

UNIVERSIDADE FEDERAL DE MINAS GERAIS

Instituto de Ciências Exatas

Departamento de Ciência da Computação

Daniel Terra Gomes

Trabalho Prático 1: Ferramentas e Transformações

Belo Horizonte, MG

11 de setembro 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)

Listas de ilustrações

Figura 1 – Cena criada no CoppeliaSim contendo o robô RobotnikSummitXL e diversos objetos.	7
Figura 2 – Diagrama de transformações mostrando os sistemas de coordenadas e as relações entre os diferentes frames na cena. Para exemplificação, as setas verdes representam transformações conhecidas, enquanto a seta vermelha mostra uma transformação desejada.	8
Figura 3 – Poses dos objetos da perspectiva do Robô.	13
Figura 4 – Cenário A: Posição na coordenada (0, 0).	14
Figura 5 – Cenário B: Posição ao lado do Bill_1.	14
Figura 6 – Cenário C: Posição com traseira para o Crate.	14
Figura 7 – Diferentes posições do robô utilizadas para validação das transformações.	14
Figura 8 – Poses dos objetos da perspectiva do Robô no Cenário A: 0 0.	15
Figura 9 – Cenário B: Posição de lado para bill_1.	15
Figura 10 – Cenário modificado para os exercícios 5-6, com o robô Pioneer P3DX e sensor laser Hokuyo.	16
Figura 11 – Dados do Laser no Referencial Global.	19
Figura 12 – Posição 2 (Após movimento).	19
Figura 13 – Mapa incremental com o caminho executado pelo robô.	22

Listas de trechos de código

1.1	Mapeamento de objetos para o CoppeliaSim.	8
1.2	Funções para matrizes de rotação	9
1.3	Função para criar matriz homogênea	10
1.4	Função para inverter matriz homogênea	11
1.5	Obtenção da pose do robô.	11
1.6	Cálculo das transformações relativas.	12
1.7	Análise de transformações em diferentes cenários.	14
1.8	Mapeamento de objetos atualizado para exercícios 5-6.	16
1.9	Função para transformar dados do laser para o referencial global.	17
1.10	Função para plotar dados do laser no referencial global.	18
1.11	Função para plotar mapa incremental.	19
1.12	Algoritmo de navegação autônoma com desvio de obstáculos.	20

Sumário

1	INTRODUÇÃO	5
1.1	Ambiente de Desenvolvimento	5
1.1.1	Estrutura do Código	5
1.2	Execução da Simulação	6
1.3	Exercício 1: Criação da Cena no CoppeliaSim	7
1.4	Exercício 2: Diagrama de Transformações	8
1.5	Exercício 3: Matrizes de Transformação Homogêneas	9
1.5.1	Análise da Posição Inicial	11
1.6	Exercício 4: Múltiplas Posições do Robô	13
1.7	Exercício 5: Transformação de Dados do Laser para o Referencial Global	15
1.7.1	Atualização do Cenário	15
1.7.2	Transformações entre Referenciais	16
1.7.3	Visualização dos Dados Transformados	18
1.8	Exercício 6: Navegação Autônoma e Mapeamento Incremental	19
2	CONCLUSÃO	23
2.1	Resultados Obtidos	23
2.2	Dificuldades Enfrentadas	23
	REFERÊNCIAS	25

1 Introdução

Esta documentação apresenta as soluções encontradas para o trabalho prático (TP1), de maneira sequencial.

Inicialmente, são detalhados os procedimentos para a criação da cena de simulação no CoppeliaSim ([Robotics, 2023](#)), a elaboração do diagrama de transformações entre os referenciais e a implementação das matrizes de transformação homogênea para análise de poses relativas para os Exercícios 1 a 4, vistos nas Seções [1.3](#), [1.4](#), [1.5](#) e [1.6](#). Em seguida, a documentação aborda a integração de um sensor a laser no robô PioneerP3DX, a transformação de suas leituras para o referencial global e, por fim, a implementação de um sistema de navegação autônoma com mapeamento incremental do ambiente, vistos nas Seções [1.7](#) e [1.8](#).

Dessa forma, este documento visa apresentar a linha de raciocínio para implementação e resultados obtidos em cada etapa do trabalho, destacando as abordagens matemáticas e computacionais utilizadas para solucionar os problemas propostos.

1.1 Ambiente de Desenvolvimento

O desenvolvimento do trabalho foi realizado utilizando as seguintes ferramentas e tecnologias:

- **CoppeliaSim V4.1.0 Edu, Ubuntu 20.04:** simulador de robótica utilizado para criar cenários virtuais e simular robôs e sensores ([Robotics, 2023](#));
- **Draw.io:** ferramenta utilizada para a criação do diagrama de transformações ([JGraph Ltd, 2025](#));
- **Python 3.13.7 via Miniconda:** utilizado para processamento de dados, visualização, uso de suas bibliotecas numéricas e plots ([Anaconda, Inc., 2025](#))...
- **Jupyter Notebooks via VScode:** ambiente interativo para execução de código, documentação e visualização de resultados ([Microsoft Corporation, 2025](#));
- **ZMQ Remote API:** interface de comunicação com o CoppeliaSim para controle dos objetos e robôs na simulação ([Hintjens, 2013](#)).

1.1.1 Estrutura do Código

Para facilitar a reutilização de código e organizar a implementação, foi criado um módulo de utilitários chamado `robotics_utils.py`, contendo classes e funções para:

- Conexão e comunicação com o CoppeliaSim;
- Transformações matemáticas (matrizes homogêneas, rotações);
- Análise e visualização de cenas;
- Simulação de sensores laser;
- Funções auxiliares para navegação e mapeamento...

Este módulo é importado nos notebooks `TP1_ex1-4.ipynb` e `TP1_ex5-6.ipynb` que implementam os exercícios propostos.

1.2 Execução da Simulação

Para a correta execução das soluções, é necessário que o CoppeliaSim esteja em execução com a cena apropriada carregada, conforme descrito a seguir:

- Para os exercícios 1 a 4, deve-se utilizar a cena `T1.ttt` e executar o notebook `TP1_ex1-4.ipynb`. Note que, para o exercício 4, é necessário movimentar o robô manualmente na cena do CoppeliaSim antes dar "enter" para cada célula de análise de cenário.
- Para os exercícios 5 e 6, a cena a ser utilizada é a `T1-ex5-6.ttt`, e o notebook correspondente é o `TP1_ex5-6.ipynb`.

