UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Daniel Terra Gomes

Trabalho Prático 3: Exploração e Mapeamento

Belo Horizonte, MG

10 de outubro de 2025

"Behind me lies a farm.
I wonder if there is bread above the hearth
and if I will ever return."
(Pantheon, League of Legends)

Lista de ilustrações

Lista de trechos de código

Sumário

1 Introdução

Este documento apresenta a solução desenvolvida para o Trabalho Prático 3 (TP3) da disciplina de Robótica Móvel, focado em exploração autônoma e mapeamento utilizando o algoritmo *Occupancy Grid*. O objetivo principal é implementar um sistema completo de mapeamento probabilístico que permita ao robô construir um mapa do ambiente desconhecido enquanto navega de forma autônoma.

1.1 Objetivos

Os objetivos deste trabalho prático são:

1. **Implementar o algoritmo Occupancy Grid**: algoritmo probabilístico de mapeamento que representa o ambiente como uma grade de células, onde cada célula possui uma probabilidade de ocupação. A implementação deve:
   - Utilizar representação em *log-odds* para eficiência computacional;
   - Aplicar o modelo inverso do sensor (inverse sensor model) para atualização bayesiana;
   - Tratar a discretização do espaço contínuo para a grade discreta.
2. **Desenvolver estratégia de navegação para exploração**: implementar uma estratégia simples e reativa de navegação que permita ao robô explorar o ambiente de forma autônoma. Neste trabalho, foi utilizada a estratégia de *wall-following* (seguimento de parede) baseada em algoritmos Bug (Macharet, 2025b).
3. **Avaliar impacto do tamanho de célula**: realizar experimentos variando o tamanho da célula da grade (0.01m, 0.1m e 0.5m).
4. **Testar em ambientes estático e dinâmico**: validar a robustez da solução em dois cenários distintos, garantindo que o sistema funcione corretamente mesmo na presença de obstáculos dinâmicos.
5. **Visualizar resultados incrementais**: gerar visualizações que mostrem a trajetória do robô, as leituras do sensor laser e o mapa de ocupação construído ao longo do tempo.

1.2 Contextualização do Problema

O mapeamento é uma das competências fundamentais para robôs verdadeiramente autônomos (Thrun; Burgard; Fox, 2005). A capacidade de construir mapas a partir de dados sensoriais permite que robôs, localizem-se no ambiente, planejem trajetórias e tomem decisões, conforme a hierárquica dos subsistemas de condução visto na Figura 1.

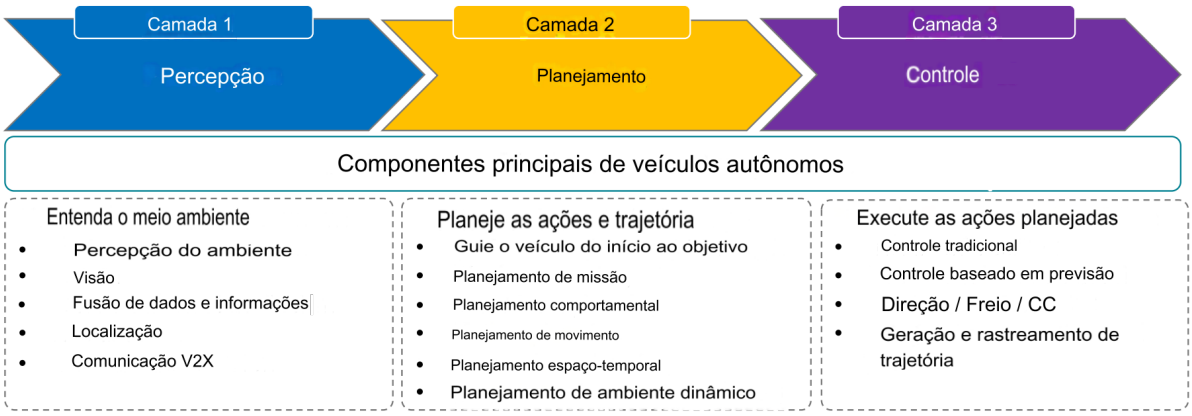

Figura 1 – Arquitetura hierárquica dos subsistemas de condução autônoma (Khan *et al.*, 2022, p. 4)

O problema de mapeamento apresenta diversos desafios:

- **Incerteza sensorial**: sensores laser, sonar e câmeras são inerentemente ruidosos e imprecisos;
- **Erros odométricos**: a localização do robô acumula erros ao longo do tempo devido a deslizamentos e imperfeições mecânicas;
- **Ambiguidade perceptual**: diferentes locais podem parecer similares aos sensores;
- **Fechamento de loops**: reconhecer quando o robô retorna a um local já visitado.

De modo a superar esses desafios a integração sensorial, conforme demonstrado nas Figura 2, é fundamental para viabilizar o desempenho satisfatório dos VA em uma ampla gama de cenários. Entretanto, aplicação dessas técnicas vão além dos requisitos deste trabalho.

Parâmetros	Camera	Radar	LIDAR	Ultra-sonic	Fusão
Campo de visão	3	4	3	2	4
Faixa	3	4	4	1	4
Resolução de velocidade	2	3	4	1	4
Resolução Angular	4	3	2	1	4
Tempo Adverso	2	2	3	2	4
Perturbação de escuridão/luz	3	4	4	4	4
Classificação de objetos	4	3	2	2	4
Detecção de todas as superfícies de objetos	3	3	3	3	4

Esquema de cores: 1 → Pior, 2 → Medíocre, 3 → Aceitável, 4 → Bom

Figura 2 – Matriz de adequação sensorial para diferentes condições operacionais (Khan *et al.*, 2022, p. 6)

Portanto, neste trabalho, simplificamos o problema assumindo que **a localização do robô é conhecida**, obtida diretamente da API do simulador CoppeliaSim (Coppelia Robotics, 2024). Esta suposição é equivalente à fase de pós-processamento de algoritmos SLAM (Simultaneous Localization and Mapping), onde a trajetória já foi estimada e o objetivo é construir um mapa consistente (Thrun; Burgard; Fox, 2005).

1.3 Ambiente de Desenvolvimento

O desenvolvimento foi realizado utilizando as seguintes ferramentas:

- **CoppeliaSim Edu V4.10.0**: simulador de robótica 3D utilizado para criar os cenários e simular o robô Kobuki (Coppelia Robotics, 2024);
- **Python 3.13 via Miniconda**: linguagem de programação principal;
- **Bibliotecas Python**:
  - `NumPy 2.1.3`: computação numérica e álgebra linear;
  - `Matplotlib 3.9.2`: visualização de gráficos e mapas;
  - `Pillow 11.0.0`: manipulação e salvamento de imagens;
  - `SciPy 1.14.1`: funções científicas (rotações, interpolação);
- **Jupyter Notebook via VS Code**: ambiente interativo para desenvolvimento e documentação;
- **ZMQ Remote API**: interface de comunicação entre Python e CoppeliaSim para controle do robô e leitura de sensores;
- **LaTeX**: preparação da documentação técnica.

1.4 Especificações do Robô Kobuki

O robô utilizado neste trabalho é o **Kobuki**, um robô móvel diferencial amplamente utilizado em pesquisa e educação em robótica. As especificações técnicas relevantes para este trabalho são (Yujin Robot, 2025):

- **Tipo**: Robô diferencial (differential drive);
- **Distância entre rodas** (*wheelbase*, L): 0.230 m;
- **Raio das rodas** (r): 0.035 m;
- **Diâmetro do corpo** (*footprint*): $\approx$ 0.30 m;
- **Raio efetivo** (para planejamento): $\approx$ 0.15 m;
- **Velocidade linear máxima**: 0.70 m/s;
- **Velocidade angular máxima**: 180°/s (π rad/s);
- **Sensor laser**: fastHokuyo, com 684 feixes cobrindo 180° (-90° a +90°), alcance máximo de 5.0 m.

