PEL 215 - Tarefa 03 - Filtro de Kalman 1D

Discente: Fábio Rossatti Gianzanti

Docente: Prof. Dr. Flávio Tonidandel

O desafio consiste em estimar a posição de um robot, durante seu percurso, somente sobre o eixo X de um mapa previamente fornecido, usando o algoritmo do Filtro de Kalman.

Como o mapa é fornecido, é possível saber a posição inicial do robot, além da posição das portas, permitindo que somente com o filtro de Kalman seja possível estimar a odometria do robot.

Links para a visualização dos vídeos demonstrativos:

TRAJETO DO ROBOT: https://youtu.be/iZlaCtKE9xs

GRAFICO DAS GAUSSIANAS: https://youtu.be/uCrD128W5yM

TRAJETO COM GAUSSIANAS: https://youtu.be/44kWNdQBiZA

Link para visualização do código:

GitHub: https://github.com/Gianzanti/PEL215_TASK_03

Estratégia da solução

Etapa 1 - Uso da arena fornecida

O primeiro desafio foi fazer o uso correto da arena previamente fornecida, haja visto que ela foi criada em versões anteriores do Webots (no desafio corrente foi usada a versão Webots R2023b).

Para fazer a correta migração foram utilizados os tutorias da documentação do Webots, que mostram os processos necessários para migrar as versões dos arquivos:

Upgrade from R2022a to R2022b: https://cyberbotics.com/doc/guide/from-2022a-to-2022b

Upgrade from R2023a to R2023b: https://cyberbotics.com/doc/guide/from-2023a-to-2023b

As consequências da não adaptação de versão não tem efeito prático na utilização da arena, apenas estético.

A arena já migrada pode ser encontrada no repositório do GitHub
https://github.com/Gianzanti/PEL215_TASK_03/blob/main/kalman.wbt

Etapa 2 - Movimentação do Robot - Classe TurtleBurguerBot

Essa etapa foi cumprida de maneira prática fazendo "junções" dos códigos das duas 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.

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

A classe MecanumRobot pode ser vista abaixo:

In [ ]:
```python
from abc import ABC, abstractmethod
from controller import Robot

INF = float("+inf")

class TurtleBurguerBot(ABC):
    def __init__(self, initPos: tuple[float, float] = (0.0, 0.0)) -> None
        """
        initPos: tuple[float, float] - defines initial position (x,y) for
        """
        self.me = Robot()
        self.timestep = int(self.me.getBasicTimeStep()) * 1
        maxVelocity = 6.67  # rad/s
        self.wheel_radius = 0.033  # m
        self.max_speed = maxVelocity * self.wheel_radius  # m/s
        self.speed_increment = 0.5 * self.max_speed
        self.v = {"x": 0.0, "y": 0.0}
        self.p = {"x": initPos[0], "y": initPos[1]}
        self.wheels = []
        self.steps = 0
        self.initMotors()
        self.initSensors()

    def initMotors(self):
        self.wheels.append(self.me.getDevice("left wheel motor"))
        self.wheels.append(self.me.getDevice("right wheel motor"))
        self.set_wheel_speeds([0, 0])

    def set_wheel_speeds(self, speeds):
        for i in range(0, 2):
            self.wheels[i].setPosition(INF)
            self.wheels[i].setVelocity(speeds[i])
```

```
    def initSensors(self):
        self.lidar = self.me.getDevice("LDS-01")
        self.lidar.enable(self.timestep)
        self.lidar.enablePointCloud()

    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"]),
        ]
        self.set_wheel_speeds(speeds)
        # print(f"Speeds: vx: {self.v['x']:2f}[m/s], vy: {self.v['y']:2f}

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

    @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 3 - Filtro de Kalman - Classe KalmanFilter

Para aplicar o algoritmo do filtro de Kalman foi criada uma classe com a versão simplificada do filtro, para movimentação 1D, sem a utilização de matrizes. Entretanto, como a aplicação foi realizada usando uma classe externa, caso seja necessário sua modificação para movimentações mais complexas, basta atualizar a classe.

A equação utilizada:

PREDIÇÃO

$$\bar{x}_t = x_{t-1} + \Delta t\, v$$

$$\bar{\Sigma}_t = \Sigma_{t-1} + R_t$$

CORREÇÃO

$$K_t = \bar{\Sigma}_t (\bar{\Sigma}_t + Q_t)^{-1}$$

$$x_t = \bar{x}_t + K_t(z_t - \bar{x}_t)$$

$$\Sigma_t = \bar{\Sigma}_t - K_t \bar{\Sigma}_t$$

A classe KalmanFilter pode ser vista abaixo:

In [ ]:
```
class KalmanFilter(object):
  def __init__(self, dt, sigma_a, sigma_z, x):
    self.dt = dt            # time step
    self.sigma_a = sigma_a  # motion model noise
    self.sigma_z = sigma_z  # measurement noise