1.3 Exercício 1: Criação da Cena no CoppeliaSim

O primeiro exercício consistiu na criação de uma cena no CoppeliaSim contendo um robô móvel e cinco elementos distintos para compor o ambiente de simulação. Sabendo disso, escolhemos 6 objetos distintos e o Robô "*RobotnikSummitXL*", conforme podemos identificar na Figura 1.

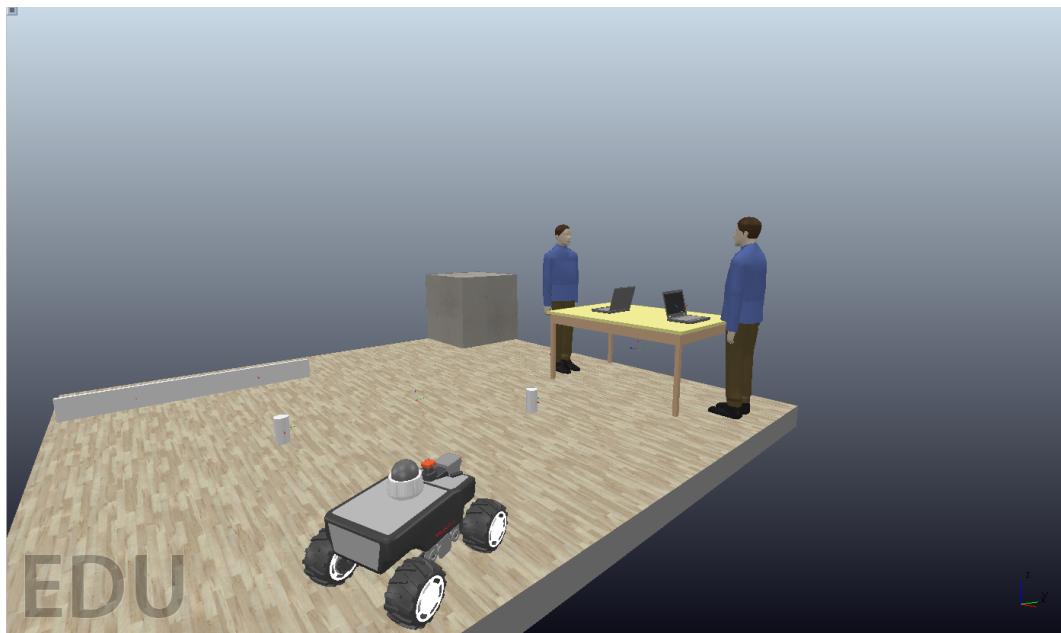


Figura 1 – Cena criada no CoppeliaSim contendo o robô RobotnikSummitXL e diversos objetos.

Desse forma, os seguintes elementos foram incluídos na cena. Dos quais, dois deles deram problema nos exercícios 5 e 6, conforme veremos no capítulo 2:

- 1 Robô móvel: RobotnikSummitXL;
- 2 Pessoas: Bill[0] e Bill[1];
- 1 Caixa: Floor/ConcretBlock;
- 2 Pilares: Floor/20cmHighPillar10cm[0] e [1];
- 1 Mesa: diningTable;
- 2 Laptops: diningTable/laptop[0] e [1];
- 2 Cercas: Floor/20cmHighWall100cm[0] e [1].

Esses mesmos elementos foram usados na implementação do mapeamento de objetos, conforme podemos identificar no Trecho de Código 1.1:

```

1 DEFAULT_OBJECT_MAPPING_EX1_4 = {
2     'Robot': 'RobotnikSummitXL',
3     'Bill_0': 'Bill[0]',
4     'Bill_1': 'Bill[1]',
5     'Crate': 'Floor/ConcretBlock',
6     'Pillar_0': 'Floor/20cmHighPillar10cm[0]',
7     'Pillar_1': 'Floor/20cmHighPillar10cm[1]',
8     'Table': 'diningTable',
9     'Laptop_0': 'diningTable/laptop[0]',
10    'Laptop_1': 'diningTable/laptop[1]',
11    'Fence_0': 'Floor/20cmHighWall100cm[0]',
12    'Fence_1': 'Floor/20cmHighWall100cm[1]'
13 }

```

Lista de código 1.1 – Mapeamento de objetos para o CoppeliaSim.

Este mapeamento permitiu a descoberta e manipulação dos objetos na cena através da interface ZMQ Remote API do CoppeliaSim de maneira mais eficiente, visto que uma cena carregada pode ter diferentes objetos de não interesse. Note, o mesmo é feito para os exercícios 5 e 6.

1.4 Exercício 2: Diagrama de Transformações

Neste exercício, foi desenvolvido um diagrama representando as relações entre os sistemas de coordenadas dos objetos na cena. O frame do Mundo $\{W\}$ serve como referência global para todas as transformações.

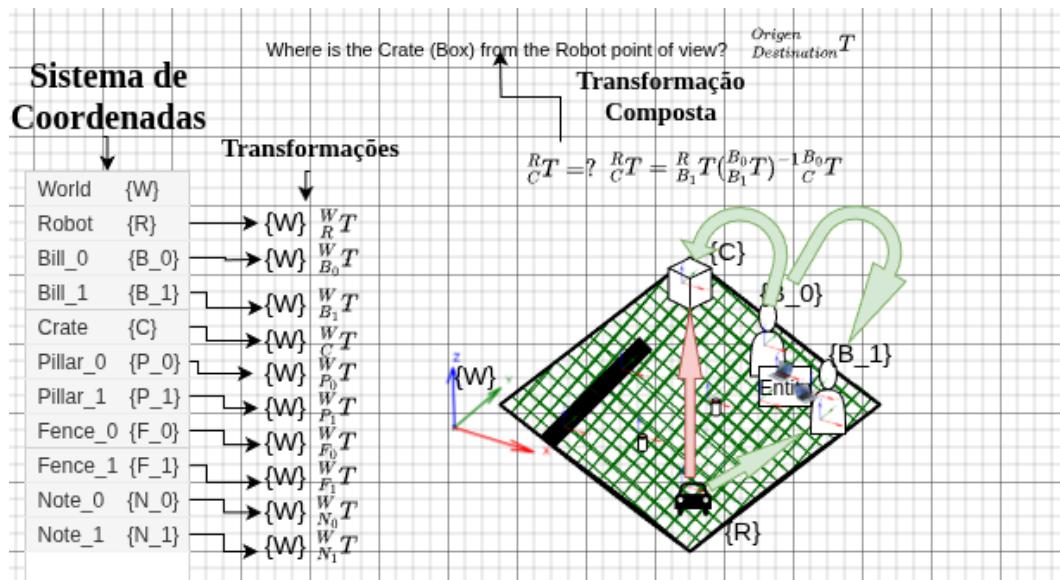


Figura 2 – Diagrama de transformações mostrando os sistemas de coordenadas e as relações entre os diferentes frames na cena. Para exemplificação, as setas verdes representam transformações conhecidas, enquanto a seta vermelha mostra uma transformação desejada.

