PEL 215 - Tarefa 04 - Mapeamento de Ambientes

Discente: Fábio Rossatti Gianzanti

Docente: Prof. Dr. Flávio Tonidandel

O desafio consiste em realizar o mapeamento de um ambiente (não fornecido) utilizando o recurso de mapeamento probabilístico, por meio de um robot que utiliza qualquer sensor disponível no simulador webots.

Links para a visualização do vídeo demonstrativo:

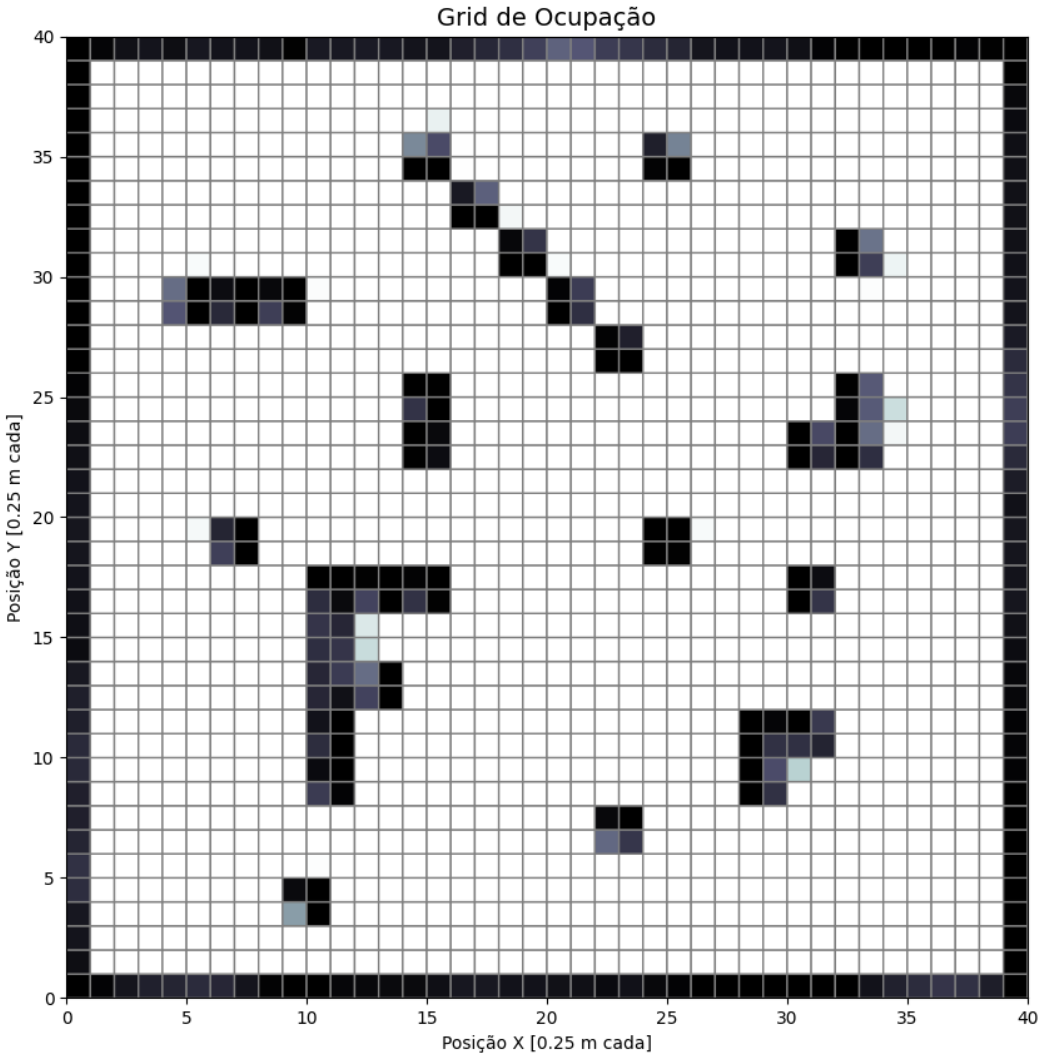
TRAJETO DO ROBOT: https://youtu.be/sMGHIkMDsE0

