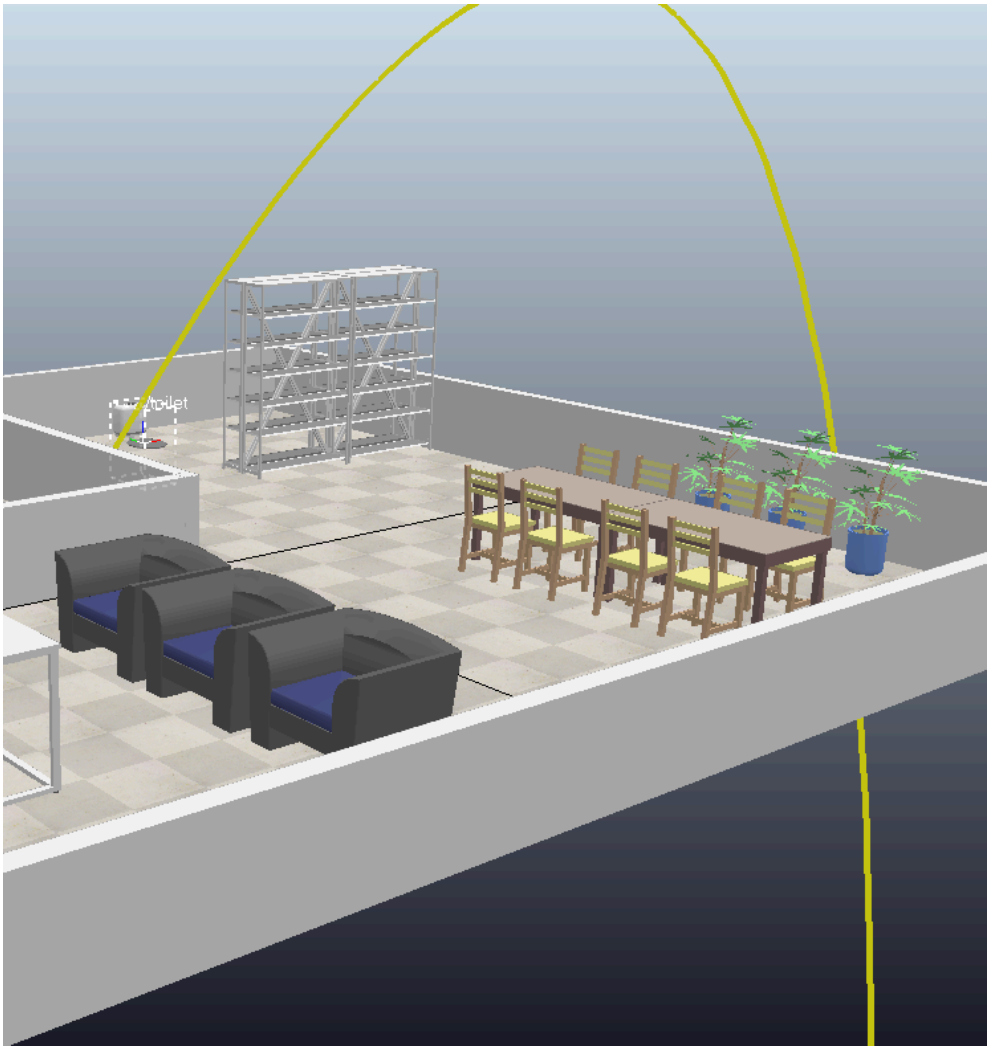
UNIVERSIDADE FEDERAL
DE MINAS GERAIS



TRABALHO PRÁTICO 3
EXPLORAÇÃO E MAPEAMENTO

ALUNO:
FELIPPE VELOSO MARINHO
MATRÍCULA: 2021072260

DISCIPLINA: ROBÓTICA MÓVEL

SUMÁRIO

1 Introdução

O presente trabalho tem como objetivo documentar a implementação e familiarização com os conceitos discutidos na disciplina de Robótica Móvel. Nele através da utilização do software CoppeliaSim são apresentados conceitos de aplicação do algoritmo de Occupancy Grid em diferentes cenários, um estático e um dinâmico. Para a exploração do ambiente também é demonstrada a aplicação do modelo cinemático utilizado no modelo de robô diferencial dado como objeto de manipulação.