Ainda, o diagrama ilustra:

- Sistemas de coordenadas de cada objeto da nossa cena T1;
- As transformações dos sistemas de coordenadas e um exemplo extra de transformação composta.

A representação matemática destas transformações é feita através de matrizes homogêneas da forma (Macharet, 2025, p. 6):

$${}^A_B T = \begin{bmatrix} {}^A_B R & {}^A_B p \\ 0_{1 \times 3} & 1 \end{bmatrix} \quad (1.1)$$

Onde: ${}^A_B T$ = Matriz de transformação homogênea do frame B para o frame A;

${}^A_B R$ = Matriz de rotação 3×3 do frame B para o frame A;

${}^A_B p$ = Vetor de translação 3×1 da origem do frame B p/ no frame A;

$0_{1 \times 3}$ = Vetor linha de zeros;

1 = Elemento escalar para completar a matriz homogênea.

1.5 Exercício 3: Matrizes de Transformação Homogêneas

O terceiro exercício consistiu na implementação das matrizes de transformação homogêneas que representam as posições de todos os elementos da cena no referencial local do robô. Para isso, foram desenvolvidas funções específicas para:

1. Criar matrizes de rotação em torno dos eixos X, Y e Z;
2. Compor matrizes homogêneas a partir de posições e orientações;
3. Calcular transformações inversas;
4. Visualizar as relações espaciais entre objetos.

Dessa forma, realizamos a implementação das Funções de Transformação. Onde as funções de rotação em torno dos eixos principais foram implementadas, conforme o Trecho de Código 1.2.

```

1 def Rz(theta: float) -> np.ndarray:
2     """
3         Create a 3x3 rotation matrix around the Z-axis.
4     """
5     return np.array([[np.cos(theta), -np.sin(theta), 0],
6                     [np.sin(theta), np.cos(theta), 0],
7                     [0, 0, 1]])

```

```

6                 [np.sin(theta), np.cos(theta), 0],
7                 [0,                      0,                  1]])
8
9 def Ry(theta: float) -> np.ndarray:
10    """
11        Create a 3x3 rotation matrix around the Y-axis.
12    """
13    return np.array([[np.cos(theta), 0, np.sin(theta)],
14                    [0,                  1, 0],
15                    [-np.sin(theta), 0, np.cos(theta)]])
16
17 def Rx(theta: float) -> np.ndarray:
18    """
19        Create a 3x3 rotation matrix around the X-axis.
20    """
21    return np.array([[1, 0, 0],
22                    [0, np.cos(theta), -np.sin(theta)],
23                    [0, np.sin(theta), np.cos(theta)]])

```

Lista de código 1.2 – Funções para matrizes de rotação

Enquanto, a função para criar matrizes homogêneas combina as rotações e translações, conforme o Trecho de Código 1.3.

```

1 def create_homogeneous_matrix(position: np.ndarray, euler_angles: np.
2     ndarray) -> np.ndarray:
3
4     """
5         Cria uma matriz de transformação homogênea 4x4 a partir da posição e
6         ângulos de Euler.
7         Assume a convenção ZYX (Yaw, Pitch, Roll).
8     """
9
10    # Atribui os ângulos de Euler (rx, ry, rz) aos seus respectivos
11    # nomes
12    roll = euler_angles[0]      # Rotação em torno de X
13    pitch = euler_angles[1]     # Rotação em torno de Y
14    yaw = euler_angles[2]       # Rotação em torno de Z
15
16    # Constrói a matriz de rotação usando a convenção ZYX (Yaw-Pitch-
17    # Roll)
18    R = Rz(yaw) @ Ry(pitch) @ Rx(roll)
19
20    # Monta a matriz de transformação homogênea 4x4
21    T = np.eye(4)
22    T[:3, :3] = R
23    T[:3, 3] = position
24
25    return T

```

Lista de código 1.3 – Função para criar matriz homogênea

Por fim, a inversão eficiente de uma matriz homogênea foi implementada utilizando a estrutura especial desse tipo de matriz, conforme o Trecho de Código 1.4.

```

1 def invert_homogeneous_matrix(T: np.ndarray) -> np.ndarray:
2     """
3         Efficiently invert a 4x4 homogeneous transformation matrix.
4         T^-1 = [[R^T, -R^T * p], [0, 1]]
5     """
6     R = T[:3, :3]
7     P = T[:3, 3]
8
9     R_inv = R.T
10    P_inv = -R_inv @ P
11
12    T_inv = np.eye(4)
13    T_inv[:3, :3] = R_inv
14    T_inv[:3, 3] = P_inv
15
16    return T_inv

```

Lista de código 1.4 – Função para inverter matriz homogênea.

Não adicionamos todos os trechos de códigos para esta solução, de modo a manter o documento mais legível. Aconselhamos a leitura diretamente no arquivo de `utils`.

1.5.1 Análise da Posição Inicial

Utilizando as funções implementadas (1.2, 1.3, 1.4...), foi realizada a análise da pose inicial do robô na cena da Figura 1.

A pose do robô foi obtida e apresentada em termos de posição e orientação, conforme o Trecho de Código 1.5.

```

1 # Verificar pose inicial do robô
2 robot_pose = connector.get_object_pose('Robot')
3
4 if robot_pose:
5     position, orientation = robot_pose
6     print("Pose inicial do robô:")
7     print(f"Posição (x, y, z): [{position[0]:.3f}, {position[1]:.3f}, {position[2]:.3f}] metros")
8     print(f"Orientação (rx, ry, rz): [{orientation[0]:.3f}, {orientation[1]:.3f}, {orientation[2]:.3f}] radianos")
9
10    # Converter para graus para melhor visualização
11    orientation_deg = np.rad2deg(orientation)

```

```

12     print(f"Orientação (rx, ry, rz): [{orientation_deg[0]:.1f}°, {orientation_deg[1]:.1f}°, {orientation_deg[2]:.1f}°]")

```

Lista de código 1.5 – Obtenção da pose do robô.

