Q = 5 \times (50 + 5) = 275$ execuções. Porém, se o parâmetro M aumenta em 5 unidades, tem-se $Q = (5 + 5) \times 50 = 500$ execuções, que é quase o dobro. Isso torna-se evidente quando percebe-se que os sensores tem de ser simulados para cada partícula individualmente. Desta forma, conclui-se que este parâmetro é o mais crítico deste algoritmo.

Vale lembrar que o parâmetro Q e o número de chamadas (*Calls*) mostrado na figura 10 são diferentes, sendo o parâmetro Q representativo do número de execuções do algoritmo que encontra-se implementado na função *Fast_ray_cast*, enquanto que o número de chamadas corresponde à quantidade de vezes que a função em sua totalidade foi chamada. É possível chegar no número de chamadas da seguinte maneira, sendo o número de iterações $k = 540$ e considerando a execução do *Ray Casting* para o robô, uma vez por iteração: $Calls = k \times N + k = 540 \times 50 + 540 = 27540$ chamadas.

3.2 Ganhos de desempenho

A seguir, foram discutidas maneiras de reduzir a carga computacional do algoritmo do Filtro. Uma solução óbvia é reduzir o parâmetro cujo impacto é maior na carga computacional, no caso o número de sensores M. Porém, o algoritmo já conta com um número baixo de sensores, sendo que a redução deste parâmetro comprometeria a precisão do Filtro. Outra alternativa é a redução do número de partículas N, que porém já estão em número bem reduzido.

É possível também substituir o algoritmo pelo qual os sensores são simulados, ou seja, trocando o *Ray Casting* por outro método de atualização, onde foram encontrados bons resultados utilizando a técnica dos campos de probabilidade (*likelihood field model*, [9]), implementada pelo professor Salton. Porém, posteriormente à implementação de outras técnicas de atualização, resolveu-se explorar uma outra maneira possível para redução da carga computacional: Reduzir o tempo de execução das funções em sí.

Desta forma, concluiu-se que era necessário uma mudança de ambiente de implementação. Como já discutido na seção 2.2, o MEX foi um passo intermediário para a implementação do código na linguagem C pura que por si só já mostrou uma grande redução em carga computacional, mesmo sendo o código do Filtro fielmente transposto a MEX. Na figura 11 pode-se ver que o tempo de execução do Filtro em sua totalidade (in-

cluindo o algoritmo de *Ray Casting*), na função *mexFP*, é muito menor do que a execução somente da função de *Ray Casting* na implementação original, e ocupando somente 2,97 % do tempo de execução total do código. Comparando os tempos de execução das funções *Fast_ray_cast* na implementação original e *mexFP* na implementação MEX, tem-se que a segunda função é aproximadamente 208 vezes mais rápida. Pode-se, enfim, comparar as duas implementações com diferentes valores para o número de partículas, que é vista na figura 12.

Function Name	Calls	**Total Time**
main_mex	1	6.957 s
ode23	540	2.121 s
Fast_ray_cast	540	1.216 s
F_anima_map	1	0.491 s
newplot	565	0.427 s
hold	541	0.375 s
funfun/private/odearguments	540	0.336 s
F_dif_drive_car	16767	0.272 s
odeget	5940	0.272 s
imshow	1	0.259 s
close	1	0.207 s
F_encoders	540	0.207 s
mexFP (MEX-file)	540	0.207 s
odeget>getknownfield	5940	0.168 s