2 Execução

Para a execução do código, é necessário ter o software CoppeliaSim instalado e configurado. Além disso, as seguintes bibliotecas Python são utilizadas: numpy, matplotlib, scipy, e a API do CoppeliaSim (sim.py). O código está organizado em três arquivos principais:

- Um notebook para a execução de códigos auxiliares e implementação do código principal. (dinamico.ipynb)
- Dois arquivos .py utilizados como bibliotecas, contendo funções de navegação em campos potenciais e uma classe para iniciar as funções do coppelia sim de maneira organizada. (PathPlanners.py e RobotControllers.py)

3 Navegação

A estratégia de navegação utilizada foi a de campos potenciais, de maneira semelhante à aplicada no Trabalho Prático 2 (Planejamento e Navegação). O algoritmo de campos potenciais calcula o campo vetorial resultante entre as forças de atração (em direção ao objetivo) e repulsão (em direção aos obstáculos). Devido à complexidade do ambiente com múltiplos obstáculos, as constantes definidas para força de repulsão foram ajustadas para serem bem baixas, comparadas às utilizadas em trabalhos anteriores:

- Constante de atração (katt) = 5
- Constante de repulsão (krep) = 0.005
- Raio de influência dos obstáculos (R) = 5

O laser Hokuyo contido no Kobuki é utilizado para detectar obstáculos ao longo do caminho, permitindo ao robô navegar de maneira reativa no ambiente não conhecido.

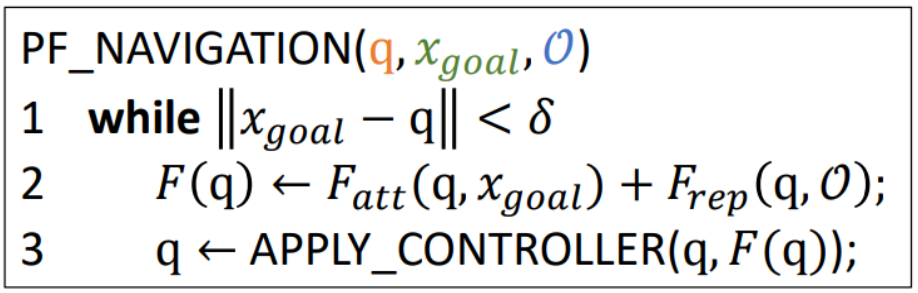


Figura 1 - Pseudocódigo do algoritmo de navegação com campos potenciais

O robô utilizado neste trabalho foi o robô diferencial, Kobuki, com o controle De Luca e Oriolo com o modelo Unicycle, semelhante a utilizada no Robotino anteriormente.

$$\begin{bmatrix} \dot{x} \\ \dot{y} \\ \dot{\theta} \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 \\ \sin\theta & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}$$

Figura 2 - Modelo Cinemático Unicycle

Este modelo cinemático é utilizado para calcular as velocidades das rodas do robô com base nas forças resultantes de atração e repulsão dos obstáculos. A abordagem reativa foi feita justamente para a adaptação ao ambiente dinâmico (não conhecido).

As constantes do robô utilizadas foram tiradas do enunciado do trabalho.

- L = 0.230
- r = 0.035

4 Implementação

Neste tópico serão abordadas de maneira mais abrangente a implementação em código do trecho responsável pela navegação do robô e seu funcionamento. Além disso, será destrinchada a implementação do algoritmo de Occupancy Grid.

4.1 Navegação

A navegação do robô é feita através de uma série de transformações entre referenciais globais e o grid de ocupação. O robô utiliza o sensor de laser Hokuyo para detectar obstáculos no ambiente e ajusta sua trajetória com base no algoritmo de campos potenciais. A estratégia de navegação inclui a transformação das coordenadas dos pontos detectados pelo laser para o referencial do robô e, em seguida, para o referencial global. A atualização do grid de ocupação é feita com base nessas leituras.