Em seguida, foram calculadas e visualizadas as transformações entre o robô e todos os demais objetos na cena, conforme o Trecho de Código 1.6.

```

1 def analyze_scene_transformations(connector, object_handles,
2                                   scenario_name="Pose Inicial"):
3     """
4         Analisa e exibe as transformações entre objetos na cena.
5     """
6     print(f"\n==== Análise de Transformações - {scenario_name} ===")
7
8     # Criar analisador de cena
9     analyzer = SceneAnalyzer(connector)
10
11    # Obter lista de objetos (excluindo o robô)
12    object_names = [name for name in object_handles.keys() if name != 'Robot']
13
14    # Calcular poses relativas
15    relative_poses = analyzer.calculate_relative_poses('Robot',
16                                                       object_names)
17
18    print(f"\nTransformações calculadas para {len(relative_poses)} objetos:")
19
20    for obj_name, T_R_O in relative_poses.items():
21        # Extrair posição e orientação no frame do robô
22        pos_R = T_R_O[:3, 3]
23
24        print(f"\n{obj_name}:")
25        print(f"  Posição no frame do robô: [{pos_R[0]:.3f}, {pos_R[1]:.3f}, {pos_R[2]:.3f}] m")
26
27        # Validar matriz de transformação
28        is_valid = validate_transformation_matrix(T_R_O)
29        print(f"  Matriz válida: {'ok' if is_valid else 'nope'}")
30
31    # Gerar plot
32    analyzer.plot_scene_from_robot_perspective('Robot', object_names,
33                                              scenario_name)
34
35    return relative_poses

```

Lista de código 1.6 – Cálculo das transformações relativas.

Resultando no gráfico da Figura 3 apresentando essas transformações.

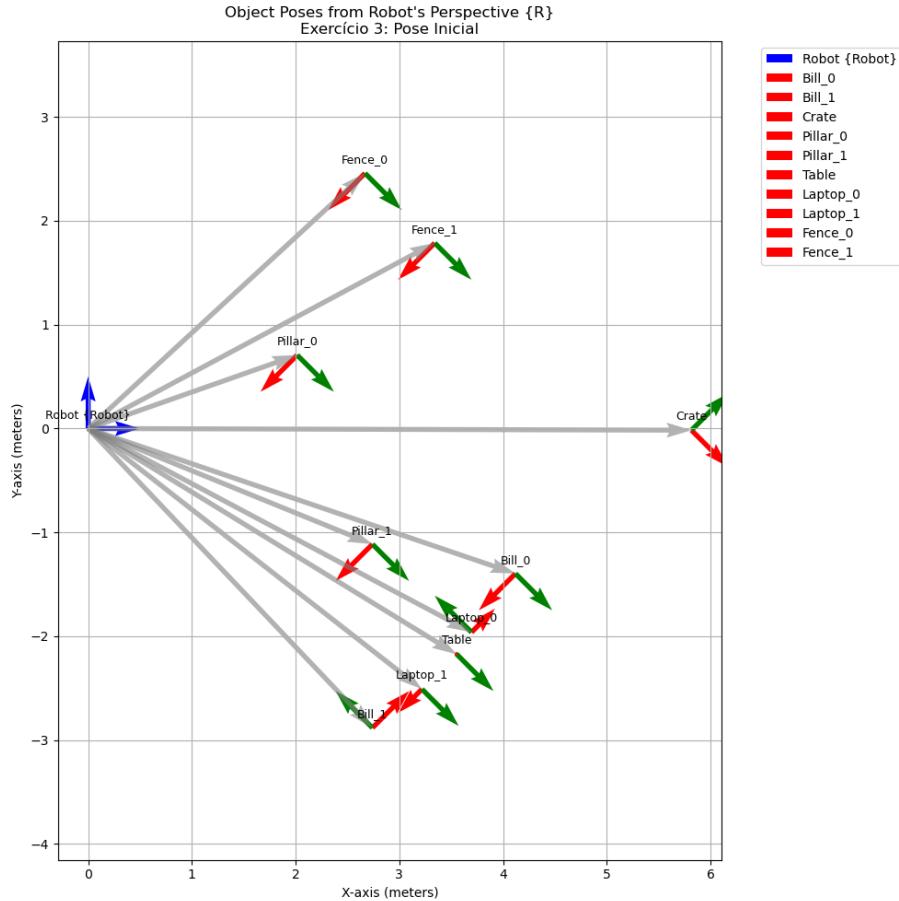


Figura 3 – Poses dos objetos da perspectiva do Robô.

1.6 Exercício 4: Múltiplas Posições do Robô

No quarto exercício, o robô foi posicionado em três localizações diferentes na cena para verificar que a implementação funcionava corretamente em diferentes configurações. As posições testadas foram:

1. Posição original (como mostrado na Figura 1);
2. Posição na coordenada (0, 0) (Cenário A, Figura 4);
3. Posição ao lado do objeto Bill_1 (Cenário B, Figura 5);
4. Posição com a traseira voltada para o objeto Crate (Cenário C, Figura 6).

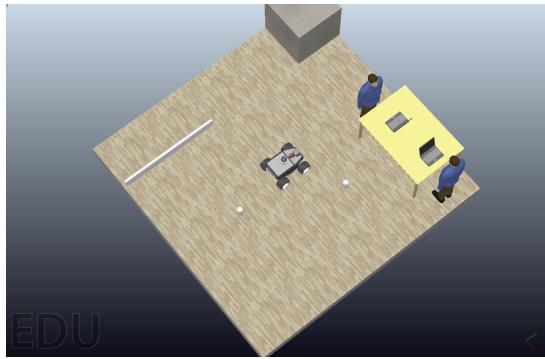


Figura 4 – Cenário A: Posição na coordenada (0, 0).

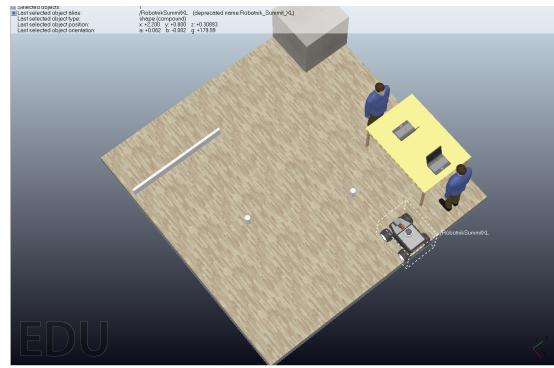


Figura 5 – Cenário B: Posição ao lado do Bill_1.

