PEL 215 - Tarefa 02

Discente: Fábio Rossatti Gianzanti

Docente: Prof. Dr. Flávio Tonidandel

O desafio está dividido em 2 partes:

1. Na primeira etapa, usando o robot Youbot, da Kuka, com rodas mecanum, é necessário percorrer um trajeto hexagonal, voltando ao local do início do trajeto, dentro do simulador Webots. Não foi fornecida nenhuma arena específica.
2. A próxima etapa consiste em distribuir uma série de obstaculos em uma arena (não fornecida) e fazer com que o robot alcance um destino (não pré-definido), usando a técnica de navegação por campos potenciais, também dentro do simulador Webots

Links para a visualização dos vídeos demonstrativos:

PARTE 1: TRAJETO HEXAGONAL: https://youtu.be/TZ6NdUCAA0Q

PARTE 2: CAMPOS POTENCIAIS: https://youtu.be/iaPjz7WkDWw

Link para visualização do código:

GitHub: https://github.com/Gianzanti/PEL215_TASK_02

Estratégia da solução

Etapa 1 - Movimentação Hexagonal - Classe MecanumRobot

Para cumprir a primeira etapa foi necessário inicialmente definir a movimentação do robo de forma adequada. Estudou-se como definir as movimentações necessárias, sem movimentar a frente do robot para a direção do movimento e foi criado uma classe com os métodos necessários.

Essa classe (MecanumRobot) é 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.

A única informação externa fornecida a essa classe é a posição inicial do robot (arg: initPos)

A classe MecanumRobot pode ser vista abaixo:

In [ ]:

```
from abc import ABC, abstractmethod
from controller import Robot

INF = float("+inf")


class MecanumRobot(ABC):
    def __init__(self, initPos: tuple[float, float] = (0.0, 0.0)) -> None
        self.me = Robot()
        self.timestep = int(self.me.getBasicTimeStep()) * 1
        maxVelocity = 14.81 / 2  # rad/s
        self.wheel_radius = 0.05  # m
        self.max_speed = maxVelocity * self.wheel_radius  # m/s
        self.speed_increment = 0.5 * self.max_speed
        # self.l = {"x": 0.228, "y": 0.158}
        self.v = {"x": 0.0, "y": 0.0}
        self.p = {"x": initPos[0], "y": initPos[1]}
        self.wheels = []
        self.steps = 0
        self.initMotors()

    def initMotors(self):
        for i in range(0, 4):
            self.wheels.append(self.me.getDevice(f"wheel{i+1}"))

        self.set_wheel_speeds([0, 0, 0, 0])

    def set_wheel_speeds(self, speeds):
        for i in range(0, 4):
            self.wheels[i].setPosition(INF)
            self.wheels[i].setVelocity(speeds[i])

    def base_move(self):
        speeds = [
            1 / self.wheel_radius * (self.v["x"] + self.v["y"]),
            1 / self.wheel_radius * (self.v["x"] - self.v["y"]),
            1 / self.wheel_radius * (self.v["x"] - self.v["y"]),
            1 / self.wheel_radius * (self.v["x"] + self.v["y"]),
        ]
        self.set_wheel_speeds(speeds)
        print(f"Speeds: vx: {self.v['x']:2f}[m/s], vy: {self.v['y']:2f}[m

    def update_position(self):
        self.p["x"] += self.v["x"] * self.timestep / 1000
        self.p["y"] += self.v["y"] * self.timestep / 1000
        print(f'Position: x: {self.p["x"]:2f}[m], y: {self.p["y"]:2f}[m]'

    def move_forward(self, speed):
        self.v["x"] += speed
        self.v["x"] = self.v["x"] if self.v["x"] < self.max_speed else se

    def move_forward_left(self, speed):
        self.v["x"] += speed
        self.v["x"] = self.v["x"] if self.v["x"] < self.max_speed else se
        self.v["y"] += speed
        self.v["y"] = self.v["y"] if self.v["y"] < self.max_speed else se

    def move_forward_right(self, speed):
        self.v["x"] += speed
```

```
        self.v["x"] = self.v["x"] if self.v["x"] < self.max_speed else se
        self.v["y"] -= speed
        self.v["y"] = self.v["y"] if self.v["y"] > -self.max_speed else -

    def move_backward(self, speed):
        self.v["x"] -= speed
        self.v["x"] = self.v["x"] if self.v["x"] > -self.max_speed else -

    def move_backward_left(self, speed):
        self.v["x"] -= speed
        self.v["x"] = self.v["x"] if self.v["x"] > -self.max_speed else -
        self.v["y"] += speed
        self.v["y"] = self.v["y"] if self.v["y"] < self.max_speed else se

    def move_backward_right(self, speed):
        self.v["x"] -= speed
        self.v["x"] = self.v["x"] if self.v["x"] > -self.max_speed else -
        self.v["y"] -= speed
        self.v["y"] = self.v["y"] if self.v["y"] > -self.max_speed else -

    def stop(self):
        self.v["x"] = 0
        self.v["y"] = 0

    def move_left(self, speed):
        self.v["y"] += speed
        self.v["y"] = self.v["y"] if self.v["y"] < self.max_speed else se

    def move_right(self, speed):
        self.v["y"] -= speed
        self.v["y"] = self.v["y"] if self.v["y"] > -self.max_speed else -

    @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.update()
            self.move()
            self.odometry()
            self.steps += 1
```