Link para visualização do código:

GitHub: https://github.com/Gianzanti/PEL215_TASK_04

Arena projetada:

Mapeamento Final:



Estratégia da solução

Etapa 1 - Seleção do Robot e montagem da arena

O robot selecionado foi o Pionner 3-DX, um robot diferencial, que foi equipado com um lidar de apenas um layer, projeção cilíndrica (360°) e alcance de 6 metros.

A arena foi montada em uma área de 10 x 10 metros, com muros circundando todo o perímetro, de 1 metro de altura. A disposição dos obstáculos (barris com 0.25m de diâmetro e 0.6m de altura) foi feita de forma a desafiar o desvio de obstáculos do robot e também para tornar um pouco mais difícil a tarefa de mapeamento.

A arena criada pode ser encontrada no repositório do GitHub
https://github.com/Gianzanti/PEL215_TASK_04/blob/main/Mapeamento2.wbt

Etapa 2 - Movimentação do Robot - Classe DifferentialRobot

Essa etapa foi cumprida de maneira prática fazendo "junções" dos códigos das tarefas anteriores, onde na primeira havia um robot diferencial (como o utilizado nessa tarefa) e na segunda foi estabelecido o método de movimentação com máquina de estados. A junção dessas duas classes permitiu a criação dessa classe, responsável por inicializar todos os dispositivos disponíveis no robot, definir sua geometria, seus limites e os movimentos disponíveis. É uma classe do tipo abstrata, que prevê sua utilização por meio de herança por outra classe que implemente os métodos abstratos update, move e odometry, que será responsável por atualizar os parâmetros necessários para o correto caminho.

O desafio na criação dessa classe foi a montagem do método de atualização da posição (update_position), que utiliza o supervidor do webots para capturar as informações de localização atual do robot e de atualização dos sensores (update_sensors), responsável por definir os pontos de obstáculos detectados pelo laser.

A classe DifferentialRobot pode ser vista abaixo:

In [ ]:
```
from abc import ABC, abstractmethod
import math
import numpy as np
from controller import Supervisor
from scipy.spatial.transform import Rotation as R


class DifferentialRobot(ABC):
    def __init__(self):
        self.me = Supervisor()
        self.node = self.me.getFromDef("ROBOT")
        self.timestep = int(self.me.getBasicTimeStep()) * 1
        self.max_wheel_linear_speed = 1  # m/s
        self.max_rotate_speed = 0.5
        self.v = {"vl": 0.0, "vr": 0.0}
        self.wheels = {"left": None, "right": None}
        self.steps = 0
        self.position = None
        self.rotationMatrix = None
        self.theta = 0.0
        self.initMotors()
        self.initSensors()

    def initMotors(self):
        self.wheels["left"] = self.me.getDevice("left wheel")
        self.wheels["left"].setPosition(float("+inf"))
        self.wheels["right"] = self.me.getDevice("right wheel")
        self.wheels["right"].setPosition(float("+inf"))
        self.v = {"vl": 0.0, "vr": 0.0}
        self.set_wheel_speeds()

    def set_wheel_speeds(self):
        self.wheels["left"].setVelocity(self.v["vl"])
        self.wheels["right"].setVelocity(self.v["vr"])

    def initSensors(self):
        self.lidar = self.me.getDevice("lidar")
        self.lidar.enable(self.timestep)
```

```
        self.lidar.enablePointCloud()
        self.lidarValues = []
        self.lidarDistances = []

    def update_position(self):
        self.position = self.node.getPosition().copy()
        self.rotationMatrix = self.node.getOrientation().copy()
        θz = math.atan2(self.rotationMatrix[3], self.rotationMatrix[0])
        self.theta = np.arccos(
            (
                self.rotationMatrix[0]
                + self.rotationMatrix[4]
                + self.rotationMatrix[8]
                - 1
            )
            / 2
        ) * (θz / abs(θz))
        if self.theta < 0:
            self.theta += 2 * math.pi
        print(
            f"Position: x: {self.position[0]:2f}[m], y: {self.position[1]
        )

    def update_sensors(self):
        self.lidarDistances = self.lidar.getRangeImage().copy()
        points = self.lidar.getPointCloud()
        points = np.vstack([[obj.x, obj.y, 0] for obj in points if obj.z
        r = R.from_matrix(np.array(self.rotationMatrix).reshape(3, 3))
        self.lidarValues = r.apply(points) + self.position

    def move_forward(self, speed):
        self.v["vl"] += speed
        self.v["vl"] = (
            self.v["vl"]
            if self.v["vl"] < self.max_wheel_linear_speed
            else self.max_wheel_linear_speed
        )
        self.v["vr"] += speed
        self.v["vr"] = (
            self.v["vr"]
            if self.v["vr"] < self.max_wheel_linear_speed
            else self.max_wheel_linear_speed
        )

    def move_backward(self, speed):
        self.v["vl"] -= speed
        self.v["vl"] = (
            self.v["vl"]
            if self.v["vl"] > -self.max_wheel_linear_speed
            else -self.max_wheel_linear_speed
        )
        self.v["vr"] -= speed
        self.v["vr"] = (
            self.v["vr"]
            if self.v["vr"] > -self.max_wheel_linear_speed
            else -self.max_wheel_linear_speed
        )

    def stop(self):
        self.v["vl"] = 0
```

```
        self.v["vr"] = 0

    def rotate_counterclockwise(self, speed):
        self.v["vl"] = -speed
        self.v["vl"] = (
            self.v["vl"]
            if self.v["vl"] > -self.max_rotate_speed
            else -self.max_rotate_speed
        )
        self.v["vr"] = speed
        self.v["vr"] = (
            self.v["vr"]
            if self.v["vr"] < self.max_rotate_speed
            else self.max_rotate_speed
        )

    def rotate_clockwise(self, speed):
        self.v["vl"] = speed
        self.v["vl"] = (
            self.v["vl"]
            if self.v["vl"] < self.max_rotate_speed
            else self.max_rotate_speed
        )
        self.v["vr"] = -speed
        self.v["vr"] = (
            self.v["vr"]
            if self.v["vr"] > -self.max_rotate_speed
            else -self.max_rotate_speed
        )

    @abstractmethod
    def update(self):
        pass

    @abstractmethod
    def move(self):
        pass

    @abstractmethod
    def odometry(self):
        pass

    def run(self):
        while self.me.step(self.timestep) != -1:
            self.odometry()
            self.update()
            self.move()
            self.steps += 1
```

Etapa 3 - Filtro de Kalman - Classe GridMap

Para realizar o mapeamento probabilistico proposto na tarefa, foi desenvolvida a classe GridMap, responsável por armazenar os valores de ocupação de cada célula do mapa. Para isso, existe uma correspondência para cada área de 0.25 x 0.25m (valor definido pela resolução) do mapa real para uma célula do mapa. Após a detecção do laser, esses dados são usados para calcular a ocupação de cada célula entre o robot e o obstáculo (utilizou-se o algoritmo de Bresenham para definir esse caminho). Somente a

coordenada definida pelo laser como obstáculo é preenchido como ocupado enquanto as outras são definidas como livres.

Foram estabelecidos limites máximos e mínimos para cada um dos estados (livre e ocupado) para facilitar a montagem do gráfico final.

Durante o desenvolvimento da tarefa foi utilizada a resolução de 0.50m para diminuir o tempo de execução dos testes e após estabelecido o correto funcionamento da tarefa utilizou-se uma resolução mais refinada de 0.25m. A criação de uma classe separada facilitou o processo de definição de diferentes valores.