4.1.1 Conversões de referenciais

Para as conversões dos referenciais do robô para os referenciais necessários foram utilizadas as duas funções abaixo.

```python
# Transformação para o referencial do robô:
def Rz(theta):
""" Gera a matriz de rotação no plano Z.
Parâmetro:
- theta: Ângulo de rotação.
Retorna:
- Matriz de rotação 4x4.
```

```
        """
        return np.array([ [np.cos(theta), -np.sin(theta), 0, 0],
[np.sin(theta), np.cos(theta), 0, 0], [0, 0, 1, 0], [0, 0, 0, 1] ])

        # Funções de transformação e conversão genérica
        def transformacao(robot_pos, robot_ori):
            """
            Calcula a matriz de transformação homogénea para uma dada posição
e orientação do robô.

            """
            # Matriz de rotação em Z
            c, s = np.cos(robot_ori[2]), np.sin(robot_ori[2])
            Rz = np.array([[c, -s, 0], [s, c, 0], [0, 0, 1]])

            # Matriz de translação
            T = np.array([robot_pos[0], robot_pos[1], 0])

            # Matriz de transformação homogênea
            H = np.eye(4)
            H[:3, :3] = Rz
            H[:3, 3] = T
            return H
```

Por fim, foi necessário a conversão das coordenadas do mundo para o grid de ocupação.

Python

```
        # Conversão das coordenadas do mundo para o grid:
        def world_to_grid(world_point, map_size, cell_size):
            """
            Converte coordenadas do mundo para coordenadas do grid

            Parâmetros:
            world_point (np.array): Coordenadas do mundo
            map_size (int): Tamanho do mapa
            cell_size (int): Tamanho da célula do grid
            """
            grid_point = np.floor((world_point[:2] + map_size / 2) /
cell_size).astype(int)
            return grid_point
```

4.1.2 Loop de navegação principal

No loop de navegação é feita a detecção dos obstáculos, atualize seu mapa interno (grid de ocupação), e ajuste sua trajetória conforme necessário. Cada leitura do laser é processada para transformar as coordenadas do referencial do robô para o referencial global, verificando se as células do grid estão ocupadas, e marcando células vazias quando nenhum obstáculo é detectado de acordo com o inverse sensor model.

Python

```
                while True:
        # Capturo a posição e orientação do robô
                    robot_pos = robot_controller.get_robot_position()
                    robot_ori = robot_controller.get_robot_orientation()

                    distance_goal = np.linalg.norm(np.array(goal_position[:2])
- np.array(robot_pos[:2]))
                    if distance_goal <= 0.5:
                        break

                    if laser_range_data is not None and laser_angle_data is
not None:
                        raw_range_data, raw_angle_data = readSensorData()
                    # Transformo o laser para o referencial global
                    laser_data = np.array([raw_angle_data, raw_range_data]).T
                    laser_transform = Rz(np.pi / 2)
                     # Adiciono o ruído pedido no enunciado
                    noisy_data = get_all_laser_data_ruido(laser_data,
laser_transform)

                    robot_cell = world_to_grid(np.array(robot_pos), map_size,
cell_size)
                    Hlw = transformacao(robot_pos, robot_ori)

                    obs = []
                    obs_pts = []
                    obstacle_detected = False

                    for i in range(len(laser_data)):
                        ang, dist = laser_data[i]

                        if (5 - dist) > 0.1:
                            x = dist * np.cos(ang)
                            y = dist * np.sin(ang)
                            point = np.array([x, y, 0, 1])

                            if len(Hlw) != 0: # Converto os pontos do laser
para o referencial global
                                point = Hlw @ point
                                obs.append(point[0:2])
                                obs_pts.append(point)
```

```
                                grid_point = world_to_grid(point, map_size,
cell_size) # Converto os pontos do laser para células válidas
                                if 0 <= grid_point[0] < rows and 0 <=
grid_point[1] < cols:
                                    occupancy_grid =
occupancy_grid_mapping(robot_cell, grid_point, occupancy_grid)
                                    obstacle_detected = True

                                laser_points.append(point[:2])

                    if not obstacle_detected:
                        empty_cell = robot_cell + np.array([5 / cell_size, 0])
                        occupancy_grid = occupancy_grid_mapping(robot_cell,
empty_cell, occupancy_grid)
```

4.2 Occupancy Grid

O occupancy grid é uma matriz que representa a probabilidade de cada célula estar ocupada. Este grid é atualizado em tempo real com base nas leituras do sensor de laser, utilizando o modelo de sensor inverso.

Para isso, o uso de log-odds facilita a fusão de múltiplas leituras sensoriais ao longo do tempo.

$$l(\mathrm{A}) = \log(o(A)) = \log\left(\frac{p(A)}{p(\neg A)}\right)$$

Figura 3 - Cálculo do Log-odds

A regra de atualização do occupancy grid funciona e a célula mi está no campo perceptual do sensor, o log-odds l{t,i} é atualizado somando o resultado do modelo inverso do sensor (inverse_sensor_model) aplicado à célula mi, à posição do robô xt e às leituras do sensor zt, subtraindo o valor do log-odds inicial lo.