Etapa 1 - Movimentação Hexagonal - Classe KukaPath

Essa classe é a responsável por executar o caminho pré-definido para a movimentação hexagonal do robot e herda da classe anterior (MecanumRobot) as capacidades de movimentação.

Essa classe implementa os seguintes métodos: - odometry: após a movimentação do robot, aplica as velocidades estabelecidas (durante o timestep) nos eixos X e Y à posição anterior do robot, para definir sua posição atual. - move: aplica as velocidades calculadas pelo método update às rodas do robot, conforme o movimento desejado, executando efetivamente sua movimentação; - update: consiste de uma máquina de estados responsável por controlar o próximo passo no caminho pré-definido;

A essa classe devem ser fornecidas a posição inicial do robot e a lista com os diversos passos que o robot deve executar.

In [ ]:
```
from MecanumRobot import MecanumRobot


class KukaPath(MecanumRobot):
    def __init__(self, pos: tuple[float, float], path: list[tuple[float,
        super().__init__(pos)

        self.state = "checking"
        self.path = path
        self.next = 0
        self.target = {"x": pos[0], "y": pos[1]}

    def move(self):
        self.base_move()

    def odometry(self):
        self.update_position()

    def update(self):
        print(f"steps: {self.steps}")
        print(f"state: {self.state}")
        match self.state:
            case "checking":
                checkPosX = abs(self.p["x"] - self.target["x"])
                checkPosY = abs(self.p["y"] - self.target["y"])
                print(f'Position: X:{self.p["x"]}, Y:{self.p["y"]}')
                print(f'Target: X:{self.target["x"]}, Y:{self.target["y"]
                print(
                    f"ΔposX: {checkPosX} [{checkPosX < 0.01}] - ΔposY: {c
                )

                if checkPosX < 0.01 and checkPosY < 0.01:
                    self.stop()
                    if (self.next + 1) < len(self.path):
                        self.next += 1
                        self.state = "decideMovement"
                        print(f"Going to next step: {self.path[self.next]
                    else:
                        self.state = "stopped"

            case "decideMovement":
                Δx = self.path[self.next][0] - round(self.p["x"], 0)
                Δy = self.path[self.next][1] - round(self.p["y"], 0)
                print(f"Δx: {Δx}, Δy: {Δy}")

                if Δx > 0:
                    if Δy > 0:
```

```
                    self.move_forward_left(self.speed_increment)
                elif Δy == 0:
                    self.move_forward(self.speed_increment)
                elif Δy < 0:
                    self.move_forward_right(self.speed_increment)
                else:
                    self.stop()

            elif Δx == 0:
                if Δy > 0:
                    self.move_left(self.speed_increment)
                elif Δy == 0:
                    self.stop()
                elif Δy < 0:
                    self.move_right(self.speed_increment)
                else:
                    self.stop()

            elif Δx < 0:
                if Δy > 0:
                    self.move_backward_left(self.speed_increment)
                elif Δy == 0:
                    self.move_backward(self.speed_increment)
                elif Δy < 0:
                    self.move_backward_right(self.speed_increment)
                else:
                    self.stop()

            else:
                self.stop()

            self.target = {"x": self.p["x"] + Δx, "y": self.p["y"] +
            self.state = "checking"

            print(
                f'Current Position: {round(self.p["x"], 0)}, {round(s
            )
            print(f"Next state: {self.state}")
```

Etapa 1 - Movimentação Hexagonal - Youbot Controller

Para o controle do robot foi criado um script em Python que instancia a classe KukaPath e define os parâmetros necessários à movimentação hexagonal.