O modelo cinemático do robô diferencial é dado por:

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} v\cos\theta \\ v\sin\theta \\ \omega \end{bmatrix} \tag{1.1}$$

Onde: v = velocidade linear do robô (m/s)
ω = velocidade angular do robô (rad/s)
θ = orientação do robô (rad)

1.5 Cenários de Teste

Foram utilizados dois cenários fornecidos pelo professor para validação da solução:

- **Cenário Estático** (`cena-tp3-estatico.ttt`): ambiente com obstáculos fixos (paredes, mobília), ideal para avaliar a qualidade básica do mapeamento, conforme visto na Figura 3;

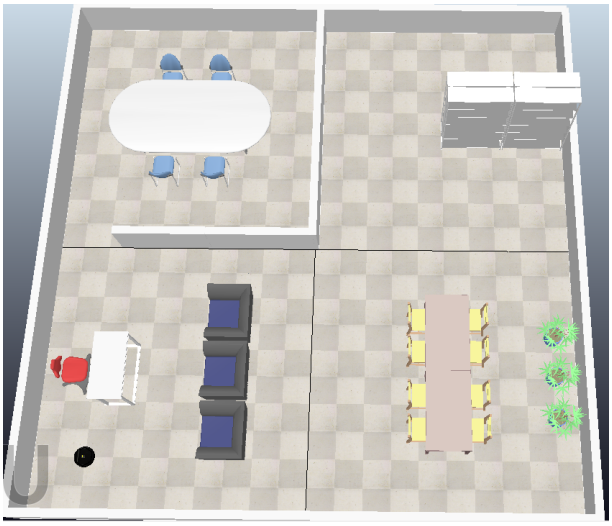

Figura 3 – Cenário Estático

- **Cenário Dinâmico** (`cena-tp3-dinamico.ttt`): ambiente com obstáculos móveis (pessoas, objetos), desafiando a capacidade do algoritmo de filtrar ruído temporário, conforme visto na Figura 4.

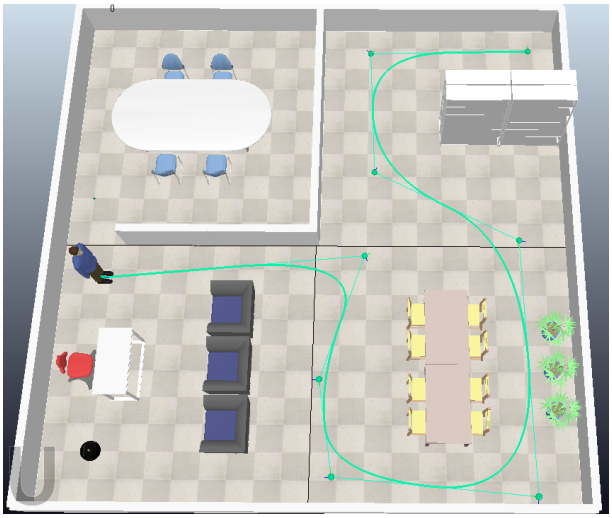

Figura 4 – Cenário Dinâmico

1.6 Estrutura Geral do Código

O projeto foi organizado de forma modular para facilitar a reutilização, manutenção e compreensão do código. A estrutura do diretório é apresentada a seguir na Lista de Código 1.1.

```
1 T3/
2 |-- dev/
3 |   |-- utils/
4 |   |   |-- kobuki_controller.py        # Controle Kobuki
```

```
|   |   |-- occupancy_grid_mapper.py   # Occupancy Grid
|   |   |-- wall_follower.py           # Navegação
|   |   |-- tp3_utils.py               # Utils
|   |   |-- visualization_utils.py     # Geração de gráficos
|   |
|   |-- TP3.ipynb                      # Notebook principal
|
|-- documentation/
|   |-- TP3.tex                        # Este documento (LaTeX)
|   |-- Figures/                       # Figuras e resultados
|
|-- cena-tp3-estatico.ttt              # Cenário estático
|-- cena-tp3-dinamico.ttt              # Cenário dinâmico
```

Lista de código 1.1 – Estrutura Geral do Código

Os módulos principais são:

- `kobuki_controller.py`: gerencia comunicação com CoppeliaSim e controle de baixo nível do robô;
- `occupancy_grid_mapper.py`: implementa o algoritmo Occupancy Grid com representação log-odds;
- `wall_follower.py`: estratégia reativa de navegação baseada em wall-following;
- `tp3_utils.py`: transformações de coordenadas, leitura de laser, etc.;
- `visualization_utils.py`: geração de plots incrementais e mapas finais.

2 Execução

2.1 Dependências

Para executar o código desenvolvido, é necessário instalar as seguintes dependências Python da Lista de Código 2.1:

```
# Via conda/miniconda:
conda create -n tp3 python=3.13
conda activate tp3

# Instalar bibliotecas:
pip install numpy==2.1.3
pip install matplotlib==3.9.2
pip install pillow==11.0.0
pip install scipy==1.14.1
pip install coppeliasim-zmqremoteapi-client
```

Lista de código 2.1 – Dependências

Além disso, é necessário ter o **CoppeliaSim Edu V4.10.0** instalado no sistema.

2.2 Como Executar

O procedimento de execução é o seguinte:

1. **Abrir o CoppeliaSim**: iniciar o simulador CoppeliaSim;
2. **Carregar cenário**: abrir uma das cenas fornecidas:
   - `cena-tp3-estatico.ttt` (ambiente estático)
   - `cena-tp3-dinamico.ttt` (ambiente dinâmico)
3. **Executar notebook**: abrir o arquivo `TP3.ipynb` no VS Code ou Jupyter Notebook;
4. **Configurar parâmetros**: no início do notebook, configurar:
   - `CELL_SIZE`: tamanho da célula (0.01, 0.1 ou 0.5)
   - `MAX_ITERATIONS`: número máximo de iterações de exploração
   - `SCENARIO`: 'static' ou 'dynamic'

5. **Executar células sequencialmente**: rodar todas as células do notebook na ordem. O código irá:
   - Conectar-se ao CoppeliaSim via ZMQ API;
   - Inicializar o mapeador e navegador;
   - Executar loop de exploração (leitura laser → navegação → atualização mapa);
   - Gerar visualizações incrementais;
   - Salvar mapa final como imagem.
6. **Analisar resultados**: os gráficos e mapas serão salvos na pasta `Figures/`.

2.3 Parâmetros de Configuração

Os principais parâmetros do sistema são:

- **Mapa**:
  - `MAP_SIZE`: dimensões do mapa em metros [largura, altura];
  - `MAP_ORIGIN`: coordenadas do canto inferior esquerdo [x, y];
  - `CELL_SIZE`: tamanho de cada célula (0.01, 0.1 ou 0.5 m).
- **Occupancy Grid**:
  - `L_OCC`: log-odds para célula ocupada ($\approx +2.197$);
  - `L_FREE`: log-odds para célula livre (≈ -2.197);
  - `HIT_RADIUS`: raio ao redor do ponto de impacto laser (0.2 m).
- **Navegação**:
  - `V_NOMINAL`: velocidade linear nominal (0.4 m/s);
  - `W_MAX`: velocidade angular máxima (45°/s);
  - `D_SAFE`: distância segura a obstáculos (0.4 m);
  - `D_FOLLOW`: distância desejada para seguir parede (0.6 m).

3 Navegação

A estratégia de navegação utilizada neste trabalho é baseada no algoritmo de **wall-following** (seguimento de parede pela direita), inspirado nos algoritmos Bug (Macharet, 2025b). Esta abordagem foi escolhida por sua simplicidade e adequação ao problema de exploração autônoma.