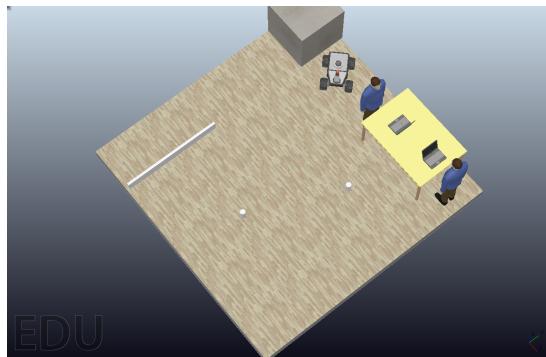


Figura 6 – Cenário C: Posição com traseira para o Crate.

Figura 7 – Diferentes posições do robô utilizadas para validação das transformações.

Em cada uma dessas posições, foram calculadas as matrizes de transformação homogêneas entre o robô e todos os objetos, demonstrando a corretude da implementação independentemente da localização do robô na cena, conforme o Trecho de Código 1.7:

```

1 # Cenário A
2 wait_for_user_input("Pressione Enter após mover o robô para a x0 y0...")
3 scenario_a_poses = analyze_scene_transformations(connector,
4     object_handles, "Cenário A: 0 0")
5
5 # Cenário B
6 wait_for_user_input("Pressione Enter após mover o robô para a posição de
7     lado para bill_1...")
7 scenario_b_poses = analyze_scene_transformations(connector,
8     object_handles, "Cenário B: Posição de lado para bill_1")
8
9 # Cenário C
10 wait_for_user_input("Pressione Enter após mover o robô para a posição
11     com a traseira para Crate)...")
11 scenario_c_poses = analyze_scene_transformations(connector,
```

```
object_handles, "Cenário C: Posição Traseira para Crate)")
```

Lista de código 1.7 – Análise de transformações em diferentes cenários.

Em cada cenário, foram gerados gráficos visualizando as posições relativas dos objetos a partir do referencial do robô, confirmando que o sistema de transformações funciona corretamente para qualquer configuração da cena. Por exemplo, o gráfico da Figura 8 representando o Cenário A: 0 0 e Figura 9. Os demais gráficos podem ser encontrados nos arquivos fontes disponibilizados juntamente com este documento.

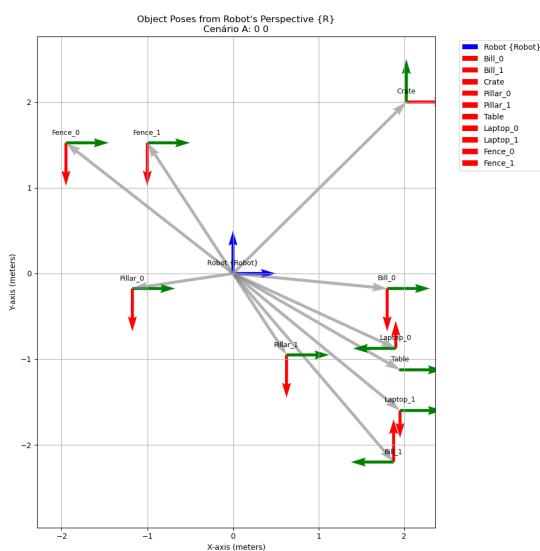


Figura 8 – Poses dos objetos da perspectiva do Robô no Cenário A: 0 0.

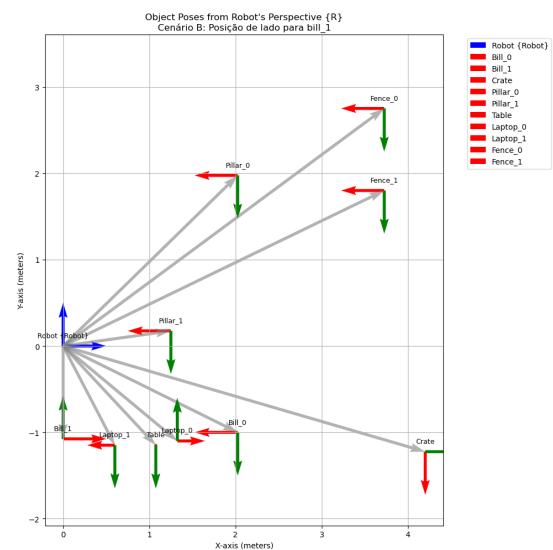


Figura 9 – || Cenário B: Posição de lado para bill_1.

1.7 Exercício 5: Transformação de Dados do Laser para o Referencial Global

No quinto exercício, o foco foi a integração do sensor laser (Hokuyo) ao robô Pioneer P3DX e a transformação dos dados do sensor do referencial local para o referencial global do mundo.

1.7.1 Atualização do Cenário

Para os exercícios 5 e 6, foi necessário modificar o cenário devido a problemas com os pilares originais, que causavam erros de *Crashing* no simulador. Portanto, a cena para os próximos exercícios pode ser visto na Figura 10.

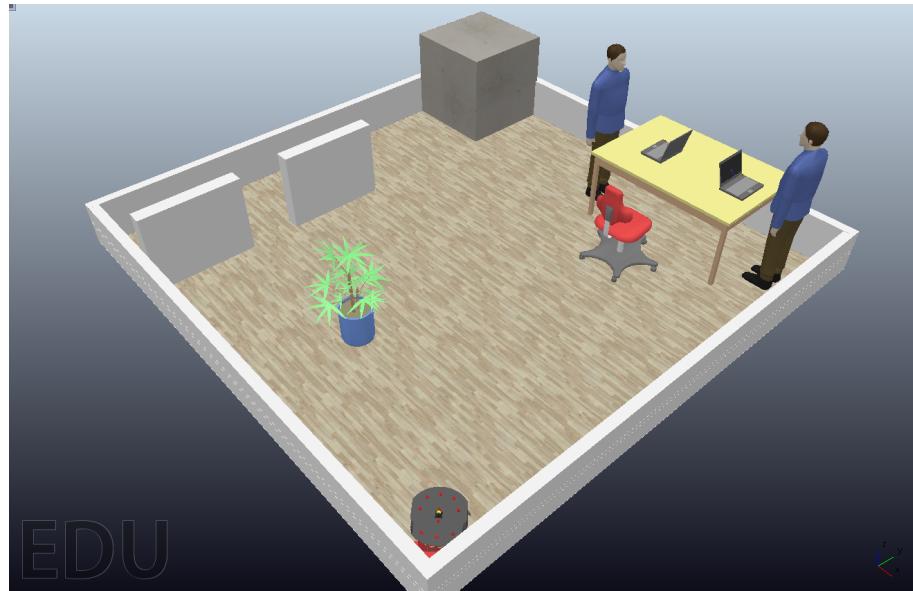


Figura 10 – Cenário modificado para os exercícios 5-6, com o robô Pioneer P3DX e sensor laser Hokuyo.