    self.A = 1
    self.B = self.dt
    self.R = self.dt ** 2 * self.sigma_a ** 2
    self.Q = sigma_z ** 2
    self.E = 1
    self.C = 1
    self.x = x

  def predict(self, u):
    self.x = (self.A * self.x) + (self.B * u)
    self.E = self.E + self.R
    return [self.x, self.E]

  def update(self, z):
    K = self.E / (self.E + self.Q)
    self.x = self.x + K * (z-self.x)
    self.E = self.E - K * self.E
    return [self.x, self.E]
```

Etapa 4 - Sensoriamento e Captura de dados - Classe TurtlePath

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

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.

Nessa classe são definidos os parâmetros do filtro de Kalman, a respeito dos erros de odometria e sensoriamento. Como não foram realizados testes empíricos para determinar esses níveis de ruído, foram escolhidos os melhores valores de forma que os gráficos das gaussianas fossem o mais didático possível para o bom entendimento da aplicação do filtro de Kalman.

A classe TurtlePath pode ser vista abaixo:

In [ ]:
```
from Kalman import KalmanFilter
from TurtleBurguerBot import TurtleBurguerBot
import numpy as np
from icecream import ic

class TurtlePath(TurtleBurguerBot):
    def __init__(self, pos: tuple[float, float]):
        super().__init__(pos)
        self.state = "find-next-door"

        self.sigma_a = 0.2  # Desvio padrão de odometria [m]
        self.sigma_z = 0.1   # Desvio padrão do sensor [m]

        self.kf = KalmanFilter(self.timestep/1000, self.sigma_a, self.sig

        FIRST_DOOR = (-2.73, 0)
        SECOND_DOOR = (-0.735, 0)
        THIRD_DOOR = (2.73, 0)
        self.doors = [FIRST_DOOR[0], SECOND_DOOR[0], THIRD_DOOR[0]]

        steps = 1000
        self.data_predictions = np.zeros((steps,), dtype='f,f')
        self.data_measurements = np.zeros((steps,), dtype='f,f')
        self.data_corrections = np.zeros((steps,), dtype='f,f')


    def move(self):
        self.base_move()

    def odometry(self):
        if (self.state != "stop"):
            print(f'previous Position: x: {self.kf.x:2f}[m]')
            [xbarra, Ebarra] = self.kf.predict(self.v["x"])
            print(f'Position Predicted: x: {xbarra:2f}[m]')
            self.data_predictions[self.steps] = (xbarra, Ebarra)

    def correctPrediction(self):
        ic(self.doors)
        z = self.doors.pop(0)
        ic(z)
        self.data_measurements[self.steps] = (z, self.sigma_z)
        [x, E] = self.kf.update(z)
        self.data_corrections[self.steps] = (x, E)
        print(f'Position Updated: x: {x:2f}[m]')

    def update(self):
        ic(self.steps, self.state)
        match self.state:
            case "checking":
                lidar_values = self.lidar.getRangeImage()

                allInf = True
                for i in range(71, 110):
                    if lidar_values[i] != float('inf'):
                        allInf = False
                        continue
```