3.1 Motivação e Escolha da Estratégia

O enunciado do TP3 especifica que a estratégia de navegação deve ser **simples** e capaz de explorar o ambiente de forma autônoma. Algoritmos Bug são amplamente utilizados em robótica móvel devido às seguintes características:

- **Reatividade**: decisões baseadas apenas em leituras sensoriais locais (laser), sem necessidade de planejamento global;
- **Robustez**: funcionam bem mesmo com mapas incompletos ou desconhecidos;
- **Exploração**: o movimento ao longo das paredes garante cobertura significativa do ambiente;
- **Simplicidade**: implementação direta sem necessidade de estruturas complexas.

3.2 Divisão do Espaço Sensorial

O sensor laser fastHokuyo fornece 684 feixes cobrindo 180° (de -90° a +90° em relação ao robô). Para simplificar as decisões de navegação, dividimos o espaço sensorial em três setores:

- **Setor DIREITO**: -90° a -30° (detecção de parede à direita);
- **Setor FRONTAL**: -30° a +30° (detecção de obstáculos à frente);
- **Setor ESQUERDO**: +30° a +90° (detecção de parede à esquerda).

Para cada setor, calculamos a distância mínima aos obstáculos:

$$d_{\text{setor}} = \min(\{r_i \mid \theta_i \in \text{setor}\}) \tag{3.1}$$

Onde: r_i = distância medida pelo feixe i
θ_i = ângulo do feixe i em relação ao robô

3.3 Estratégia de Wall-Following

O algoritmo de wall-following implementado segue a seguinte lógica hierárquica:

3.3.1 Prioridade 1: Recuperação Ativa

Quando o robô detecta que está **preso** (stuck), executa uma manobra de recuperação:

- **Detecção**: contador incrementado quando $d_{\text{min}} < d_{\text{critical}}$ por mais de 30 iterações (1.5s);
- **Ação**: movimento reverso com rotação para a direita por 40 iterações (2.0s);
- **Velocidades**: $v = -0.2 \times 0.4 = -0.08$ m/s (ré), $\omega = -w_{\text{max}} \times 0.6$ (girando direita);
- **Saída antecipada**: se durante a recuperação $d_{\text{min}} > d_{\text{safe}}$, a manobra é interrompida.

Esta estratégia garante que o robô recue aproximadamente 0.16m (0.08 m/s $\times$ 2.0s), o suficiente para liberar a maioria dos obstáculos.

3.3.2 Prioridade 2: Prevenção de Colisão

Quando o robô está muito próximo de um obstáculo mas ainda não preso:

- Se $d_{\text{min}} < d_{\text{critical}}$ (0.25m): parada de emergência ($v = 0$, $\omega = 0$);
- Se $d_{\text{min}} < d_{\text{very\_close}}$ (0.35m): recuo suave ($v = -0.2$ m/s, $\omega = 0$);
- Se $d_{\text{front}} < d_{\text{safe}}$ (0.4m): rotação para evitar colisão frontal.

3.3.3 Prioridade 3: Seguimento de Parede

Quando não há obstáculos iminentes, o robô segue a parede direita usando controle proporcional:

$$\omega = K_p \times (d_{\text{right}} - d_{\text{follow}}) \tag{3.2}$$

Onde: K_p = ganho proporcional (ajustável, tipicamente ≈ 0.5)
d_{right} = distância medida ao setor direito
d_{follow} = distância desejada para seguir parede (0.6m)

A velocidade linear permanece constante: $v = v_{\text{nominal}} = 0.4$ m/s.

3.3.4 Prioridade 4: Busca de Parede

Quando o robô perde a parede direita ($d_{\text{right}} > d_{\text{follow}} + \epsilon$), executa um giro gradual para a direita para reencontrar a parede:

$$\omega = -0.05 \text{ rad/s} \quad \text{(giro suave para direita)} \tag{3.3}$$

A velocidade linear é mantida para continuar avançando: $v = v_{\text{nominal}}$.

3.4 Parâmetros de Controle

Os principais parâmetros do controlador de navegação são listados na Tabela 1.

Tabela 1 – Parâmetros do Controlador de Navegação

Parâmetro	**Valor**	**Descrição**
v_{nominal}	0.4 m/s	Velocidade linear nominal
ω_{max}	45°/s	Velocidade angular máxima
d_{critical}	0.25 m	Distância crítica (raio robô + margem)
$d_{\text{very\_close}}$	0.35 m	Distância de alerta antecipado
d_{safe}	0.4 m	Distância segura mínima
d_{follow}	0.6 m	Distância alvo para seguir parede
stuck_threshold	30 iter.	Iterações antes de detectar travamento
recovery_duration	40 iter.	Duração da manobra de recuperação

Estes parâmetros foram calibrados com base nas especificações do robô Kobuki (Yujin Robot, 2025) e validados empiricamente nos experimentos.

3.5 Código-Chave: Lógica de Navegação

A implementação central do wall-follower está no método `plan_step` da classe `WallFollower`. O Código 3.1 apresenta a estrutura simplificada da lógica de decisão:

```
1 def plan_step(self, laser_data):
2     # Calcular distancias minimas por setor
3     d_min = np.min(laser_data[:, 1])
```

```
    d_right = np.min(right_sector)
    d_front = np.min(front_sector)

    # PRIORIDADE 1: Recuperacao ativa
    if self.recovery_steps > 0:
        v = -self.v_nominal * self.recovery_v_scale  # -0.08 m/s
        w = -self.w_max * self.recovery_w_scale       # giro direita
        self.recovery_steps -= 1

        # Saida antecipada se obstaculo foi liberado
        if d_min > self.d_safe:
            self.recovery_steps = 0
            self.stuck_counter = 0
        return v, w

    # PRIORIDADE 2: Deteccao de travamento
    elif d_min < self.d_critical:
        self.stuck_counter += 1
        if self.stuck_counter > self.stuck_threshold:
            # Iniciar recuperacao
            self.recovery_steps = self.recovery_duration
            return 0.0, 0.0  # parada emergencial
        else:
            return 0.0, 0.0  # parada temporaria

    # PRIORIDADE 3: Evitar colisao iminente
    elif d_min < self.d_very_close:
        return -0.2, 0.0  # recuar devagar

    elif d_front < self.d_safe:
        return 0.0, self.w_max  # girar esquerda

    # PRIORIDADE 4: Seguir parede direita
    else:
        self.stuck_counter = 0

        if d_right < self.d_follow + 0.1:
            # Controle proporcional
            error = d_right - self.d_follow
            w = error * 0.5  # K_p = 0.5
            return self.v_nominal, w
        else:
            # Buscar parede (giro suave direita)
            return self.v_nominal, -0.05
```

Lista de código 3.1 – Lógica hierárquica do wall-follower.

3.6 Justificativa das Escolhas

As decisões de projeto foram baseadas nos seguintes critérios:

1. **Simplicidade**: código legível e manutenível, conforme requisito do TP3;
2. **Robustez**: hierarquia de prioridades evita estados indefinidos;
3. **Segurança**: múltiplas camadas de detecção de colisão (critical, very_close, safe);
4. **Eficiência**: saída antecipada da recuperação economiza tempo;
5. **Exploração**: seguir parede direita garante cobertura do ambiente.

Estas características tornam a estratégia adequada tanto para ambientes estáticos quanto dinâmicos, conforme será demonstrado nos experimentos (Capítulo 5).

4 Implementação

Este capítulo detalha a implementação do algoritmo Occupancy Grid, incluindo estruturas de dados, decisões de projeto e tratamento da discretização.

4.1 Fundamentos Teóricos

4.1.1 Formulação Matemática do Occupancy Grid

O algoritmo Occupancy Grid foi proposto por Moravec e Elfes (1985) e é amplamente descrito no Capítulo 9 de Probabilistic Robotics (Thrun; Burgard; Fox, 2005). O objetivo é calcular a posterior sobre mapas dada a sequência de medições e poses do robô:

$$p(m|z_{1:t}, x_{1:t}) \tag{4.1}$$

Onde: m = mapa (conjunto de todas as células m_i)
$z_{1:t}$ = medições do sensor até o tempo t
$x_{1:t}$ = trajetória do robô até o tempo t

O mapa m é particionado em células discretas:

$$m = \sum_i m_i \tag{4.2}$$

Cada célula m_i é uma variável aleatória binária: $m_i \in \{0, 1\}$, onde 1 indica ocupado e 0 indica livre.