Figura 11 – Tempos de execução do código com o Filtro em MEX

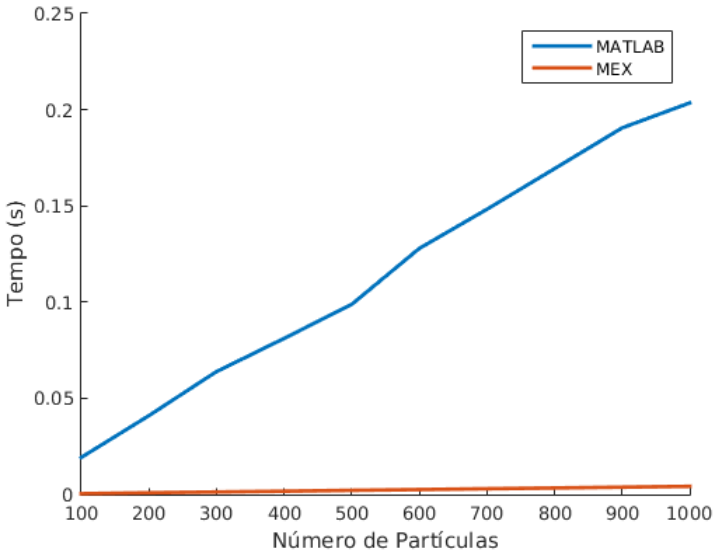


Figura 12 – Gráfico comparativo dos tempos de execução do código do Filtro entre MATLAB e MEX

A respeito da implementação no simulador STAGE, notou-se um desempenho substancialmente inferior em relação ao MEX. Utilizando contadores de tempo no código

principal do comportamento do robô no STAGE (código **XX**), pode-se ver que o Filtro nesta implementação possui tempos de execução de em média entre 0,012 e 0,014 segundos, o que é aproximadamente 5 vezes maior que os 0,0022 segundos médios de execução do Filtro em MEX. A figura 13 mostra um gráfico comparativo entre os tempos de execução do MEX e STAGE.

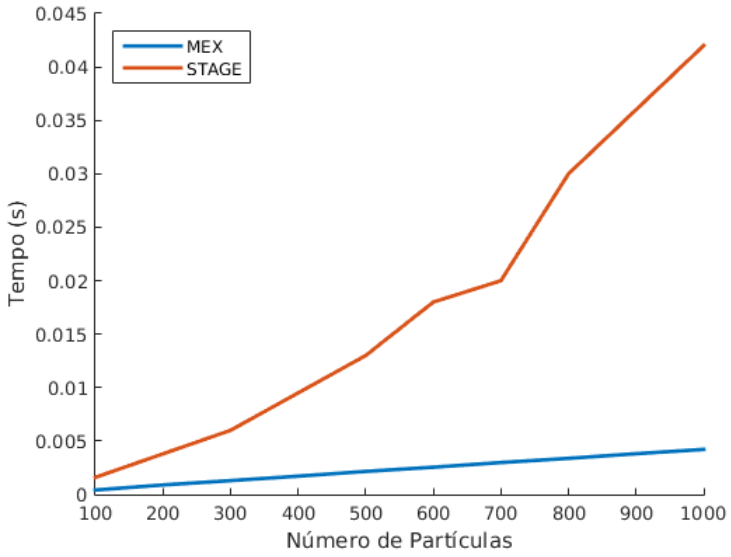


Figura 13 – Gráfico comparativo dos tempos de execução do código do Filtro entre MEX e STAGE

3.3 Comparação de desempenho com o AMCL

O *AMCL*[1] é o módulo do framework ROS [13] que executa o algoritmo de filtro de partículas. O AMCL representa o estado da arte em termos da implementação do algoritmo e, portanto, é utilizado para fins de comparação a implementação desenvolvida neste projeto.

Assim, decidiu-se comparar o desempenho da implementação em STAGE com este módulo. Para uma melhor comparação, foi utilizado o pacote do ROS chamado *navigation_stage*[2], que implementa o algoritmo do AMCL utilizando algumas funções existentes no STAGE (e.g. *Raytrace*). Este pacote também implementa o módulo de controle robótico *move_base* e o ambiente de simulação *RVIZ*, sendo que assim fica delegado ao STAGE somente a simulação dos sensores. Na figura 14, pode-se ver o mapa de um ambiente simulado, no *RVIZ*, onde a nuvem de pontos vermelhos representa a nuvem de partículas.