Para a simulação foi criada uma arena com 4 tiles de 1 m x 1 m (totalizando 4 m x 4 m) e foi posicionada de forma que o ponto central do tile do canto inferior esquerdo corresponda às coordenadas X: 0.0 m e Y :0.0 m.

A posição inicial do robot é definida como X: 1,0 m e Y: 0,0 m

O caminho definido para a movimentação é: Passo 1 = X: 1,0 m , Y: 0,0 m Passo 2 = X: 2,0 m , Y: 0,0 m Passo 3 = X: 3,0 m , Y: 1,0 m Passo 4 = X: 3,0 m , Y: 2,0 m Passo 5 = X: 2,0 m , Y: 3,0 m Passo 6 = X: 1,0 m , Y: 3,0 m Passo 7 = X: 0,0 m , Y: 2,0 m Passo 8 = X: 0,0 m , Y: 1,0 m Passo 9 = X: 1,0 m , Y: 0,0 m

Esse script é então 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/TZ6NdUCAA0Q

In [ ]:
```
from KukaPath import KukaPath


if __name__ == "__main__":
    initial_position = (1, 0)
    path = [
        (1, 0),
        (2, 0),
        (3, 1),
        (3, 2),
        (2, 3),
        (1, 3),
        (0, 2),
        (0, 1),
        (1, 0),
    ]
    kuka = KukaPath(initial_position, path)
    kuka.run()
```

Etapa 2 - Navegação por Campos Potenciais - Classe PotencialFields

Estabelecida a correta movimentação do robot, agora se faz necessário implementar um método de navegação em que o caminho seja calculado e não pré-definido. Para isso, usou-se a técnica de navegação por campos potenciais.

À essa classe fornece-se a dimensão do campo, onde foi usada as dimensões em metros da area, as coordenadas do objetivo e as coordenadas dos obstáculos. Solicita-se então o cálculo do Campo Potencial de cada célula, fornecendo como parâmetros o Katt (fator de escala positivo do potencial atrativo) e o Krep (fator de escala positivo do potencial repulsivo).

Essa classe também fornece uma visualização gráfica dos campos potenciais.

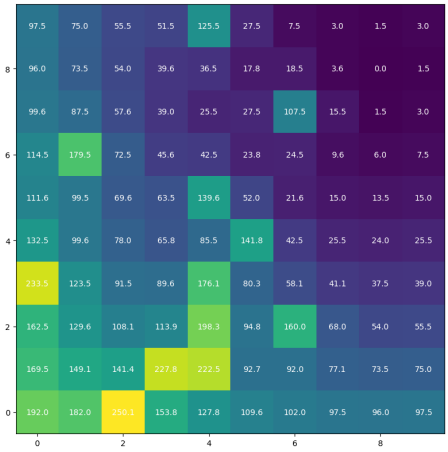
A classe PotencialFields pode ser vista abaixo:

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

class PotencialFields:
    def __init__(
        self,
        size: tuple[int, int],
        goal: tuple[int, int],
        obstacles: list[tuple[int, int, int]],
    ) -> None:
        """
        size [Tamanho da arena]: (X units, Y units)

        goal [Coordenada do objetivo]: (X, Y)
```

```
        obstacles [Lista das coordenas dos obstáculos]: [(X, Y, radius),
        """
        self.sizeX, self.sizeY = size
        self.coords = np.zeros(size)
        self.goal = goal
        self.obstacles = obstacles

    def attractivePotencial(self, Katt=1):
        ua = np.zeros_like(self.coords)

        for x in range(self.sizeX):
            for y in range(self.sizeY):
                dist = np.sqrt((self.goal[0] - x) ** 2 + (self.goal[1] -
                ua[x][y] = 0.5 * Katt * dist**2

        return ua

    def repulsionPotencial(self, Krep=50):
        up = np.zeros_like(self.coords)

        for x in range(self.sizeX):
            for y in range(self.sizeY):
                for obstacle in self.obstacles:
                    dist = np.sqrt((obstacle[0] - x) ** 2 + (obstacle[1]
                    if dist == 0:
                        up[x][y] += Krep
                    elif dist <= obstacle[2]:
                        up[x][y] += 0.5 * Krep * (1 / dist - 1 / obstacle