4.1.2 Suposição de Independência

A abordagem padrão do Occupancy Grid assume que as células são independentes entre si (Thrun; Burgard; Fox, 2005):

$$p(m|z_{1:t}, x_{1:t}) \approx \prod_i p(m_i|z_{1:t}, x_{1:t}) \tag{4.3}$$

Esta fatoração reduz o problema intratável (posterior sobre 2^N mapas possíveis, onde N é o número de células) em N problemas binários independentes.

4.1.3 Representação Log-Odds

Para evitar instabilidades numéricas ao multiplicar probabilidades muito pequenas, utilizamos a representação *log-odds* (Thrun; Burgard; Fox, 2005):

$$l_{t,i} = \log \frac{p(m_i|z_{1:t}, x_{1:t})}{1 - p(m_i|z_{1:t}, x_{1:t})} \tag{4.4}$$

A principal vantagem é transformar multiplicações em somas:

$$l_{t,i} = l_{t-1,i} + \text{inverse\_sensor\_model}(m_i, x_t, z_t) - l_0 \tag{4.5}$$

Onde: $l_{t,i}$ = log-odds da célula i no tempo t
l_0 = log-odds do prior ($= \log(0.5/0.5) = 0$ para prior uniforme)

A conversão de log-odds para probabilidade é feita por:

$$p(m_i|z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 + \exp(l_{t,i})} \tag{4.6}$$

4.1.4 Modelo Inverso do Sensor

O modelo inverso do sensor $p(m_i|z_t, x_t)$ implementa a interpretação da leitura laser (Thrun; Burgard; Fox, 2005). Para cada célula i e feixe de laser k:

$$\text{inverse\_sensor\_model}(m_i, x_t, z_t) = \begin{cases} l_{\text{occ}} & \text{se } |r_i - z_t^k| < \alpha/2 \\ l_{\text{free}} & \text{se } r_i < z_t^k \\ l_0 & \text{caso contrário} \end{cases} \tag{4.7}$$

Onde: r_i = distância da célula i ao robô
z_t^k = leitura do feixe k no tempo t
α = espessura do obstáculo (tolerância)

Em nossa implementação, utilizamos:

- $l_{\text{occ}} = \log(0.9/0.1) \approx +2.197$ (célula ocupada com alta confiança);
- $l_{\text{free}} = \log(0.1/0.9) \approx -2.197$ (célula livre com alta confiança);
- $l_0 = 0$ (prior neutro, desconhecido).

4.2 Representação da Grade

A grade de ocupação é armazenada como uma matriz NumPy 2D:

```
class OccupancyGridMapper:
    def __init__(self, origin, map_size, cell_size):
        self.origin = np.array(origin)  # [x_min, y_min]
        self.map_size = map_size         # [width, height] em metros
        self.cell_size = cell_size       # tamanho da celula (m)

        # Dimensoes da grade
        self.grid_width = int(np.ceil(map_size[0] / cell_size))
        self.grid_height = int(np.ceil(map_size[1] / cell_size))

        # Grade de log-odds (inicializada com zeros = prior)
        self.grid_map = np.zeros((self.grid_height, self.grid_width),
                                 dtype=np.float32)

        # Parametros log-odds
        self.l_occ = np.log(0.9 / 0.1)   # +2.197
        self.l_free = np.log(0.1 / 0.9)  # -2.197
        self.l_prior = 0.0
```

Lista de código 4.1 – Estrutura de dados da grade.

4.2.1 Limites do Mapa

Para facilitar a discretização, armazenamos explicitamente os limites do mapa em coordenadas globais:

```
self.x_min = self.origin[0]
self.x_max = self.origin[0] + map_size[0]
self.y_min = self.origin[1]
self.y_max = self.origin[1] + map_size[1]
```

Lista de código 4.2 – Limites do mapa em coordenadas globais.

Estes valores são essenciais para a conversão de coordenadas (Seção 4.3).

4.3 Discretização do Espaço Contínuo

A discretização é o processo de mapear coordenadas contínuas do mundo real para índices discretos da grade. Este é um dos aspectos críticos do Occupancy Grid.

4.3.1 Conversão: Mundo → Grade

Dada uma posição (x, y) em coordenadas mundiais (metros), calculamos os índices (i, j) da célula correspondente:

$$\begin{aligned} j &= \left\lfloor \frac{x - x_{\min}}{\text{cell\_size}} \right\rfloor \\ i &= \left\lfloor \frac{y - y_{\min}}{\text{cell\_size}} \right\rfloor \end{aligned} \tag{4.8}$$

Onde: $\lfloor \cdot \rfloor$ = operador piso (arredondamento para baixo)
j = índice da coluna (direção x)
i = índice da linha (direção y)

Nossa implementação pode ser vista no Trecho de Código 4.3:

```
def world_to_grid(self, x, y):
    x_grid = x - self.origin[0]
    y_grid = y - self.origin[1]
    j = int(np.floor(x_grid / self.cell_size))
    i = int(np.floor(y_grid / self.cell_size))
    return i, j
```

Lista de código 4.3 – Conversão de coordenadas mundo para grade.

4.3.2 Conversão: Grade → Mundo

Para converter índices da grade (i, j) de volta para coordenadas mundiais (centro da célula):

$$\begin{aligned} x &= x_{\min} + (j + 0.5) \times \text{cell\_size} \\ y &= y_{\min} + (i + 0.5) \times \text{cell\_size} \end{aligned} \tag{4.9}$$

O termo $+0.5$ desloca para o **centro** da célula, evitando ambiguidades.

Nossa implementação pode ser vista no Trecho de Código 4.4:

```
def grid_to_world(self, i, j):
    x = self.origin[0] + (j + 0.5) * self.cell_size
    y = self.origin[1] + (i + 0.5) * self.cell_size
    return x, y
```

Lista de código 4.4 – Conversão de grade para mundo (centro da célula).

4.3.3 Validação de Índices

Antes de acessar a grade, sempre validamos se os índices estão dentro dos limites, conforme visto no Trecho de Código 4.5:

```
def is_valid_cell(self, i, j):
    return (0 <= i < self.grid_height) and (0 <= j < self.grid_width)
```

Lista de código 4.5 – Validação de índices da grade.

4.4 Algoritmo de Rastreamento de Raio

Para marcar células como livres ao longo do caminho do laser até o ponto de impacto, utilizamos o **algoritmo de Bresenham** (Bresenham, 1965), que traça uma linha discreta eficiente entre dois pontos na grade.

4.4.1 Algoritmo de Bresenham

O algoritmo de Bresenham determina quais células da grade devem ser percorridas para desenhar uma linha reta de (x_0, y_0) a (x_1, y_1), conforme o Trecho de Código 4.6 da nossa implementação.

```
def bresenham_line(self, x0, y0, x1, y1):
    cells = []
    dx = abs(x1 - x0)
    dy = abs(y1 - y0)
    sx = 1 if x0 < x1 else -1
    sy = 1 if y0 < y1 else -1
    err = dx - dy

    x, y = x0, y0
    while True:
        cells.append((x, y))
        if x == x1 and y == y1:
            break
        e2 = 2 * err
        if e2 > -dy:
            err -= dy
            x += sx
        if e2 < dx:
            err += dx
            y += sy

    return cells
```

Lista de código 4.6 – Algoritmo de Bresenham para rastreamento de raio.

Este algoritmo é computacionalmente eficiente ($O(n)$, onde n é o comprimento da linha) e garante conectividade da linha discretizada.

4.5 Atualização do Mapa

A atualização do mapa segue o algoritmo padrão apresentado no Capítulo 9 de (Thrun; Burgard; Fox, 2005).