As principais alterações incluíram:

- Remoção dos pilares problemáticos;
- Adição de novos elementos (cadeiras, plantas, paredes);
- Aumento da altura das paredes para melhor detecção pelo laser;
- Atualização do mapeamento de objetos.

O novo mapeamento de objetos foi definido, conforme o Trecho de Código 1.8.

```

1 DEFAULT_OBJECT_MAPPING_EX5_6 = {
2     'Robot': 'PioneerP3DX',
3 ...
4     'SwivelChair': 'swivelChair',
5     'Plant': 'indoorPlant',
6     'Hokuyo': 'PioneerP3DX/fastHokuyo',
7 ...
8 }
```

Lista de código 1.8 – Mapeamento de objetos atualizado para exercícios 5-6.

1.7.2 Transformações entre Referenciais

Para transformar os dados do laser do seu referencial local para o referencial global, foram definidas as seguintes transformações:

1. R_L^T (laser → robô): Transformação do referencial do laser para o referencial do robô;

2. ${}^W_R T$ (robô → mundo): Transformação do referencial do robô para o referencial global;
3. ${}^L W T = {}^W_R T \cdot {}^R_L T$: Transformação completa do referencial do laser para o referencial global.

A implementação desta transformação foi realizada através da função do Trecho de Código 1.9.

```

1 def transform_laser_data_to_global_frame(sim, laser_data, robot_handle,
2                                         hokuyo_handle):
3     """
4         Transforma os dados do laser do referencial local para o referencial
5         global.
6     """
7     # Obter pose do laser em relação ao robô
8     laser_pos_robot = sim.getObjectPosition(hokuyo_handle, robot_handle)
9     laser_orient_robot = sim.getObjectOrientation(hokuyo_handle,
10                                                 robot_handle)
11
12     # Criar matriz de transformação do laser para o robô (R_T_L)
13     R_T_L = create_homogeneous_matrix(
14         np.array(laser_pos_robot),
15         np.array(laser_orient_robot)
16     )
17
18     # Obter pose do robô no referencial global
19     robot_pos_world = sim.getObjectPosition(robot_handle, sim.
20                                             handle_world)
21     robot_orient_world = sim.getObjectOrientation(robot_handle, sim.
22                                                 handle_world)
23
24     # Criar matriz de transformação do robô para o mundo (W_T_R)
25     W_T_R = create_homogeneous_matrix(
26         np.array(robot_pos_world),
27         np.array(robot_orient_world)
28     )
29
30     # Transformação completa do laser para o mundo (W_T_L = W_T_R *
31     R_T_L)
32     W_T_L = W_T_R @ R_T_L
33
34     # Converter dados do laser para pontos 3D no referencial do laser
35     points_in_laser_frame = []
36     for angle, distance in laser_data:
37         # Converter de coordenadas polares para cartesianas no plano xy
38         do laser
39             x = distance * np.cos(angle)
40             y = distance * np.sin(angle)

```

```

34     z = 0 # O laser está no plano xy
35
36     # Ponto homogêneo no referencial do laser [x, y, z, 1]
37     point_laser = np.array([x, y, z, 1])
38
39     # Transformar para o referencial global
40     point_global = W_T_L @ point_laser
41
42     # Armazenar as coordenadas x, y, z
43     points_in_global_frame = point_global[:3]
44     points_in_laser_frame.append(points_in_global_frame)
45
46 return np.array(points_in_laser_frame)

```

Lista de código 1.9 – Função para transformar dados do laser para o referencial global.

1.7.3 Visualização dos Dados Transformados

Para visualizar os resultados, foram implementadas funções para plotar os dados do laser tanto no referencial local quanto no referencial global, reutilizando as funções do notebook da aula03, conforme o Trecho de Código 1.10.

```

1 def plot_laser_data_global(global_points, robot_pos_global, max_range
2 =10):
3     """
4         Plota os dados do laser no referencial global.
5     """
6     fig = plt.figure(figsize=(10, 10))
7     ax = fig.add_subplot(111, aspect='equal')
8     ax.set_title("Dados do Laser no Referencial Global")
9     ax.set_xlabel("X (metros)")
10    ax.set_ylabel("Y (metros)")
11
12    # Plotar os pontos do laser
13    ax.scatter(global_points[:, 0], global_points[:, 1], c='r', marker='.',
14               label='Pontos do Laser')
15
16    # Plotar a posição do robô
17    ax.plot(robot_pos_global[0], robot_pos_global[1], 'bo',
18            markersize=10, label='Robô')
19
20    ax.grid(True)
21    ax.legend()
22    ax.set_xlim([robot_pos_global[0] - max_range, robot_pos_global[0] +
23                max_range])
24    ax.set_ylim([robot_pos_global[1] - max_range, robot_pos_global[1] +
25                max_range])

```

```

21
22     plt.show()

```

Lista de código 1.10 – Função para plotar dados do laser no referencial global.

Os gráficos resultantes para os Dados do Laser no Referencial Global podem ser vistos na Figura 11. Enquanto os dados do laser após uma movimentação podem ser visualizados no gráfico da Figura 12. Por fim, os demais gráficos podem ser encontrados no arquivo fonte.