A classe GridMap pode ser vista abaixo:

In [ ]:
```
import math
import numpy as np
from icecream import ic


class GridMap(object):
    def __init__(self, origin_x=0, origin_y=0, resolution=0.25, width=40,
        self.origin_x = origin_x
        self.origin_y = origin_y
        self.resolution = resolution
        self.width = width
        self.height = height
        self.grid = np.zeros((self.width, self.height))
        self.thresholdFree = -4000
        self.thresholdOccupied = 4000

        self.cost = {
            "free": math.log(0.35 / 0.65),
            "occupied": math.log(0.65 / 0.35),
        }

    def bresenham(self, start, end):
        # setup initial conditions
        x1, y1 = start
        x2, y2 = end
        dx = x2 - x1
        dy = y2 - y1
        is_steep = abs(dy) > abs(dx)  # determine how steep the line is
        if is_steep:  # rotate line
            x1, y1 = y1, x1
            x2, y2 = y2, x2
        # swap start and end points if necessary and store swap state
        swapped = False
        if x1 > x2:
            x1, x2 = x2, x1
            y1, y2 = y2, y1
            swapped = True
        dx = x2 - x1  # recalculate differentials
        dy = y2 - y1  # recalculate differentials
        error = int(dx / 2.0)  # calculate error
        # error = dy - dx  # calculate error
        y_step = 1 if y1 < y2 else -1
        # iterate over bounding box generating points between start and e
        y = y1
```

```
        points = []
        for x in range(x1, x2 + 1):
            coord = [y, x] if is_steep else (x, y)
            points.append(coord)
            error -= abs(dy)
            if error < 0:
                y += y_step
                error += dx

        if swapped:  # reverse the list if the coordinates were swapped
            points.reverse()

        points = np.array(points)
        return points

    def calc_xy_index_from_pos(self, pos, lower_pos, max_index):
        ind = int(np.floor((pos - lower_pos) / self.resolution))
        if 0 <= ind <= max_index:
            return ind
        else:
            return None

    def get_xy_index_from_pos(self, pos):
        indx = int(np.floor((pos[0] - self.origin_x) / self.resolution))
        indy = int(np.floor((pos[1] - self.origin_y) / self.resolution))
        if (0 <= indx < self.width) and (0 <= indy < self.height):
            return (indx, indy)
        else:
            return None

    def setCell(self, idx, val, max_value=None, min_value=None):
        try:
            current = self.grid[idx[0]][idx[1]] + val
            if min_value is not None and current < min_value:
                current = min_value

            if max_value is not None and current > max_value:
                current = max_value

            self.grid[idx[0]][idx[1]] = current

        except ValueError: #ic("***************** Invalid Cell")
            pass

    def set_occupancy_grid(self, lidar, position):
        iPos = self.get_xy_index_from_pos(position)
        if iPos is None: #ic("***************** Invalid position")
            return

        cells = []
        for beam in lidar:
            idx = self.get_xy_index_from_pos((beam[0], beam[1]))
            if idx is None:
                continue

            if cells.count(idx) > 0:
                continue

            cells.append(idx)
```

```
            line_path = self.bresenham(iPos, idx)
            for z in line_path[:-1]:
                self.setCell(z, self.cost["free"], min_value=self.thresho

            self.setCell(idx, self.cost["occupied"], max_value=self.thres
```

Etapa 4 - Sensoriamento e movimentação e captura de dados - Classe PioneerRun

Para efetivamente movimentar o robot, seguindo um planejamento pré-definido e também realizando a captura de dados para posterior plotagem, foi utilizada a classe PioneerRun, que implementa os métodos abstratos de movimentação da classe DifferentialRobot e também instancia um objeto da classe GridMap.