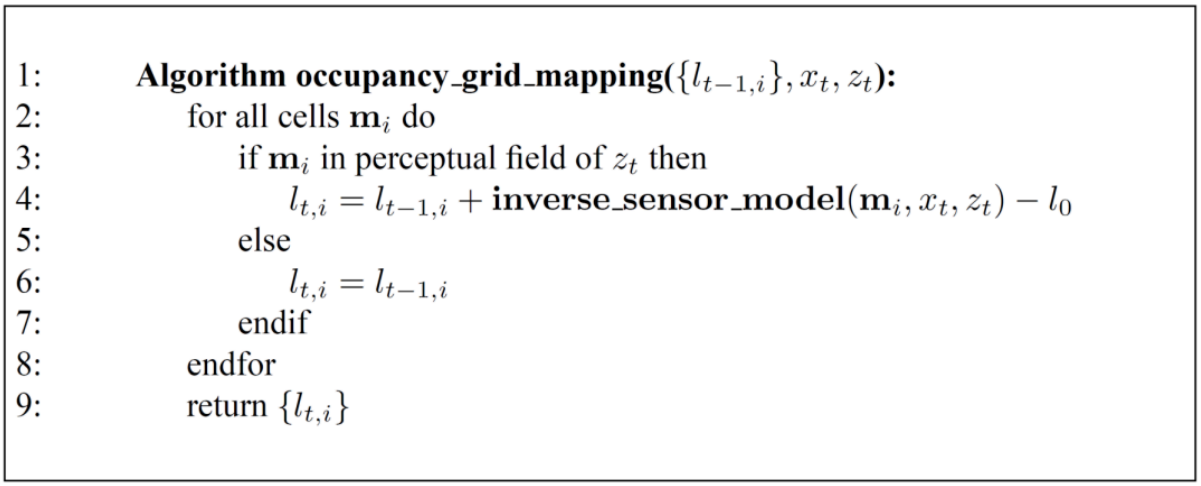


Figura 4 - Pseudocódigo do Algoritmo de Occupancy Grid Mapping

O modelo de sensor inverso presente no algoritmo de mapeamento é usado para calcular a probabilidade de ocupação de uma célula com base na leitura do laser.

No código abaixo é possível ver a implementação onde os valores de p_occ e p_free foram definidos empiricamente. Seu funcionamento consiste em calcular as chances de acordo com os valores definidos e:

- Se a célula está fora do alcance do sensor, retorna o valor inicial de log odds.
- Se a célula está ocupada, retorna o log odds de ocupação
- Se a célula está livre, retorna o log odds de célula livre
- Se não, retorna o valor inicial de log odds

Dessa forma o grid é atualizado corretamente.

Python

```
        def inverse_sensor_model(m_i, x_t, z_t, l0):
            max_distance = 5 / cell_size
            alpha = 0.2 / cell_size
            distance_to_cell = np.linalg.norm(m_i - x_t[:2])
            p_occ = 0.85 # Valor definido empíricamente após testes
            p_free = 0.1
            l_occ = log_odds(p_occ)
            l_free = log_odds(p_free)

            if distance_to_cell > min(max_distance, z_t + alpha / 2):
                return log_odds(l0)
            if z_t < max_distance and abs(distance_to_cell - z_t) <= alpha /
2:
                return l_occ
            if distance_to_cell <= z_t:
                return l_free

            return log_odds(l0)
```

5 Testes

Para a definição dos goals no mapa foram levadas em consideração pontos onde o funcionamento do algoritmo de campos potenciais não fosse prejudicado e pontos suficientes para cobrir boa parte do cenário. Sendo assim, foram utilizados os seguintes pontos:

- goal_positions = [[0.4, -4.45, 1], [0.8, -1.8, 1], [1.7, 0, 1], [0.7, 3, 1], [2.5, 4.3, 1]]