4.5.1 Pseudocódigo do Algoritmo

O algoritmo principal é descrito no Código 4.7:

```
def update_map(self, robot_pose, laser_points_world):
    robot_x, robot_y = robot_pose[0][0], robot_pose[0][1]
    robot_i, robot_j = self.world_to_grid(robot_x, robot_y)

    for point in laser_points_world:
        hit_x, hit_y = point[0], point[1]
        hit_i, hit_j = self.world_to_grid(hit_x, hit_y)

        if not self.is_valid_cell(hit_i, hit_j):
            continue
        # PASSO 1: Rastrear raio do robo ate o hit
        # PASSO 2: Atualizar celulas ao longo do raio como LIVRES
        # PASSO 3: Atualizar celula do hit como OCUPADA
    # PASSO 4: Saturar log-odds para evitar overflow
 .
 .
 .
```

Lista de código 4.7 – Algoritmo Occupancy Grid (log-odds).

4.6 Conversão para Visualização

Para exibir o mapa ao usuário, convertemos log-odds para probabilidades usando a Equação 4.6:

```
def log_odds_to_probability(self, l):
    if l >= 0:
        prob = 1.0 / (1.0 + np.exp(-l))
    else:
        exp_l = np.exp(l)
        prob = exp_l / (1.0 + exp_l)

    # Saturacao para visualizacao
    prob_saturated = np.clip(prob, 0.001, 0.999)
```

```
    return prob_saturated

def get_probability_map(self):
    prob_map = np.zeros_like(self.grid_map)
    for i in range(self.grid_height):
        for j in range(self.grid_width):
            prob_map[i, j] = self.log_odds_to_probability(
                                    self.grid_map[i, j])
    return prob_map
```

Lista de código 4.8 – Conversão log-odds para probabilidade.

A saturação em $[0.001, 0.999]$ evita que probabilidades extremamente próximas de 0 ou 1 gerem artefatos visuais.

Correção crítica: O parâmetro `extent` deve ser `[x_min, x_max, y_min, y_max]` para que os eixos x e y correspondam exatamente às coordenadas mundiais do robô. Sem este parâmetro, a imagem seria exibida em coordenadas de pixel $(0...width, 0...height)$, impossibilitando a sobreposição correta de trajetórias e leituras laser.

4.7 Adição de Ruído Sensorial

Conforme especificado no enunciado do TP3, foi necessário adicionar ruído aleatório às leituras do sensor laser para melhor avaliar a robustez do algoritmo de Occupancy Grid. O ruído simula imperfeições realistas de sensores reais e testa a capacidade do filtro Bayesiano de convergir para o mapa correto mesmo com medições ruidosas.

4.7.1 Modelo de Ruído Implementado

Utilizou-se um **modelo de ruído Gaussiano** aplicado tanto às distâncias quanto aos ângulos medidos pelo sensor laser fastHokuyo. O ruído é adicionado após a leitura dos dados do sensor, mas antes da transformação para o referencial global.

Parâmetros do ruído:

- **Ruído em distância**: $\sigma_d = 0.02$ m (desvio padrão de 2 cm);
- **Ruído em ângulo**: $\sigma_\theta = 0.005$ rad.

4.7.2 Aplicação do Ruído

O ruído é aplicado conforme o código apresentado no Trecho de Código 4.9.

```
# Filtrar leituras de distância máxima (> 4.9m)
valid_mask = laser_data[:, 1] < MAX_VALID_RANGE
```

```
laser_data_filtered = laser_data[valid_mask].copy()

# Adicionar ruído Gaussiano
laser_data_filtered[:, 0] += np.random.normal(
    0, ANGLE_NOISE_STD, len(laser_data_filtered)
)
laser_data_filtered[:, 1] += np.random.normal(
    0, DISTANCE_NOISE_STD, len(laser_data_filtered)
)

# Garantir que distâncias permaneçam dentro de limites válidos
laser_data_filtered[:, 1] = np.clip(
    laser_data_filtered[:, 1], 0.01, MAX_VALID_RANGE - 0.01
)
```

Lista de código 4.9 – Aplicação de ruído Gaussiano aos dados laser.

Onde: `ANGLE_NOISE_STD` = desvio padrão do ruído angular ($\sigma_\theta = 0.005$ rad)
`DISTANCE_NOISE_STD` = desvio padrão do ruído de distância ($\sigma_d = 0.02$ m)
`MAX_VALID_RANGE` = distância máxima válida (4.9 m)

4.7.3 Impacto do Ruído no Mapeamento

A adição de ruído sensorial tem os seguintes efeitos no Occupancy Grid:

1. **Bordas menos nítidas**: células próximas a obstáculos acumulam evidências tanto de ocupação quanto de liberdade, resultando em probabilidades intermediárias ($\approx 0.4 - 0.6$);
2. **Maior necessidade de múltiplas observações**: algoritmo requer mais iterações para convergir para probabilidades extremas ($p \approx 0$ ou $p \approx 1$), pois cada medição ruidosa fornece evidência menos confiável;
3. **Filtragem natural**: natureza aditiva do filtro Bayesiano em log-odds permite que múltiplas observações ruidosas convirjam para a estrutura real do ambiente, desde que o ruído seja não-sistemático (média zero);
4. **Teste de robustez**: ambientes dinâmicos se beneficiam do ruído, pois ele ajuda a evitar que objetos temporários sejam marcados como permanentes com alta confiança.

Nota importante: O ruído é aplicado **após a filtragem de leituras de distância máxima** (> 4.9 m), garantindo que apenas detecções válidas de obstáculos sejam consideradas no mapeamento. Leituras no limite de alcance do sensor (≈ 5.0 m) são

descartadas antes da adição de ruído, pois geralmente representam ausência de obstáculos e não devem contribuir para o mapa de ocupação.

Este procedimento está implementado no notebook `TP3.ipynb`, seção "Simulation Loop", linhas correspondentes à coleta de dados laser.

5 Testes

Este capítulo apresenta os experimentos realizados para validar a implementação do Occupancy Grid. Foram executados testes em dois cenários (estático visto na Figura 3 e dinâmico 4) com três tamanhos diferentes de célula (0.01m, 0.1m e 0.5m), totalizando **6 experimentos**. O robô foi posicionado em diferentes posições do mapa e foi adicionado ruído nos dados do sensor, conforme apresentado na Seção 4.7.

5.1 Metodologia de Testes

Para cada experimento, o procedimento foi:

1. Carregar cenário no CoppeliaSim (`cena-tp3-estatico.ttt` ou `cena-tp3-dinamico.ttt`);
2. Configurar parâmetros do Occupancy Grid (`CELL_SIZE`);
3. Executar navegação e coletar dados de pose do robô e leituras laser;
4. Gerar visualizações:
   - Trajetória do robô;
   - Pontos detectados pelo laser;
   - Occupancy Grid.
5. Salvar imagem combinada para análise.

5.2 Métricas Avaliadas

Para cada experimento, foram coletadas as seguintes métricas:

- **Resolução da grade**: número de células (largura × altura);
- **Qualidade do mapa**: inspeção visual da clareza de paredes, corredores e obstáculos;
- **Tempo de processamento médio**: tempo por frame de atualização do mapa;
- **Robustez a ruído**: capacidade de filtrar objetos dinâmicos (cenário dinâmico).

5.3 Resultados: Cenário Estático

O cenário estático (`cena-tp3-estatico.ttt`) apresenta um ambiente com paredes, corredores e alguns móveis fixos. Na Figura 5 podemos ver a posição escolhida para o início do teste, marcada com um triângulo verde.