A ferramenta de análise de utilização de processamento *sysprof*[3] foi utilizada, por ter algumas vantagens em relação a outras aplicações semelhantes. Por exemplo, outra

[1] `http://wiki.ros.org/amcl`
[2] `http://wiki.ros.org/navigation_stage`
[3] `http://sysprof.com/`

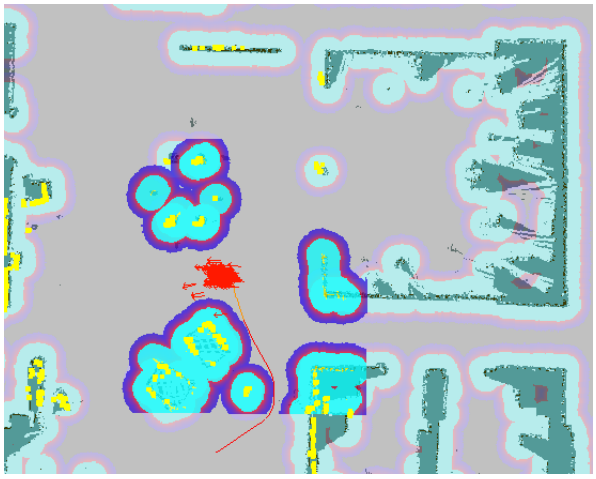


Figura 14 – Ambiente de simulação RVIZ

ferramenta popular é o framework *Valgrind*[4], sendo possível rodar o nodo AMCL diretamente com este profiler. A sua desvantagem frente ao *sysprof*, porém, é que o nodo desta forma é executado em um ambiente virtual, utilizando uma única *thread*, o que pode vir a mascarar o desempenho real do AMCL se este utilizar várias *threads*. O *sysprof* analisa todas as funções ativas no sistema em dado momento, assim contornando este problema. Ambas as implementações foram executadas utilizando parâmetros equivalentes, com 1000 partículas. Vale lembrar, porém, que os dados de tempos obtidos com o *sysprof* não são constantes, visto que dependem do que o processador pode alocar no momento do profiling para cada função. Ele é utilizado apenas para ter uma base do desempenho das implementações AMCL e STAGE.

Na figura 15a pode-se ver, primeiramente, um apanhado geral das funções e suas porcentagens de utilização com a implementação STAGE rodando. A figura mostra a função *[stage]* ocupando 38,7 % do tempo de utilização do processador, que é uma porcentagem bem alta. Também é possível ver as funções do STAGE *ModelRanger*, mais detalhada na figura 15b, e *Raytrace*. Como percebe-se pelas figuras, a função de *Raytrace* aparece duplamente, uma vez fora e uma vez dentro da função *ModelRanger*. Há ainda uma diferença gritante de utilização entre elas, 11 e 0,06 % respectivamente, que se dá porque a função *Raytrace* dentro da *ModelRanger* é responsável pela simulação dos sensores de distância do robô, enquanto que a outra é responsável pela simulação dos sensores de distância das partículas.

Já para a implementação em ROS, dada a natureza do framework de separar suas execuções em várias funções, considerou-se impraticável compor uma figura tal como a 15a. Ainda assim, foram encontradas as funções relevantes ao nodo AMCL, como se por ver na figura 16a. Nesta figura, tem-se evidência da técnica dos campos de probabilidade, utilizada na etapa de atualização, que é mais eficiente do que a técnica de *Raytrace*

[4] `http://valgrind.org/`

Functions	Self	Total ▾
[Everything]	0,00 %	100,00 %
In file [heap]	0,00 %	65,82 %
[/usr/bin/perl]	0,00 %	50,35 %
No map [/usr/bin/perl]	0,00 %	43,12 %
[stage]	0,00 %	38,47 %
No map [stage]	0,00 %	24,47 %
In file /lib/x86_64-linux-gnu/libgli…	22,35 %	22,35 %
Stg::ModelRanger::~ModelRanger()	0,00 %	17,12 %
In file /usr/local/lib64/libstage.so…	0,00 %	16,18 %
Stg::World::Raytrace(Stg::Ray co…	11,00 %	11,00 %

(a)

Descendants	Self	Cumulative ▾
▾ [stage]	0,00 %	38,47 %
▾ No map [stage]	0,00 %	24,35 %
▾ Stg::ModelRanger::~ModelRanger()	0,00 %	17,12 %
▾ In file [heap]	0,00 %	17,12 %
▸ In file /usr/local/lib64/libstage.so.4.1.1	0,00 %	16,12 %
Stg::Model::IsDescendent(Stg::Mode…	0,71 %	0,71 %
PositionUpdate(Stg::ModelPosition*…	0,12 %	0,12 %
exp	0,06 %	0,06 %
Stg::World::Raytrace(Stg::Ray const&)	0,06 %	0,06 %
__ieee754_exp_avx	0,06 %	0,06 %