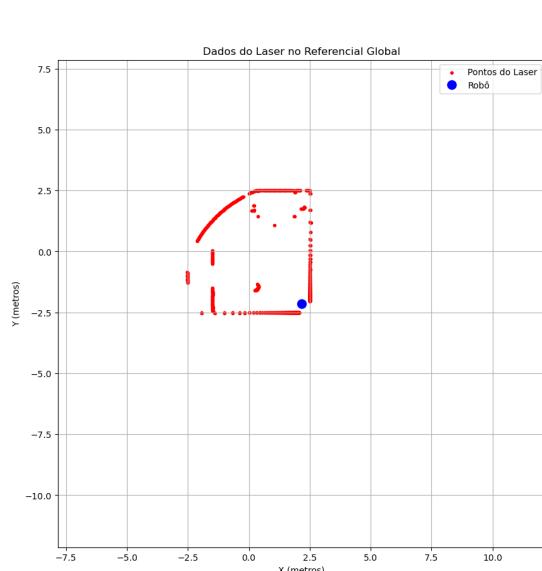


Figura 11 – Dados do Laser no Referencial Global.

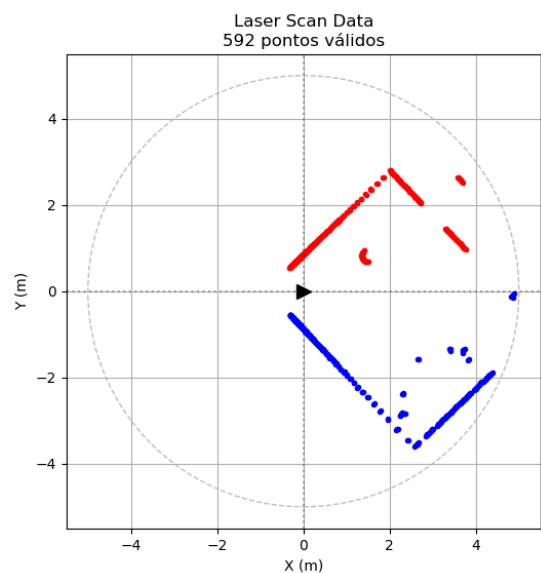


Figura 12 – Posição 2 (Após movimento).

1.8 Exercício 6: Navegação Autônoma e Mapeamento Incremental

O sexto e último exercício combinou os conceitos anteriores para implementar uma navegação autônoma básica do robô, enquanto realizava um mapeamento incremental do ambiente utilizando os dados do sensor laser. Sabendo disso, a função principal para realizar o mapeamento incremental pode ser vista no Trecho de Código 1.11.

```

1 def plot_incremental_map(robot_trajectory, all_laser_points):
2     """
3         Cria um plot incremental mostrando a trajetória do robô e todos os
4         pontos do laser.
5     """
6     fig = plt.figure(figsize=(12, 12))
7     ax = fig.add_subplot(111, aspect='equal')
8     ax.set_title("Mapeamento Incremental - Trajetória do Robô e Pontos
9         do Laser")
10    ax.set_xlabel("X (metros)")
11    ax.set_ylabel("Y (metros)")

```

```

10
11     # Converter trajetória para arrays
12     traj_x = [pos[0] for pos in robot_trajectory]
13     traj_y = [pos[1] for pos in robot_trajectory]
14
15     # Plotar a trajetória do robô como uma linha tracejada
16     ax.plot(traj_x, traj_y, 'b--', linewidth=2, label='Trajetória do Robô')
17
18     # Plotar pontos iniciais e finais da trajetória
19     ax.plot(traj_x[0], traj_y[0], 'go', markersize=8, label='Posição Inicial')
20     ax.plot(traj_x[-1], traj_y[-1], 'ro', markersize=8, label='Posição Final')
21
22     # Plotar todos os pontos do laser (mapa combinado)
23     all_points = np.vstack(all_laser_points)
24     ax.scatter(all_points[:, 0], all_points[:, 1], c='r', marker='.', s=2, alpha=0.6, label='Leituras do Laser')
25
26     ax.grid(True)
27     ax.legend()
28
29     # Ajustar os limites para cobrir toda a área
30     min_x = min(np.min(all_points[:, 0]), np.min(traj_x)) - 1
31     max_x = max(np.max(all_points[:, 0]), np.max(traj_x)) + 1
32     min_y = min(np.min(all_points[:, 1]), np.min(traj_y)) - 1
33     max_y = max(np.max(all_points[:, 1]), np.max(traj_y)) + 1
34
35     ax.set_xlim([min_x, max_x])
36     ax.set_ylim([min_y, max_y])
37
38     plt.show()
39
40     return fig, ax

```

Lista de código 1.11 – Função para plotar mapa incremental.

Por fim, foi implementado um algoritmo simplificado de navegação autônoma com desvio de obstáculos, seguindo o código base do notebook da aula03, conforme o Trecho de Código 1.12.

```

1 # Identificar pontos de interesse no laser
2 frente_idx = min(int(len(laser_data) / 2), len(laser_data) - 1)
3 direita_idx = min(int(len(laser_data) * 1 / 4), len(laser_data) - 1)
4 esquerda_idx = min(int(len(laser_data) * 3 / 4), len(laser_data) - 1)
5
6 # Obter distâncias em direções específicas

```

```

7 dist_frente = laser_data[frente_idx, 1] if frente_idx < len(laser_data)
     else 5.0
8 dist_direita = laser_data[direita_idx, 1] if direita_idx < len(
     laser_data) else 5.0
9 dist_esquerda = laser_data[esquerda_idx, 1] if esquerda_idx < len(
     laser_data) else 5.0
10
11 # Lógica de navegação com desvio de obstáculos
12 threshold_dist = 1.5 # distância limiar para detecção de obstáculo (m)
13
14 if dist_frente > threshold_dist:
15     # Caminho livre à frente
16     v = 0.3
17     w = 0
18
19     # Ajuste fino de direção se há obstáculo próximo
20     if dist_direita < threshold_dist and dist_esquerda > threshold_dist:
21         # Obstáculo à direita, ajuste suave para a esquerda
22         w = 0.2
23     elif dist_esquerda < threshold_dist and dist_direita >
threshold_dist:
24         # Obstáculo à esquerda, ajuste suave para a direita
25         w = -0.2
26 elif dist_direita > dist_esquerda:
27     # Obstáculo à frente, mas espaço à direita
28     v = 0.1
29     w = -0.5 # girar à direita
30 else:
31     # Obstáculo à frente, mas espaço à esquerda
32     v = 0.1
33     w = 0.5 # girar à esquerda
34
35 # Converter para velocidades das rodas usando o modelo cinemático
36 wl = v / r - (w * L) / (2 * r)
37 wr = v / r + (w * L) / (2 * r)

```

Lista de código 1.12 – Algoritmo de navegação autônoma com desvio de obstáculos.