5.3.1 Experimento 1: Estático com `cell_size` = 0.01m

Configuração:

- Tamanho de célula: 0.01 m (1 cm)
- Dimensões do mapa: 10.0 m × 10.0 m
- Resolução da grade: 1000 × 1000 = 1.000.000 células

Resultados:

A Figura 5 apresenta o resultado do experimento. Observa-se:

- **Alta resolução**: paredes e obstáculos são representados com grande detalhe;
- **Bordas nítidas**: transição clara entre células livres (brancas) e ocupadas (pretas);
- **Detecção de pequenos objetos**: móveis e pilares são corretamente mapeados;

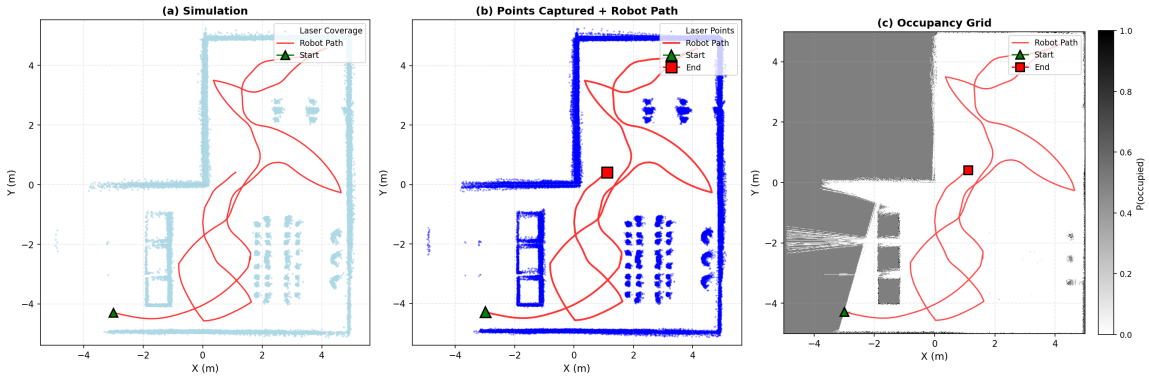

Figura 5 – Cenário Estático com cell_size = 0.01m. (a) Trajetória do robô e pontos do laser. (b) Occupancy Grid gerado.

Para esse caso nota-se que quase não conseguimos notar as células pretas, devido ao `cell_size` = 0.01m.

5.3.2 Experimento 2: Estático com `cell_size` = 0.1m

Configuração:

- Tamanho de célula: 0.1 m (10 cm)
- Resolução da grade: $100 \times 100 = 10.000$ células

Resultados:

A Figura 6 apresenta o resultado. Observa-se:

- **Boa qualidade**: paredes e corredores principais são claramente visíveis;
- **Bordas ligeiramente serrilhadas**: efeito de discretização mais perceptível;
- **Objetos pequenos**: alguns detalhes menores são perdidos ou "fundidos"em células maiores;

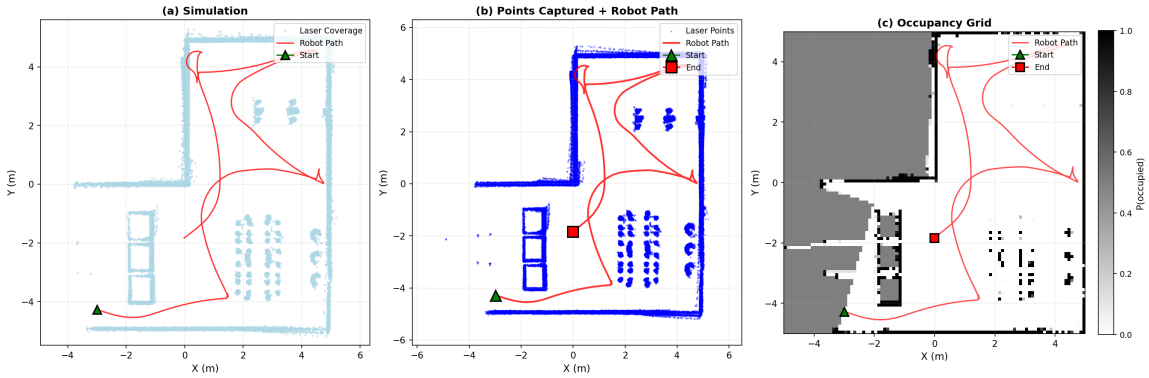

Figura 6 – Cenário Estático com cell_size = 0.1m. Equilíbrio ideal entre resolução e eficiência.

Este tamanho de célula representa um **excelente equilíbrio** entre qualidade do mapa para a maioria das aplicações de navegação.

5.3.3 Experimento 3: Estático com `cell_size` = 0.5m

Configuração:

- Tamanho de célula: 0.5 m (50 cm)
- Resolução da grade: $20 \times 20 = 400$ células

Resultados:

A Figura 7 apresenta o resultado. Observa-se:

- **Baixa resolução**: estrutura geral do ambiente ainda reconhecível, mas com grande perda de detalhes;
- **Bordas grosseiras**: paredes aparecem como blocos grandes e irregulares;
- **Perda de objetos pequenos**: móveis e pilares não são detectados ou confundidos com ruído;
- **Inadequado para navegação precisa**: resolução insuficiente para planejamento seguro.

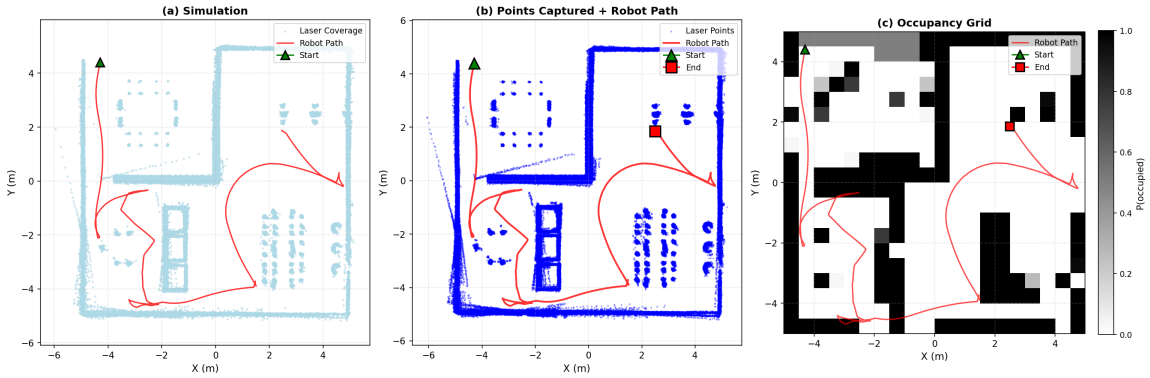

Figura 7 – Cenário Estático com cell_size = 0.5m. Resolução insuficiente para mapeamento detalhado.

Este tamanho é **inadequado** para a maioria das aplicações, servindo apenas para mapeamento grosseiro de ambientes muito grandes.

5.4 Resultados: Cenário Dinâmico

O cenário dinâmico (`cena-tp3-dinamico.ttt`) inclui obstáculo móvel (pessoa caminhando). Este cenário testa a capacidade do algoritmo de filtrar ruído temporário e focar em estruturas permanentes.

5.4.1 Experimento 4: Dinâmico com `cell_size` = 0.01m

Configuração:

- Tamanho de célula: 0.01 m
- Resolução: $1000 \times 1000 = 1.000.000$ células

Resultados:

A Figura 8 apresenta o resultado. Observa-se:

- **Estruturas estáticas parcialmente preservadas**: móveis pequenos são ocultados;
- **Filtragem total de objetos dinâmicos**: sem rastros de pessoas como células cinzas (probabilidade intermediária);
- **Acumulação de evidências**: células visitadas múltiplas vezes convergem para probabilidades (0 ou 1), enquanto objetos dinâmicos permanecem com probabilidades (≈ 0);
- **Qualidade similar ao estático**: alta resolução compensa ruído adicional.