(b)

Figura 15 – Profiling da implementação STAGE

utilizada na implementação STAGE. Também encontrou-se a função *ModelRanger*, que realiza a simulação dos sensores de distância do robô (figura 16b). Finalmente, com relação às porcentagens de utilização, percebe-se que o AMCL é de fato melhor, com uma utilização de apenas 1,35 % em sua totalidade. A função *Raytrace* aqui também possui um desempenho melhor, utilizando apenas 1,57 % do processamento.

Descendants	Self	Cumulative ▾
▾ No map [/opt/ros/indigo/lib/amcl/amcl]	0,00 %	1,35 %
▸ pthread_cond_timedwait@@GLIBC_2.3.2	0,03 %	0,46 %
amcl::AMCLLaser::LikelihoodFieldModel(amc…	0,11 %	0,11 %
In file /lib/x86_64-linux-gnu/libc-2.19.so	0,09 %	0,09 %
▸ bool boost::condition_variable::timed_wait<…	0,04 %	0,07 %

(a)

Descendants	Self	Cumulative ▾
▾ Stg::ModelRanger::~ModelRanger()	0,00 %	1,67 %
▾ No map [/opt/ros/indigo/lib/stage_ros/stageros]	0,00 %	1,67 %
▾ In file /opt/ros/indigo/lib/stage_ros/stageros	0,00 %	1,67 %
Stg::World::Raytrace(Stg::Ray const&)	1,57 %	1,57 %
std::_Rb_tree<Stg::point_int_t, std::pair<Stg:…	0,07 %	0,07 %
Stg::Color::Color()	0,03 %	0,03 %

(b)

Figura 16 – Profiling do AMCL com o pacote *navigation_stage* do ROS

4 Conclusões

Este projeto apresentou o desenvolvimento e os resultados obtidos no sentido de otimizar a execução do algoritmo de filtro de partículas. Três novas versões do filtro foram desenvolvidas, cada uma com suas vantagens e desvantagens. A versão MATLAB+MEX é muito similar à versão original, porém possui um desempenho muito melhor. Entretanto ele ainda está atrelado ao MATLAB, o que pode limitar o seu uso. A versão C executa fora do ambiente do MATLAB, mas ainda precisa que o MATLAB esteja instalado no computador para fins de visualização dos resultados obtidos. Apesar do seu desempenho não ter sido avaliado, espera-se que seja ainda melhor que a versão MATLAB-MEX. Por fim, as versão STAGE está totalmente desvinculada do MATLAB, é baseada somente em código aberto, e possui uma versatilidade muito melhor que o ambiente de simulação original baseado em MATLAB, embora seu desempenho seja menor do que a implementação em MEX. Foi também realizada uma comparação de utilização de processamento entre a implementação em STAGE e o nodo AMCL do framework ROS, que é considerada uma implementação estado da arte. Novamente, a implementação em STAGE provou-se menos eficiente, ocupando porcentagens substancialmente maiores em relação ao AMCL. Possivelmente isto seja porque o AMCL utiliza a técnica de campos de probabilidade para a etapa de atualização do Filtro, o que reduz bastante o tempo computacional desta etapa, ou também porque o algoritmo de *Raytrace* seja implementado de uma maneira otimizada. As seções a seguir discutem caminhos para a continuação deste projeto.

4.1 Discussões

4.1.1 Sobre a execução do algoritmo em FPGAs

O projeto original mencionava a possibilidade de explorar o potencial paralelismo do filtro de partículas utilizando FPGAs [1]. Entretanto, ao estudar melhor o algoritmo, percebeu-se que o mesmo utiliza intensivamente de operações de ponto flutuante e operações transcendentais. A implementação destas operações em FPGA ocuparia uma grande área de lógica reconfigurável, limitando o número de operações que o FPGA suportaria em paralelo, por fim, limitando o potencial de paralelismo do algoritmo. Outra desvantagem desta abordagem é que o projeto da lógica do filtro em VHDL seria complexa, exigindo grande esforço em codificação VHDL e verificação da lógica. Estimamos que o esforço necessário seria maior que o possível considerando o número de alunos disponíveis

[1] Field-programmable gate array `https://en.wikipedia.org/wiki/Field-programmable_gate_array`

no projeto e o tempo disposto para tal desenvolvimento. Desta forma, a opção de projetar parte do filtro em FPGA foi descartada logo no início do projeto.