        return up

    def calculatePotencialField(self, Katt=1, Krep=50):
        return self.attractivePotencial(Katt) + self.repulsionPotencial(K

    def showPlot(self, u):
        fig, ax = plt.subplots(figsize=(20, 20))
        plt.imshow(u.T)
        ax.invert_yaxis()

        for x in range(self.sizeX):
            for y in range(self.sizeY):
                text = ax.text(
                    x,
                    y,
                    "{:.1f}".format(u[x, y]),
                    ha="center",
                    va="center",
                    color="w",
                )

        plt.show()
```

Etapa 2 - Definição do caminho - Classe PotencialFields

Definidos os campos potenciais, como visto nessa imagem:



, passa-se à definição do caminho, por meio da busca do gradiente descendente dos campos potenciais.

Foi implementado um algoritmo simples de busca pelo menor valor das 8 células adjacentes à célula em análise, a partir da célula de início de posição do robot. A busca continua até que não haja células de menor valor, indicando o objetivo final ou então um local de mínimo, que causa o mal funcionamento do algoritmo. Essa situação foi encontrada em algumas configurações de distribuição dos obstáculos e como não foi implementado nenhum método de "fuga do local mínimo" foram necessários ajustes manuais no posicionamento dos obstáculos para evitar esse problema.

A função gradient_descent_algorithm pode ser vista abaixo:

In [ ]:
```
def gradient_descent_algorithm(cellsPF, start_node):
    shortest_path = []
    current_node = ""
    next_node = start_node

    motion = [[1, 0], [0, 1], [-1, 0], [0, -1], [-1, -1], [-1, 1], [1, -1

    while next_node != current_node:
        current_node = next_node
        shortest_path.append(current_node)
        cost = cellsPF[current_node[0]][current_node[1]]

        for move in motion:
            neighbor = (current_node[0] + move[0], current_node[1] + move

            valX = cellsPF[neighbor[0] : neighbor[0] + 1]
            if valX.size == 0:
                continue

            valY = valX[0][neighbor[1] : neighbor[1] + 1]
            if valY.size == 0:
                continue

            tentative_value = valY[0]