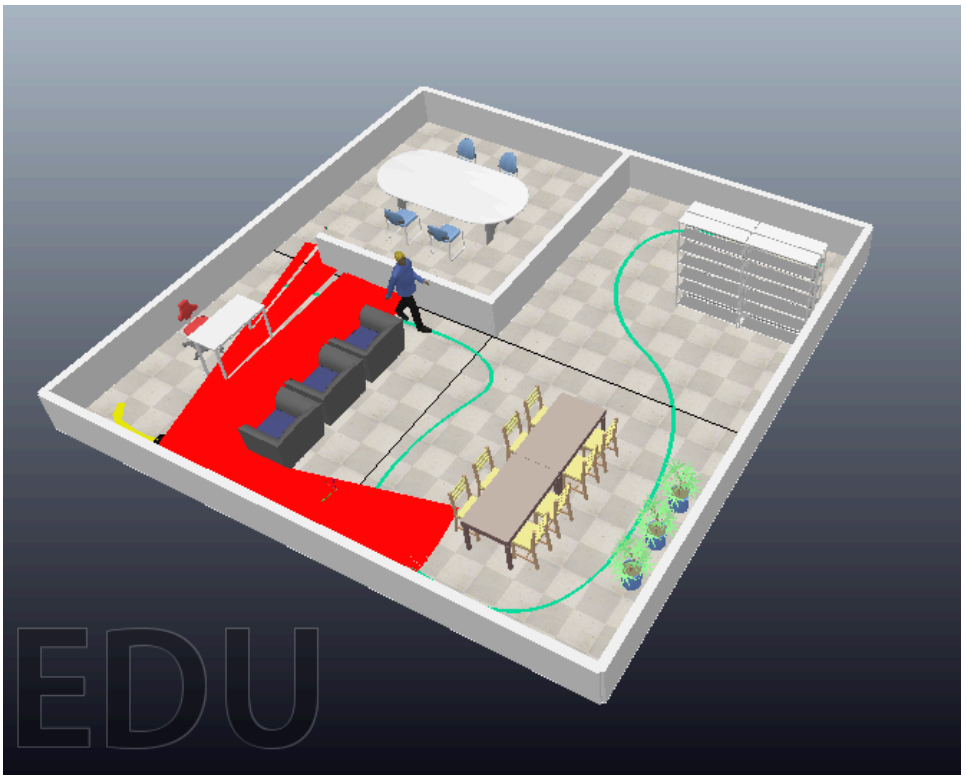
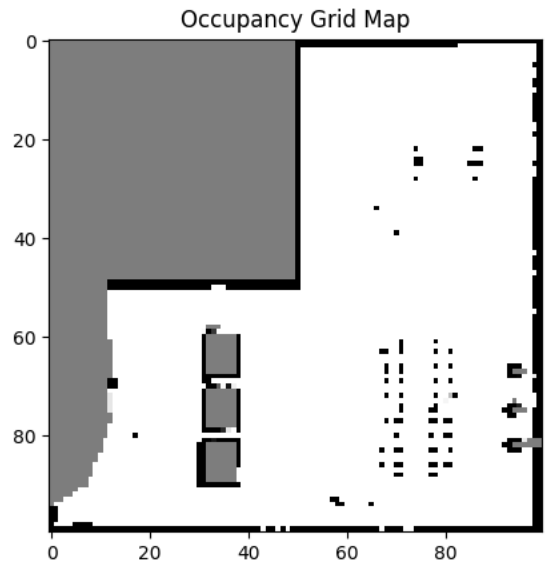
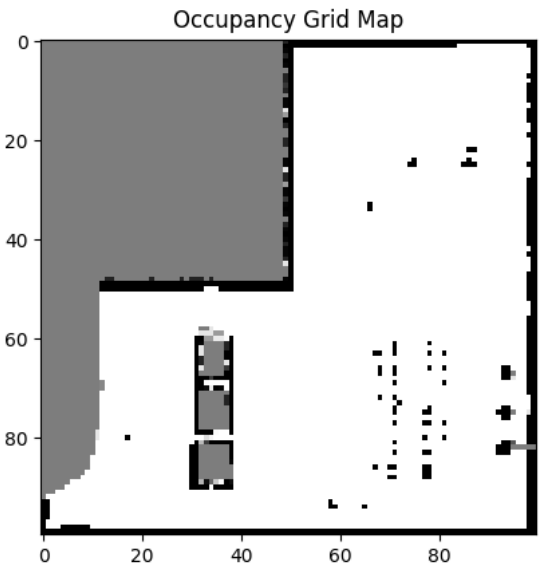