4.1.2 Sobre a execução do algoritmo em GPUs

Originalmente no projeto submetido havíamos pensado em ganhar desempenho explorando o possível paralelismo do filtro, uma vez que ele repete as mesmas operações para cada partícula. Intuitivamente isto parecia factível, porém, ao estudar o algoritmo mais detalhadamente percebeu-se alguns complicações para a sua implementação em arquiteturas paralelas.

Primeiro, a maioria das operações são em ponto flutuante. Por outro lado as GPUs atuais possuem número limitado de unidades de ponto flutuante, reduzindo o potencial de paralelismo. Por exemplo, a GPU NVIDIA Quadro 4000[2] disponível no laboratório suporta somente $4,86 \times 10^{11}$ operações de ponto flutuante em paralelo (single-precision). No segundo semestre de 2015 recebemos uma outra GPU com mais recursos, modelo NVIDIA Tesla K40[3]. Esta GPU suporta $4,29 \times 10^{12}$ operações de ponto flutuante em paralelo (também single-precision), entretanto o seu custo está estimado em US$ 3,115.42[4], limitando a utilização da versão otimizada do filtro.

Segundo, as GPUs possuem quantidade de memória interna limitada. Quando este limite é ultrapassado, a GPU começa a utilizar a memória externa, que é muito mais lenta e possui o gargalo de largura de banda da interface externa da GPU. O principal problema do filtro é que o cálculo de cada partícula efetua leituras no mapa, que pode ocupar uma grande quantidade de memória. No caso de cálculo das partículas em paralelo, isto significa que haveriam várias leituras simultâneas para a memória que representa o mapa. Geralmente a quantidade de memória utilizada pelo mapa ultrapassa a quantidade de memória interna da GPU, fazendo com que a mesma acesse a memória externa que é lenta e com acesso congestionado devida à limitação da largura de banda da interface.

Algumas alternativas para este segundo problema foram pensadas, porém não houve tempo de testá-las, ficando a ideia para trabalhos futuros. Como mencionado anterioremente, foi desenvolvida uma representação compacta do mapa, que ocupa 64 vezes menos memória que a representação original. Isto ajudaria a aliviar o problema, entretanto, à medida que o mapa cresça, eventualmente ele ainda não vai caber na memória interna da GPU, causando perda de desempenho. Outra alternativa promissora é o uso das memórias de constantes e de texturas [5] existentes nas GPUs CUDA. Estas memórias

[2] `http://www.nvidia.com/object/product-quadro-4000-us.html`
[3] `https://www.nvidia.com/content/tesla/pdf/nvidia-tesla-k40-2014mar-lr.pdf`
[4] `http://www.amazon.com/NVIDIA-Tesla-Graphic-Card-900-22081-2250-000/dp/B00KDRRTB8`
[5] Texture Memory in CUDA | What is Texture Memory in CUDA programming `http://cuda-programming.blogspot.com.br/2013/02/texture-memory-in-cuda-what-is-texture.html`

possuem hierarquia de memória e são usadas em modo somente de leitura. Estas memórias são adequadas pois o mapa é somente lido duranto do cálculo das partículas.

Outra alternativa para o segundo problema, relacionado ao mapa do ambiente, seria mudar o algoritmo do programa para ele não acessar o mapa inteiro, mas sim seções do mapa. Isto permitiria colocar somente um pedaço do mapa na memória e calcular somente as partículas que estão próximas da área representada pelo mapa. Esta alternativa tem o melhor potencial de ganhos de desempenho, porém é mais complexa e exigiria mais estudos de formas alternativas de representações de mapas em memória.