            if tentative_value < cost:
```

```
                cost = tentative_value
                next_node = neighbor

    return shortest_path
```

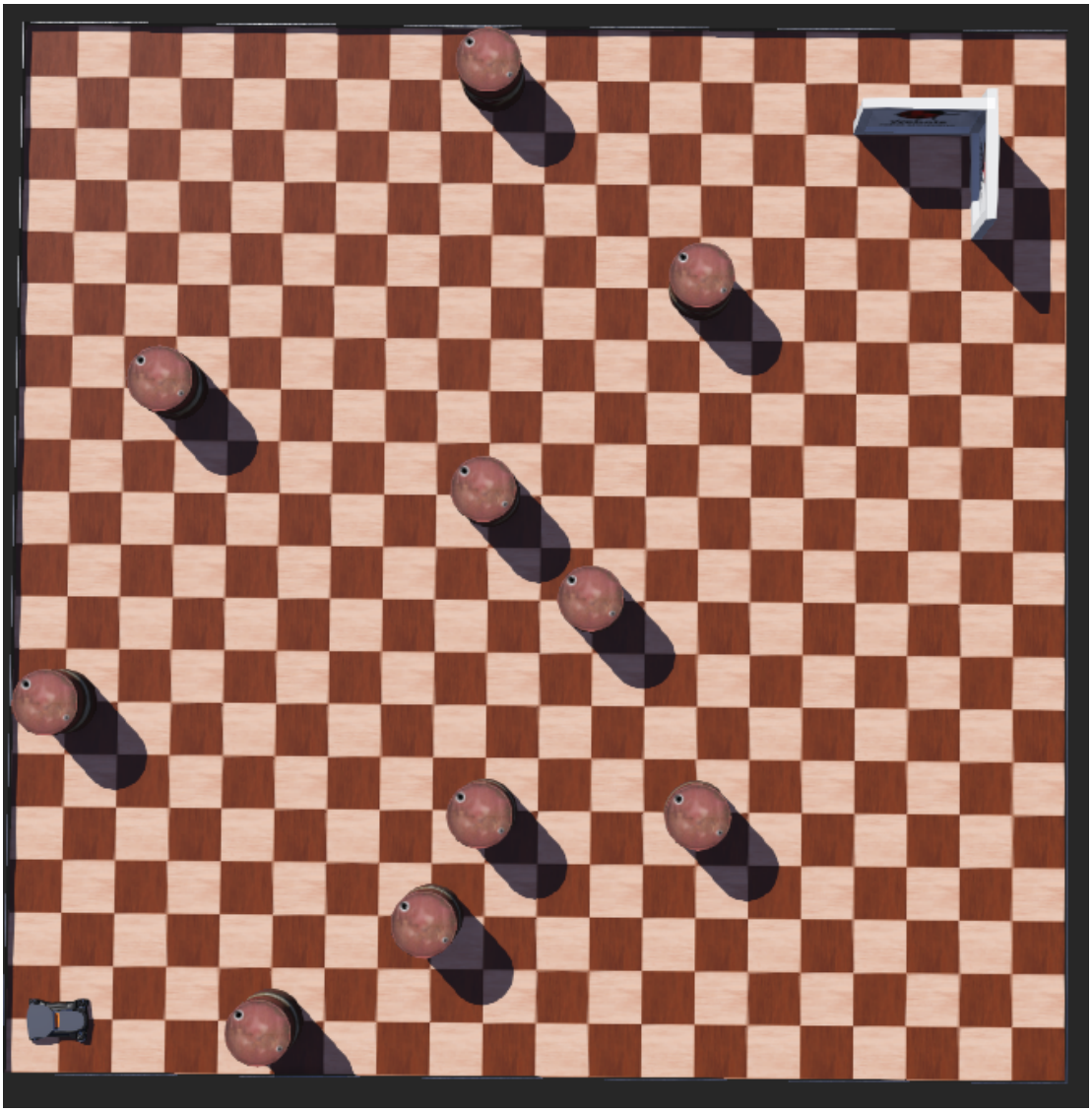
Etapa 2 - Youbot Controller

Por fim, para fazer a integração do novo processo de navegação ao simulador Webots o script controlador foi modificado para que os novos recursos fossem implementados.

Para a simulação foi criada uma arena com 100 tiles de 1 m x 1 m (totalizando 10 m x 10 m) e foi posicionada de forma que o ponto central do tile do canto inferior esquerdo corresponda às coordenadas X: 0.0 m e Y :0.0 m.

A posição inicial do robot é definida, manualmente, como X: 0,0 m e Y: 0,0 m.

Foram posicionados alguns obstáculos (barris) no espaço da arena e também um objetivo foi demarcado com 2 paredes, como é possível ver abaixo:



Utilizou-se como Katt o valor 3 e como Krep o valor 100. Para definir a distância de influência do obstáculo utilizou-se como parâmetro a soma dos raios dos obstáculos e do robot.

O caminho resultante do processo do cálculo de potenciais e descida de gradiente é então fornecido à mesma classe de movimentação do robot vista anteriormente (KukaPath).

Esse script é então 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/iaPjz7WkDWw

In [ ]:
```
import math
from Gradient import gradient_descent_algorithm
from KukaPath import KukaPath
from PotencialFields import PotencialFields

if __name__ == "__main__":
    start_position = (0, 0)
    kuka = KukaPath(start_position)

    arenaDimensions = (10, 10)
    goal = (8, 8)
    obstacle_radius = 0.305  # m
    robot_radius = kuka.robot_radius  # 0.456 m
    repulsion_radius = math.ceil(obstacle_radius + robot_radius) * 2  # m
    obstacles = [
        (0, 3, repulsion_radius),
        (1, 6, repulsion_radius),
        (2, 0, repulsion_radius),
        (3.5, 1, repulsion_radius),
        (4, 2, repulsion_radius),
        (4, 3, repulsion_radius),
        (4, 5, repulsion_radius),
        (4, 9, repulsion_radius),
        (5, 4, repulsion_radius),
        (6, 2, repulsion_radius),
        (6, 7, repulsion_radius),
    ]
    pf = PotencialFields(arenaDimensions, goal, obstacles)

    Katt = 3
    Krep = 100
    cellsPF = pf.calculatePotencialField(Katt, Krep)
    shortest_path = gradient_descent_algorithm(cellsPF, start_position)

    kuka.setPath(shortest_path)
    kuka.run()
```