A lógica para a movimentação do robot foi estabelecida da seguinte forma:

1. Procurar no mapa uma célula que esteja com valor igual zero (o que significa que ela ainda não "avaliada" em nenhum momento). Caso não existem mais células zerada o processo deve continuar no passo 5;
2. Define-se essa célula como o próximo objetivo do robot;
3. Executa-se os cálculos necessários para definir o ângulo que o robot deve estar para que, em linha reta, seja possível atingir o objetivo (o desvio de obstáculos não é considerado nessa fase). Executados os cálculos o robot é rotacionado para o ângulo desejado;
4. Alcançado o ângulo é iniciado o processo de movimentação em linha reta. A cada passo executado verifica-se se o robot já atingiu o objetivo (quando reinicia-se o processo de busca de células zeradas), ou se ele deve desviar de algum obstáculo;
5. Caso todas as células zeradas já tenham sido visitadas é iniciado um percurso pré-definido, que percorre o mapa nas suas diagonais principais e também ao longo dos eixos, afim de melhorar a qualidade dos dados de mapeamento;
6. Finalizado o percurso a simulação os dados atuais do grid são salvos em disco para posterior plotagem do gráfico;

A cada passo realizado durante o processo de movimentação é feita a leitura do lidar e seu resultado é utilizado para realizar o mapeamento, usando a classe GridMap.

A lógica de funcionamento da máquina de estados: foi criada uma lista com as coordenadas de cada uma das portas existentes no mapa, enquanto houver portas a serem localizadas o robot continua andando pra frente. A cada passo dado pelo robot, a odometria é atualizada, usando a predição do filtro de Kalman (e armazenada em um numpy array), é feita uma consulta ao retorno do lidar e, caso uma porta tenha sido localizada (quando a resposta do lidar entre os ângulos 71 e 109 for infinito), é lido da lista de portas a coordenada da porta atual (essa porta então sai da lista) e é feita uma atualização da predição do filtro de Kalman (esses dados também são armazenados). Esse processo é feito continuamente até que não existam mais portas a serem localizadas e então os dados capturados são armazenados em disco.

A classe PioneerRun pode ser vista abaixo:

In [ ]:
```
from itertools import cycle, islice
import math
import random

from matplotlib import animation, pyplot as plt
```

```python
from DifferentialRobot import DifferentialRobot
import numpy as np

from icecream import ic
from GridMap import GridMap


class PioneerRun(DifferentialRobot):
    def __init__(self):
        super().__init__()
        random.seed(10)
        self.state = "find_next_target"

        self.target = {
            "ix": None,
            "iy": None,
            "px": None,
            "py": None,
            "θ": None,
            "avoiding": 0,
            "pre": False,
        }

        mapResolution = 0.25
        mapSize = (10, 10)
        mapWidth = int(mapSize[0] / mapResolution)
        mapHeight = int(mapSize[1] / mapResolution)
        self.map = GridMap(resolution=mapResolution, width=mapWidth, heig

        minimum = math.ceil(0.6 / mapResolution)

        self.predefined_targets = [
            {"x": minimum, "y": minimum},
            {"x": mapWidth - minimum, "y": mapHeight - minimum},
            {"x": minimum, "y": mapHeight - minimum},
            {"x": mapWidth - minimum, "y": minimum},
            {"x": minimum, "y": minimum},
            {"x": minimum, "y": mapHeight - minimum},
            {"x": mapWidth - minimum, "y": mapHeight - minimum},
            {"x": mapWidth - minimum, "y": minimum},
            {"x": int(mapWidth / 2), "y": minimum},
            {"x": int(mapWidth / 2), "y": int(mapHeight / 2)},
        ]
        self.lastRotateDirection = None

    def get_target_angle(self, target):
        delta_x = target["x"] - self.position[0]
        delta_y = target["y"] - self.position[1]

        theta = math.atan2(delta_y, delta_x)
        if theta < 0:
            theta += 2 * math.pi

        return theta

    def rotate(self):
        delta_theta = self.target["θ"] - self.theta

        if abs(delta_theta) < 0.01:
            self.lastRotateDirection = None
```

```
            self.stop()
            return True

        if delta_theta > math.pi:
            delta_theta -= 2 * math.pi
        elif delta_theta < -math.pi:
            delta_theta += 2 * math.pi

        if delta_theta > 0:
            if self.lastRotateDirection == "cw":
                self.lastRotateDirection = None
                self.stop()
                return True

            self.rotate_counterclockwise(self.max_wheel_linear_speed)
            self.lastRotateDirection = "ccw"
        else:
            if self.lastRotateDirection == "ccw":
                self.lastRotateDirection = None
                self.stop()
                return True

            self.rotate_clockwise(self.max_wheel_linear_speed)
            self.lastRotateDirection = "cw"

        return False

    def move(self):
        self.set_wheel_speeds()

    def odometry(self):
        if self.state != "stop":
            self.update_position()

    def cyclic_range(self, start, stop):
        return list(islice(cycle(range(stop)), start, start + stop))

    def find_target(self):
        cyclePos = self.map.get_xy_index_from_pos(self.position)
        cycleX = self.cyclic_range(cyclePos[0], self.map.width - 1)
        cycleY = self.cyclic_range(cyclePos[1], self.map.height - 1)

        # FIND ZERO VALUE TARGETS
        for x in cycleX:
            for y in cycleY:
                if (
                    self.map.grid[x][y] == 0
                    and (x != self.target["ix"] and y != self.target["iy"]
                    # and self.lastTargets[-1] != (x, y)
                ):
                    posX = x * self.map.resolution + self.map.origin_x
                    posY = y * self.map.resolution + self.map.origin_y
                    delta_x = posX - self.position[0]
                    delta_y = posY - self.position[1]
                    angle = math.atan2(delta_y, delta_x)
                    if angle < 0:
                        angle += 2 * math.pi

                    self.target = {
                        "ix": x,
```

```
                    "iy": y,
                    "px": posX,
                    "py": posY,
                    "θ": angle,
                    "avoiding": 0,
                    "value": self.map.grid[x][y],
                    "pre": False,
                }
                return True

        # PREDEFINED TARGETS
        if len(self.predefined_targets) > 0:
            predefined = self.predefined_targets[0]
            posX = predefined["x"] * self.map.resolution + self.map.origi
            posY = predefined["y"] * self.map.resolution + self.map.origi
            delta_x = posX - self.position[0]
            delta_y = posY - self.position[1]
            angle = math.atan2(delta_y, delta_x)
            if angle < 0:
                angle += 2 * math.pi

            self.target = {
                "ix": predefined["x"],
                "iy": predefined["y"],
                "px": posX,
                "py": posY,
                "θ": angle,
                "avoiding": 0,
                "value": self.map.grid[predefined["x"]][predefined["y"]],
                "pre": True,
            }
            return True
        else:
            return False

    def follow_target(self):
        delta_x = self.target["px"] - self.position[0]
        delta_y = self.target["py"] - self.position[1]

        # checks if the robot is close enough to the target
        if abs(delta_x) < 0.05 and abs(delta_y) < 0.05:
            ic("Close enough to target")
            if self.target["pre"]:
                if len(self.predefined_targets) > 0:
                    self.predefined_targets.pop(0)
            self.stop()
            self.state = "find_next_target"
            return True

        # avoid obstacles
        left_side = self.lidarDistances[150:180]
        right_side = self.lidarDistances[180:210]
        obstacle_left = min(left_side) <= 0.4
        obstacle_right = min(right_side) <= 0.4
        left_value = sum(
            [len(left_side) if x == float("inf") else x for x in left_sid
        ) / len(left_side)
        right_value = sum(
            [len(right_side) if x == float("inf") else x for x in right_s
        ) / len(right_side)
```

```
        if obstacle_left or obstacle_right:
            if left_value < right_value:
                ic("avoiding obstacles at left")
                angle = self.theta - math.pi / 4
            else:
                ic("avoiding obstacles at right")
                angle = self.theta + math.pi / 4

            # Calculate new coordinates
            iPos = None
            while iPos == None:
                dist = (
                    2 if self.target["avoiding"] == 0 else self.target["a
                )
                new_x = self.position[0] + ((random.random() * dist) * ma
                new_y = self.position[1] + ((random.random() * dist) * ma
                iPos = self.map.get_xy_index_from_pos([new_x, new_y])

            delta_x = new_x - self.position[0]
            delta_y = new_y - self.position[1]
            angle = math.atan2(delta_y, delta_x)
            if angle < 0:
                angle += 2 * math.pi

            self.target = {
                "ix": iPos[0],
                "iy": iPos[1],
                "px": new_x,
                "py": new_y,
                "0": angle,
                "avoiding": dist if dist <= 5 else 5,
                "value": self.map.grid[iPos[0]][iPos[1]],
                "pre": False,
            }

            # ic("avoiding obstacles")
            self.state = "rotate_to_target"
            return True

    def update(self):
        ic(self.state, self.target)
        self.update_sensors()
        self.map.set_occupancy_grid(
            self.lidarValues, [self.position[0], self.position[1]]
        )

        match self.state:
            case "find_next_target":
                foundTarget = self.find_target()
                if not foundTarget:
                    ic(self.steps)
                    self.state = "stop"
                    ic(self.map.grid)

                    np.savez_compressed(f"./data_grid.npz", grid=self.map

                    new_grid = np.rot90(self.map.grid, k=3)
                    new_grid = np.fliplr(new_grid)
                    plt.pcolor(
```

```
                        new_grid,
                        cmap="Blues",
                        edgecolor="tab:gray",
                        linewidths=1,
                    )
                    plt.title("Occupancy Grid")
                    plt.tight_layout()
                    plt.axis("equal")
                    plt.show()

                    return

                self.state = "rotate_to_target"

            case "move_to_target":
                if self.follow_target():
                    return

                self.move_forward(self.max_wheel_linear_speed)

            case "rotate_to_target":
                if self.rotate():
                    self.state = "move_to_target"

            case "stop":
                self.stop()
```