4.2 Trabalhos Futuros

4.2.1 Representar o mapa em memórias read-only da GPU

Como mencionado antes, o uso das memórias de constantes e de texturas pode ser uma solução para colocar o mapa do ambiente nas memórias internas da GPU, potencialmente levando a um grande aumento de desempenho.

4.2.2 Representações mais eficientes do mapa em memórias

Apesar das otimizações obtidas com a representação do mapa, acreditamos que será necessário otimizar ainda mais a utilização de memória. Será necessário fazer uma revisão das possíveis representações. Por exemplo, a memória do mapa possui uma característica esparsa, onde poucas posições da memória são utilizadas, havendo desperdícios. Pode-se tentar tirar proveito desta característica para tornar a representação ainda mais eficiente e compacta. Outra possibilidade é fazer a carga em memória de seções do mapa, ao invés do mapa completo.

4.2.3 Artigo sobre uso do STAGE para fins educacionais

O uso de Stage como ferramenta de simulação do filtro mostrou-se muito promissora e versátil, permitindo facilmente mudar o ambiente simulado, suportando múltiplos robôs, possibilitando modelar facilmente os recursos típicos de um robô. Além disso o seu desempenho foi satisfatório, um pouco mais lento que a versão do filtro sozinho, sem recursos de simulação. Considerando estas vantagens, planejamos no primeiro semestre de 2016 escrever um artigo sobre o uso da implementação do filtro com STAGE para fins educacionais, explicando o funcionamento do mesmo e como as varíaveis do filtro influenciam o seu comportamento.

4.2.4 Continuar a implementaçaão da versão C

A versão C do filtro ainda tem uma dependência com o MATLAB. No futuro esta dependência pode ser eliminada e criado um pacote para facilitar a distribuição aberta do software.

4.2.5 Estudar o método de Likehood Field

O estudo do AMCL mostrou que ele utiliza um método alternativo ao Raytrace, chamado Likehood Field, que é computacionalmente mais eficiente. Seria interessante estudar mais este método para integrá-lo à implementação proposta do filtro. O uso deste método possivelmente faria com que o desempenho da implementação proposta ficasse similar ao desempenho do AMCL.

4.2.6 Integrar a versão otimizada do filtro ao ROS

Estima-se que o versão desenvolvida do filtro tenha desempenho similar ao AMCL, hoje disponível no ROS. Porém, com algumas otimizações que poderiam ser realizadas, espera-se que a nova versão seja mais rápida que o AMCL. Se isto se confirmar, seria interessante montar um novo pacote para o ROS, que substitua o AMCL atual.

Referências

1 NEHMZOW, U. *Mobile Robotics: A Practical Introduction*. [S.l.]: Springer, 2003. 280 p. Citado 2 vezes nas páginas 7 e 9.

2 SIEGWART, R.; NOURBAKHSH, I. R. *Introduction to Autonomous Mobile Robots*. [S.l.]: MIT Press, 2004. 321 p. Citado 3 vezes nas páginas 7, 9 e 11.

3 DUDEK, G.; JENKIN, M. Computational principles of mobile robotics. Cambridge University Press, p. 406, 2010. Citado 2 vezes nas páginas 7 e 9.

4 SCARAMUZZA, D.; FRAUNDORFER, F. Visual odometry: Part i - the first 30 years and fundamentals. *IEEE Robotics and Automation Magazine*, v. 18, n. 4, 2011. Citado na página 7.

5 REKLEITIS, I. M. *A Particle Filter Tutorial for Mobile Robot Localization*. [S.l.]. Citado 4 vezes nas páginas 7, 9, 13 e 16.

6 CANDY, J. V. *Bayesian Signal Processing: Classical, Modern and Particle Filtering Methods*. [S.l.]: John Wiley and Sons, 2011. Citado na página 9.

7 HAUG, A. J. *Bayesian Estimation and Tracking: A Practical Guide*. [S.l.]: John Wiley & Sons, 2012. 448 p. Citado na página 9.

8 PEREIRA, A. S. *Navegação de Veículo Diferencial Empregando Robot Operating System (ROS)*. 27 p. Trabalho de Conclusão de Curso — Faculdade de Engenharia, Pontifícia Universidade Católica do Rio Grande do Sul, Porto Alegre, 2014. Disponível em: <http://www.inf.pucrs.br/alexandre.amory/research/bib/docs-/2014_TCC_Alexandre_Pereira.pdf>. Citado na página 11.