Durante a navegação, o algoritmo realizou as seguintes etapas para o mapeamento incremental:

1. Armazenamento da trajetória do robô (sequência de posições);
2. Captura de dados do sensor laser em cada passo;
3. Transformação dos pontos do laser para o referencial global;
4. Acumulação de todos os pontos em uma única representação;

5. Visualização da trajetória e do mapa resultante.

Resultando, assim, no gráfico da Figura 13.

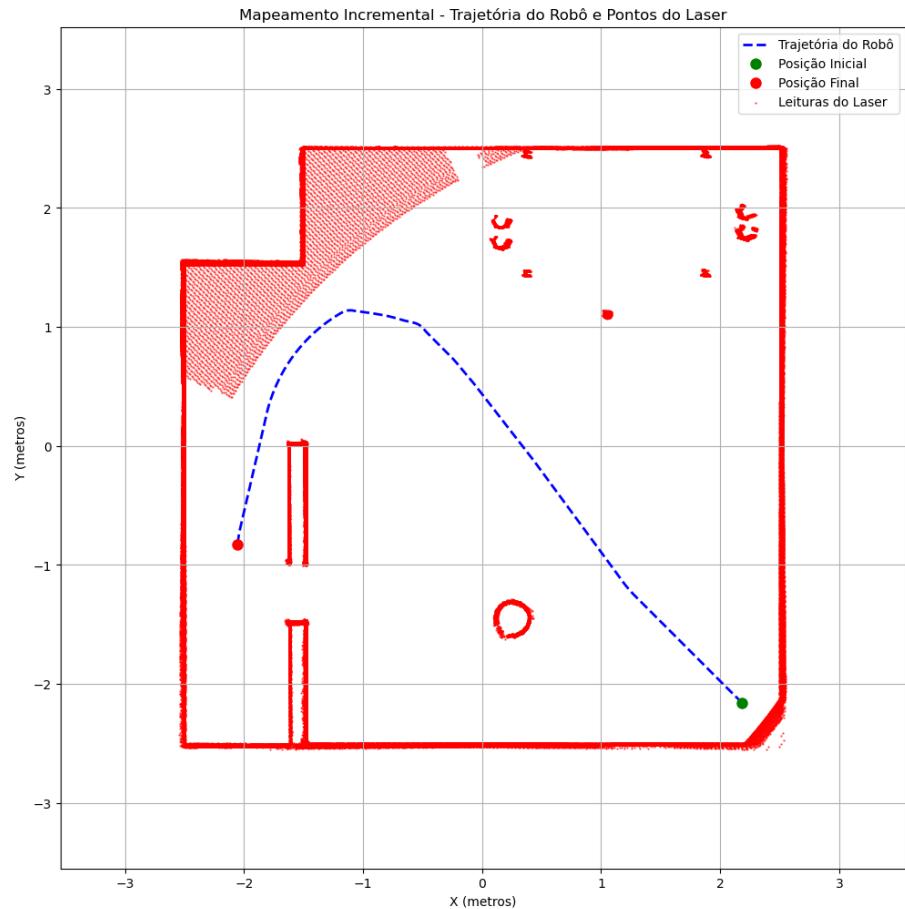


Figura 13 – Mapa incremental com o caminho executado pelo robô.

2 Conclusão

Este trabalho prático proporcionou uma compreensão aprofundada sobre sistemas de coordenadas, transformações homogêneas, integração de sensores e navegação básica de robôs móveis.

2.1 Resultados Obtidos

Através deste trabalho prático, foram implementados com sucesso todos os exercícios propostos, os quais possibilitaram:

1. **Criação de Cena:** ambiente de simulação com robô e múltiplos objetos;
2. **Diagrama de Transformações:** representação visual das relações entre frames;
3. **Matrizes Homogêneas:** implementação e validação de transformações homogêneas;
4. **Múltiplas Posições:** verificação do funcionamento em diferentes configurações;
5. **Transformação de Dados de Laser:** conversão entre referenciais local e global;
6. **Navegação e Mapeamento:** algoritmo de navegação autônoma com mapeamento incremental.

Os resultados demonstram a correta aplicação dos conceitos de transformações homogêneas e integração de sensores em um ambiente de simulação.

2.2 Dificuldades Enfrentadas

Durante o desenvolvimento do trabalho, foram enfrentadas algumas dificuldades:

1. **Problemas com a Cena do CoppeliaSim:** os pilares usados na primeira cena causavam erros para execução da simulação, exigindo a modificação do cenário para os exercícios 5-6;
2. **Acesso aos Sensores:** o acesso aos sensores de visão requer um conhecimento específico da API do CoppeliaSim, o que levou à reutilização de código fornecido pelo professor para interagir com o sensor laser Hokuyo;

3. **Integração entre Matrizes de Transformação:** garantir a consistência nas multiplicações de matrizes homogêneas entre os diferentes referenciais exigiu atenção especial à ordem das operações, às convenções de rotação adotadas e testes massivos.

Referências

Anaconda, Inc. **Miniconda – Getting Started.** [S.l.], 2025. Documentação para a distribuição que inclui Python 3.13.7. Disponível em: <https://www.anaconda.com/docs/getting-started/miniconda/main>. Acesso em: 10 set. 2025. Citado na página 5.

HINTJENS, P. **ZeroMQ: Messaging for Many Applications.** [S.l.: s.n.]: O'Reilly Media, Inc., 2013. ISBN 978-1449334059. Citado na página 5.

JGraph Ltd. **diagrams.net (formerly Draw.io).** 2025. Disponível em: <https://www.diagrams.net/>. Acesso em: 10 set. 2025. Citado na página 5.

MACHARET, D. G. **Robótica Móvel - Transformações homogêneas e Espaço de configurações.** 2025. Material de aula, Departamento de Ciência da Computação, Universidade Federal de Minas Gerais. Disponível online ou via material do curso. Citado na página 9.

Microsoft Corporation. **Visual Studio Code.** 2025. Disponível em: <https://code.visualstudio.com/>. Acesso em: 10 set. 2025. Citado na página 5.

ROBOTICS, C. **CoppeliaSim User Manual.** [S.l.], 2023. Disponível em: <https://www.coppeliarobotics.com/helpFiles/>. Acesso em: 8 set. 2025. Citado na página 5.