Etapa 5 - Simulação - Pionner 3DX Controller

Para a execução da simulação no Webots é criado um script python que integra as classes descritas acima e então esse script é definido como o controlador do robot no simulador Webots (código listado abaixo) e então inicia-se a simulação. O resultado pode ser visto no vídeo https://youtu.be/sMGHIkMDsE0

In [ ]:
```
from PioneerRun import PioneerRun

if __name__ == "__main__":
    robot = PioneerRun()
    robot.run()
```

Etapa 6 - Plotagem do mapeamento

Para exibir a progressão do resultado do mapeamento, a idéia original era criar uma animação que exibe-se o progresso do mapeamento. Entretanto, devido à dificuldade de armazenamento de dados em tempo hábil para que a geração do vídeo da trajetória não ficasse prejudicado, optou-se por fazer uma simples exibição do resultado final do mapeamento, em uma única imagem, a partir dos dados salvos no processo anterior.

O código para a criação dos gráficos pode ser visto abaixo:

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt

data = np.load("data_grid.npz")
```

```
fig, ax = plt.subplots(1, 1, figsize=(20, 6))

ax.set_ylabel("Posição Y [0.25 m cada]")
ax.set_xlabel("Posição X [0.25 m cada]")
ax.set_title("Grid de Ocupação", fontsize=14)
ax.set_aspect("equal", "box")

new_grid = np.rot90(data["grid"], k=3)
new_grid = np.fliplr(new_grid)
ax.pcolor(
    new_grid,
    cmap="bone_r",
    edgecolor="tab:gray",
    linewidths=1,
)
plt.show()
```