9 THRUN, S.; BURGARD, W.; FOX, D. *Probabilistic Robotics*. [S.l.]: MIT Press, 2005. 647 p. Citado 2 vezes nas páginas 13 e 27.

10 VANDEVENNE, L. *Raycasting*. 2007. Disponível em: <http://lodev.org/cgtutor-/raycasting.html>. Citado na página 15.

11 Udacity. *Resampling Wheel - Artificial Intelligence for Robotics*. 2012. Disponível em: <https://www.youtube.com/watch?v=wNQVo6uOgYA>. Citado na página 17.

12 VAUGHAN, R. Massively multi-robot simulation in stage. *Swarm Intelligence*, Springer, v. 2, n. 2-4, p. 189–208, 2008. Citado na página 21.

13 MARTINEZ, A.; FERNáNDEZ, E. *Learning ROS for Robotics Programming*. [S.l.]: Packt Publishing, 2013. 332 p. Citado na página 29.

Anexos

ANEXO A – Configuração e compilação de arquivos MEX

1. Para configurar o compilador C-MEX no MATLAB, rodar o comando **mex -setup**. Como configuração padrão, o MATLAB usará o compilador padrão da linguagem C do sistema operacional (e.g. gcc em Linux).

2. O framework MEX atualmente suporta as linguagens C, C++ e FORTRAN. Para mudar de linguagem, rodar o comando **mex -setup <linguagem>**.

3. Para compilar um programa em MEX, rodar o comando **mex <programa>**.

ANEXO B – Compilação e instalação do STAGE

O procedimento apresentado foi testado no Ubuntu 14.04.

1. Fazer download do Stage do repositório principal (`https://github.com/rtv/Stage`)
2. Abrir o arquivo CMakeLists.txt, na pasta **/Stage/examples/ctrl**, e adicionar o arquivo **random_particles.cc** à lista **SET(PLUGINS). Copiar o arquivo random_particles.cc para esta mesma pasta**.
3. Voltar à pasta **/Stage** e prosseguir com a configuração, compilação e instalação:
   a) Criar diretório build **mkdir build; cd build**.
   b) Rodar o comando **ccmake ..**
   c) Para confirmar a configuração, pressionar a tecla "c"seguido da tecla "g".
   d) Rodar o comando **cmake ..**
   e) Rodar o comando **make**.
   f) Rodar o comando **sudo make install**
4. Copiar o arquivo **random_particles.world** para a pasta **/Stage/worlds**.
5. Para executar o ambiente de simulação, ir até a pasta **/Stage/worlds** e rodar o comando **stage random_particles.world**.
6. Em caso de erro com a biblioteca **libstage.so.4.1.1**, adicionar o diretório desta biblioteca à variável de ambiente **LD_LIBRARY_PATH**. A biblioteca normalmente encontra-se na pasta **/usr/local/lib** ou **/usr/local/lib64**.

ANEXO C – Profiling no ROS e Stage

Para realizar o profiling do pacote **navigation_stage**, deve-se primeiro instalar o framework ROS, seguindo os passos do tutorial de instalação disponível em `http://wiki.ros.org/indigo/Installation/Ubuntu`. Este instala a versão *Indigo* do framework, que é a utilizada aqui. Ela foi testada com sucesso no sistema operacional Ubuntu 14.04. Para instalar o pacote **navigation_stage**, rodar o comando **sudo apt-get install ros-indigo-navigation-stage**. Para rodar o AMCL, utilizar o comando **roslaunch navigation_stage move_base_amcl_2.5cm.launch**.

Finalmente, para realizar o profiling, é necessária a ferramenta **sysprof**, que pode ser instalada com o comando **sudo apt-get install sysprof**. Para utilizar o profiler, certifique-se de que as partes a serem analisadas estão rodando e então execute o comando **sudo sysprof**. Este abrirá uma interface gráfica, onde deve-se pressionar **Start** e logo em seguida **Profile**. Irá ser populada a tabela **Functions**, onde é possível ver as funções em execução e suas porcentagens de utilização.