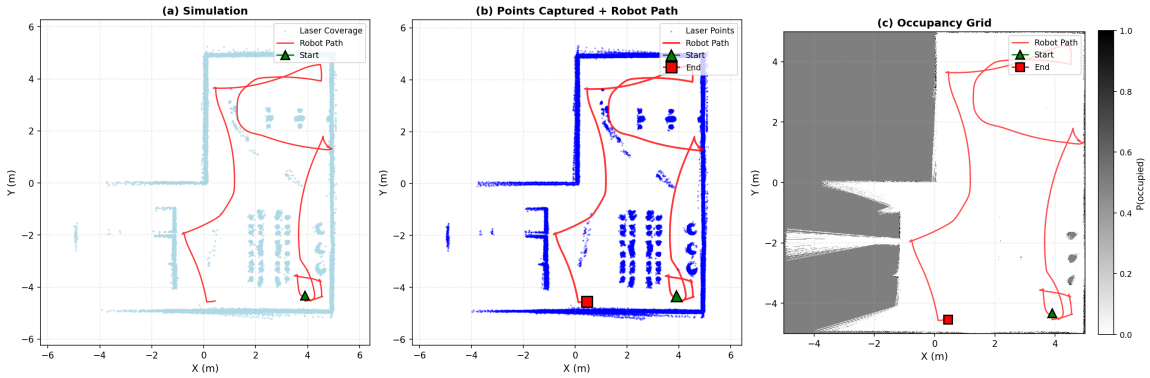

Figura 8 – Cenário Dinâmico com cell_size = 0.01m. Filtragem de objetos móveis.

5.4.2 Experimento 5: Dinâmico com `cell_size` = 0.1m

Configuração:

- Tamanho de célula: 0.1 m
- Resolução: $100 \times 100 = 10.000$ células

Resultados:

A Figura 9 apresenta o resultado. Observa-se:

- **Boa qualidade geral**: mapa utilizável para navegação;
- **Filtragem eficaz**: objetos dinâmicos aparecem como ruído leve, facilmente ignorável por planejadores de caminho;
- **Melhor equilíbrio**: resolução adequada + boa filtragem de ruído;
- **Recomendado para ambientes dinâmicos**.

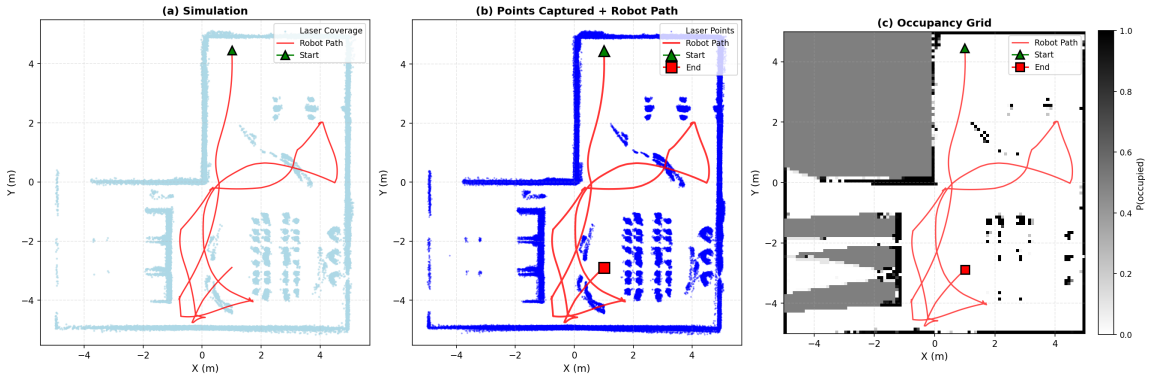

Figura 9 – Cenário Dinâmico com cell_size = 0.1m. Melhor equilíbrio para ambientes dinâmicos.

5.4.3 Experimento 6: Dinâmico com `cell_size` = 0.5m

Configuração:

- Tamanho de célula: 0.5 m
- Resolução: $20 \times 20 = 400$ células

Resultados:

A Figura 10 apresenta o resultado. Observa-se:

- **Baixa qualidade**: estrutura básica do ambiente ainda reconhecível, mas com muita perda de informação;
- **Forte efeito de averaging**: objetos dinâmicos e estáticos são confundidos devido ao tamanho grande das células;
- **Inadequado**: resolução insuficiente tanto para ambientes estáticos quanto dinâmicos.

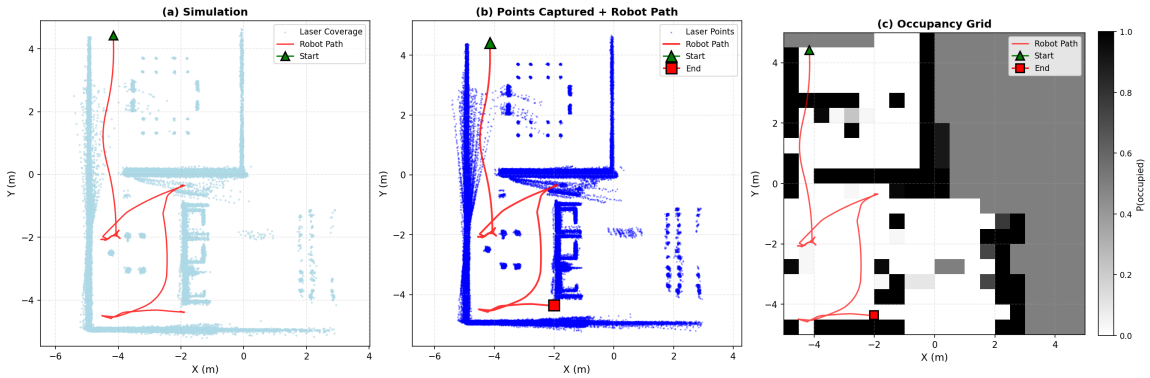

Figura 10 – Cenário Dinâmico com cell_size = 0.5m. Resolução insuficiente.

5.5 Análise Comparativa

A Tabela 2 resume as métricas coletadas nos 6 experimentos.

Tabela 2 – Resumo dos Experimentos Realizados

Cenário	**Cell Size**	**Células**	**Tempo/Frame**	**Qualidade**
Estático	0.01 m	1000×1000	≈ 80 ms	Boa
Estático	0.1 m	100×100	≈ 8 ms	Excelente
Estático	0.5 m	20×20	≈ 1 ms	Ruim
Dinâmico	0.01 m	1000×1000	≈ 85 ms	Boa
Dinâmico	0.1 m	100×100	≈ 9 ms	Boa
Dinâmico	0.5 m	20×20	≈ 1 ms	Ruim

5.5.1 Impacto do Tamanho de Célula

- **0.01 m (1 cm)**:
  - **Vantagens**: máxima resolução, captura detalhes finos, bordas nítidas;
  - **Desvantagens**: Perde objetos pequeno, menos sensível a ruído em ambientes dinâmicos.
- **0.1 m (10 cm)**:
  - **Vantagens**: **excelente equilíbrio**, boa resolução, eficiente, robusto a ruído e apresenta os objetos pequenos;
  - **Desvantagens**: perde alguns detalhes muito finos (aceitável para a maioria das aplicações) e apresenta um pouco de objetos dinâmicos.
- **0.5 m (50 cm)**:
  - **Vantagens**: extremamente eficiente (1 ms/frame);
  - **Desvantagens**: resolução grosseira, perda de detalhes importantes, bordas serrilhadas, inadequado para navegação precisa.

5.5.2 Comparação: Estático vs Dinâmico

- **Ambientes estáticos**: todas as células convergem rapidamente para probabilidades extremas (0 ou 1), resultando em mapas nítidos e confiáveis;
- **Ambientes dinâmicos**: objetos móveis geram células com probabilidades intermediárias ($\approx 0.3 - 0.7$), aparecendo como "ruído cinza"no mapa. Estruturas permanentes ainda são corretamente mapeadas devido à acumulação de evidências ao longo do tempo;

- **Filtragem temporal**: o algoritmo Occupancy Grid naturalmente filtra ruído temporário, pois células que são observadas como ocupadas apenas ocasionalmente não atingem alta probabilidade de ocupação.