```
                if allInf:
                    if (len(self.doors) > 0):
                        self.correctPrediction()
                        self.state = "find-next-door"
                    else:
                        self.state = "graph"


            case "find-next-door":
                self.move_forward(self.max_speed)
                self.state = "checking"

            case "graph":
                self.stop()
                self.state = "stop"
                np.savez_compressed(
                    f"./data_kalman.npz",
                    timeStep=self.timestep,
                    predictions=self.data_predictions,
                    measurements=self.data_measurements,
                    corrections=self.data_corrections,
                )

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

Etapa 5 - Simulação - TurtleBurgueBot 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/iZlaCtKE9xs

In [ ]:
```
from TurtlePath import TurtlePath

if __name__ == "__main__":
    start_position = (-4,0)
    turtle = TurtlePath(start_position)
    turtle.run()
```

Etapa 6 - Plotagem das Gaussianas

Para exibir a progressão do resultado da aplicação do filtro de Kalman, decidiu-se por fazer uma animação, usando o matplotlib, em que a cada timestep da simulação do webots fosse criado um frame com o plot da gaussiana da odometria obtida pelo filtro de Kalman.

Caso estivesse em um ponto de observação (ou seja, onde havia uma porta) é também feita a plotagem da gaussiana da observação e também da correção da predição.

Os dados para a obtenção dos gráficos são obtidos do arquivo em que foram salvos os dados da simulação e ao final da plotagem da animação é criado um vídeo, que pode ser observado em: https://youtu.be/uCrD128W5yM

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

In [ ]:

```
import numpy as np
from matplotlib import pyplot as mp
from scipy.stats import norm
from matplotlib import animation

def showGaussian():
    load = np.load("./data_kalman.npz")
    data = {'timeStep': load["timeStep"], 'predictions': load["prediction

    data_points = 960

    x = np.linspace(-5, 5, data_points)

    fig, ax = mp.subplots(1,1,figsize=(20,6))
    ln1, = ax.plot([], [], 'b-', animated=True)

    ax.text(-4.9, 5.5, 'Odometria', color='blue', fontsize=14)
    ax.text(-4.9, 4.5, 'Observação', color='green', fontsize=14)
    ax.text(-4.9, 3.5, 'Correção', color='red', fontsize=14)

    ax.set_ylabel('Densidade')
    ax.set_xlabel('Posição X [m]')
    ax.set_title('Filtro de Kalman', fontsize=14)

    ax.set_xlim(-5, 5)
    ax.set_ylim(0, 6)
    ax.grid()

    def init():
        ln1.set_data([], [])
        return ln1

    def update(frame):
        if (data['predictions'][frame][0] == 0 or data['predictions'][fra
            ln1.set_data([], [])
        else:
            y = norm.pdf(x, data['predictions'][frame][0], np.sqrt(data['
            ln1.set_data(x, y)

        # plotting measurements
        if (data['measurements'][frame][0] != 0 and data['measurements'][
            y = norm.pdf(x, data['measurements'][frame][0], np.sqrt(data[
            ax.plot(x, y, 'g-')
            ax.axvline(x=data['measurements'][frame][0], color='g', lines
            ax.text(data["measurements"][frame][0] - 0.1, -0.3, f'{data["

        # plotting corrections
        if (data['corrections'][frame][0] != 0 and data['corrections'][fr
            y = norm.pdf(x, data['corrections'][frame][0], np.sqrt(data['
            ax.plot(x, y, 'r-')
            ax.axvline(x=data['corrections'][frame][0], color='r', linest

        return ln1

    ani = animation.FuncAnimation(fig, update, frames=data_points + 39, i

    writer = animation.FFMpegWriter(
```

```
        fps=15, metadata=dict(artist='Me'), bitrate=1800)
    ani.save("movie.mp4", writer=writer)


if __name__ == "__main__":
    showGaussian()
```