Figura 5 - Cenário dinâmico em execução

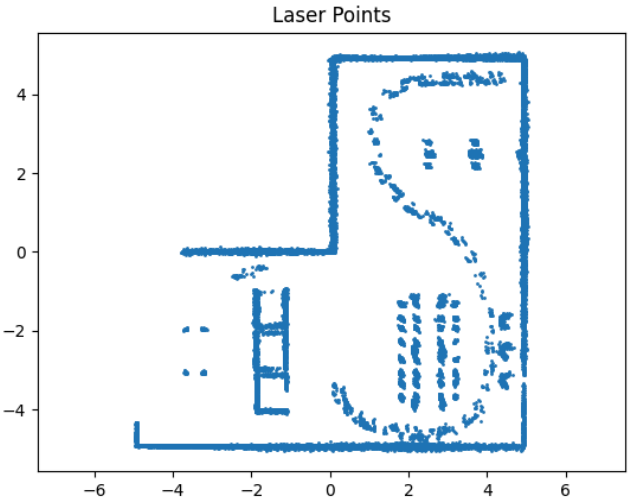
Após os testes no cenário estático, o código foi adaptado e foi escolhido como tamanho de célula 0., chegando assim a um resultado satisfatório e semelhante ao mostrado no enunciado. Onde à esquerda é exibido o Occupancy Grid Map comum e na direta o resultado com o ruído aplicado.

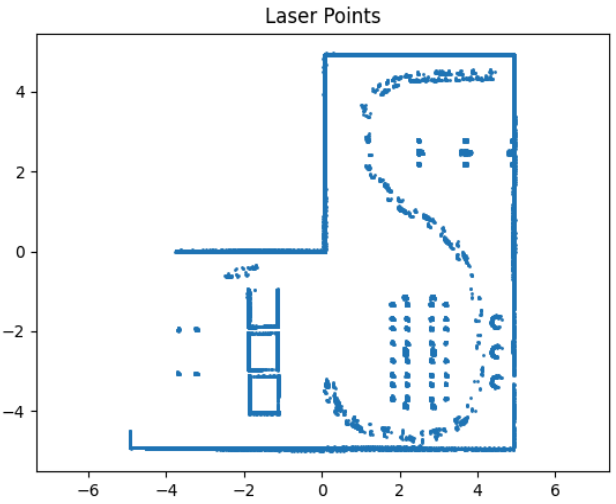




Figuras 6 e 7 - Grids de ocupação sem e com ruído

Como o teste foi feito no cenário dinâmico é possível comparar com os plots dos pontos do laser.





Figuras 8 e 9 - Plot dos pontos do laser sem e com ruído

6 Conclusão

Inicialmente, o foco se manteve no plot dos pontos ocupados e não ocupados, permitindo uma visualização clara dos dados do sensor. A partir dessa base, avançou-se para a inclusão de funcionalidades mais complexas, como a transformação de coordenadas, adição de ruído nos dados do laser, e a atualização do grid considerando obstáculos detectados e áreas desocupadas.

Como comentado anteriormente, durante os testes, identificamos que um tamanho de célula de 0.1 proporcionou resultados satisfatórios, equilibrando a precisão da representação do ambiente e a eficiência computacional. A implementação gradual, juntamente com a análise cuidadosa dos parâmetros, possibilitou a construção de um algoritmo robusto e funcional para mapear o ambiente utilizando dados de sensores laser. Em resumo, este trabalho não apenas consolidou nossos conhecimentos em planejamento e navegação, mas também nos proporcionou uma base sólida para explorar abordagens mais avançadas em robótica móvel

7 Bibliografia

Prof. Douglas G. Macharet - Robótica Móvel - Ferramental

Documentação zmqRemoteApi - regular API reference
regular API reference

Prof. Douglas G. Macharet - Robótica Móvel - Descrição espacial e Transformações rígidas

Prof. Douglas G. Macharet - Locomoção – Modelos cinemáticos

Prof. Douglas G. Macharet - Controle – Cinemático

Prof. Douglas G. Macharet - Locomoção – Modelos cinemáticos

Prof. Douglas G. Macharet - Paradigmas Robóticos

Prof. Douglas G. Macharet - Campos Potenciais

Prof. Douglas G. Macharet - Mapeamento Occupancy Grid

Robótica con Python - Robótica con Python y CoppeliaSim - YouTube

Kobuki Parameters - Parâmetros do Robô