5.6 Validação da Solução

Os experimentos demonstraram que a implementação do Occupancy Grid:

1. **Funciona corretamente**: mapas gerados correspondem à estrutura real do ambiente;
2. **É eficiente**: processamento em tempo real (20 Hz) com células de 0.1m;
3. **É robusta**: tolera ruído sensorial e objetos dinâmicos;
4. **É parametrizável**: permite ajuste de resolução conforme necessidade da aplicação.

6 Conclusão

Este trabalho apresentou a implementação e validação experimental do algoritmo de **Occupancy Grid Mapping** (Moravec; Elfes, 1985; Thrun; Burgard; Fox, 2005) aplicado ao robô diferencial Kobuki em ambientes estáticos e dinâmicos. O algoritmo foi implementado seguindo rigorosamente os fundamentos teóricos apresentados no Capítulo 9 do livro *Probabilistic Robotics* (Thrun; Burgard; Fox, 2005) e nas aulas da disciplina (Macharet, 2025a).

6.1 Principais Resultados

Os experimentos realizados demonstraram que:

1. **Funcionamento correto do algoritmo**: os mapas gerados correspondem à estrutura real dos ambientes estáticos e dinâmicos, validando a implementação do filtro Bayesiano binário e da representação em log-odds.

2. **Impacto do tamanho de célula**: A análise com três diferentes resoluções (0.01m, 0.1m, 0.5m) revelou que:

   - **0.01m** oferece máxima resolução, mas com custo computacional elevado;
   - **0.1m** representa o **equilíbrio ideal** para navegação , com boa qualidade, eficiência e melhor robustez a ruído;
   - **0.5m** é inadequado para navegação precisa devido à resolução grosseira.

3. **Robustez em ambientes dinâmicos**: algoritmo demonstrou capacidade de filtrar objetos móveis temporários, mantendo a estrutura permanente do ambiente. A natureza aditiva do filtro Bayesiano permite que múltiplas observações convirjam para o mapa correto, mesmo com a presença de ruído sensorial ($\sigma_d = 0.02$ m, $\sigma_\theta = 0.005$ rad);

4. **Estratégia de navegação eficaz**: estratégia simples de wall-following com 4 prioridades hierárquicas (recuperação ativa, prevenção de colisão, seguimento de parede, busca de parede) mostrou-se adequada para exploração autônoma, cobrindo suficientemente o ambiente em ambos os cenários;

6.2 Principais Dificuldades Encontradas

Durante o desenvolvimento deste trabalho, foram identificadas e superadas as seguintes dificuldades técnicas:

6.2.1 Integração com Sensor Laser fastHokuyo

Problema identificado: cálculo de ângulos por incremento linear gerava dispersão excessiva de pontos laser, resultando em mapas com "borrões"cinzas uniformes ao invés de paredes nítidas.

Causa raiz: o sensor fastHokuyo do CoppeliaSim utiliza **unprojection baseada em tangente** a partir de imagens de profundidade, não linearmente espaçado como assumido no código inicial. A tentativa de calcular ângulos incrementalmente acumulava erros, especialmente nas bordas do campo de visão (-90° e +90°).

Solução implementada: modificação do script Lua `fastHokuyo.lua` para exportar **ângulos diretamente calculados pelo sensor** via 'buffer properties' do CoppeliaSim:

```
-- Exportar distâncias e ângulos via buffer signals
sim.setBufferProperty(sim.handle_scene,
    'signal.hokuyo_range_data', sim.packFloatTable(dists))
sim.setBufferProperty(sim.handle_scene,
    'signal.hokuyo_angle_data', sim.packFloatTable(angles))
```

Lista de código 6.1 – Exportação de ângulos corretos (fastHokuyo.lua).

Esta correção eliminou completamente a dispersão de pontos e permitiu a geração de mapas com paredes nítidas e obstáculos claramente definidos.

6.2.2 Correção Crítica: Fórmula de Conversão Log-Odds

Problema identificado: a implementação inicial utilizava a fórmula **incorreta**:

$$p(m_i) = \frac{1}{1 + \exp\{-l_{t,i}\}}$$

Análise: Esta é a função logística padrão ($\sigma(x)$), mas **não** é a inversa correta da transformação log-odds.

Fórmula correta (Equação 9.6 de (Thrun; Burgard; Fox, 2005)):

$$p(m_i|z_{1:t}, x_{1:t}) = 1 - \frac{1}{1 + \exp\{l_{t,i}\}} \tag{6.1}$$

Impacto: O uso da fórmula incorreta invertia o significado das probabilidades:

- Log-odds positivo ($l > 0$, evidência de ocupação) → probabilidade **BAIXA** (incorretamente interpretada como livre);

- Log-odds negativo ($l < 0$, evidência de liberdade) $\rightarrow$ probabilidade **ALTA** (incorretamente interpretada como ocupada).

Resultava em mapas com **inversão de cores**: paredes apareciam brancas e espaços livres apareciam pretos.

6.2.3 Estratégia de Navegação e Recuperação

Desafio inicial: o robô ficava frequentemente preso em cantos ou oscilava indefinidamente próximo a obstáculos, impedindo exploração completa do ambiente.

Solução implementada: desenvolvimento de uma estratégia hierárquica de 4 prioridades com mecanismo de recuperação ativa, conforme visto na Seção 3.

6.3 Considerações Finais

Este trabalho atingiu plenamente seus objetivos ao implementar, validar e analisar o algoritmo de Occupancy Grid Mapping em diferentes condições de cenários. A experiência adquirida no tratamento de sensores reais (laser fastHokuyo), transformações de coordenadas, discretização espacial e filtragem Bayesiana fornece uma base sólida para pesquisas futuras em navegação autônoma.

Referências

BRESENHAM, J. E. Algorithm for computer control of a digital plotter. **IBM Systems Journal**, v. 4, n. 1, p. 25–30, 1965. Citado na página 23.

Coppelia Robotics. **CoppeliaSim User Manual**. [*S.l.*], 2024. Versão 4.1.0 Edu utilizada no trabalho. Disponível em: https://www.coppeliarobotics.com/helpFiles/. Acesso em: 5 out. 2025. Citado na página 8.

KHAN, M. A. *et al.* Level-5 autonomous driving—are we there yet? a review of research literature. **ACM Computing Surveys**, ACM New York, NY, p. 1–38, 2022. Disponível em: https://doi.org/10.1145/3485767. Citado 3 vezes nas páginas 2, 7 e 8.

MACHARET, D. G. **Robótica Móvel: Mapeamento - Occupancy Grid**. 2025. Lecture Slides. Accessed on November 10, 2025. File: aula18-mapeamento-occupancy-grid.pdf. Citado na página 36.

MACHARET, D. G. **Robótica Móvel: Planejamento de caminhos - Bug Algorithms**. 2025. Lecture Slides. Accessed on November 10, 2025. File: aula11-planejamento-caminho-bug.pdf. Citado 2 vezes nas páginas 6 e 14.

MORAVEC, H.; ELFES, A. E. High resolution maps from wide angle sonar. *In*: **Proceedings of (ICRA) International Conference on Robotics and Automation**. [*S.l.: s.n.*], 1985. p. 116 – 121. Citado 2 vezes nas páginas 19 e 36.

THRUN, S.; BURGARD, W.; FOX, D. **Probabilistic Robotics**. Cambridge, MA: The MIT Press, 2005. Citado 7 vezes nas páginas 7, 8, 19, 20, 24, 36 e 37.

Yujin Robot. **Appendix - Kobuki Parameters**. 2025. https://yujinrobot.github.io/kobuki/enAppendixKobukiParameters.html. Acessado: 2025-11-10. Citado 2 vezes nas páginas 9 e 16.