AI 3603 Artificial Intelligence: Principles and Techniques

Homework 3: Particle Filter Due Dec. 15th 18:00 p.m.

Adhere to the Code of Academic Integrity. You may discuss background issues and general strategies with others and seek help from course staff, but the implementations that you submit must be your own. In particular, you may discuss general ideas with others but you may not work out the detailed solutions with others. It is never OK for you to see or hear another student's code and it is never OK to copy code from published/Internet sources. Moss (Measure Of Software Similarity) will be used for determining the similarity of programs to detect plagiarism in the class (`https://theory.stanford.edu/~aiken/moss/`). If you encounter some difficulties or feel that you cannot complete the assignment on you own, discuss with your classmates in Discussion forum on Canvas, or seek help from the course staff. You can complete this homework ***individually*** or ***in a group of two***.

When submitting your assignment, follow the instructions summarized in Section 6 of this document.

1 Introduction

The goal of this assignment is to implement particle filters for state estimation. You will use an autonomous robot DR20 and try to estimates its location given inputs from sensors and a model of the robot motion.

In this assignment, the scene model file is **not supported** on the **macOS** version of CoppeliaSim. Please finish this homework on **Windows** or **Linux**.

1.1 Particle Filters

In general, a particle filter is a statistical model used to track the evolution of a variable with possibly non-gaussian distribution. The particle filter maintains many copies (particles) of the variable, where each particle has a weight indicating its quality.

The particle filter is continuously iterated to improve the localization estimate and update localization after the robot moves. This happens in three steps: Prediction, Update, and Resample.

(1)Prediction: In the prediction step, a motion model is needed to predict the pose of the robot by only taking motion into account. Each particle is modified based on the robots motion model. The process is Markovian since the current pose is only dependent on one pose before. Initially the particles are uniformly sampled in the map, having no prior knowledge of the pose.

(2)Update: In the update step, a measurement model is needed to incorporate sensor information. You can simply update the particle's probability based on the expected and the observed readings. The weight of each particle will be updated such that the current belief of the robot pose is usually the weighted average of particle states.

(3)Resampling: In this step, all previous particles are discarded and new particles are chosen from a weighted distribution of the previous particles. Particles from the previous iteration with a higher weight are more likely to be chosen for the succeeding iteration. This discards low-probability particles and preserves high-probability cycles for the next iteration of the filter.

1.2 Motion Model

Figure 1 describes a simple motion model. The state of the robot is represented by position and orientation: $\mathbf{x} = [x, y, \phi]^\mathsf{T}$. The control signal of the robot is $u = [v, \omega]^\mathsf{T}$, where v is the linear velocity and ω is the

angular velocity. The equations for the motion model are as follows:

$$x_{t+1} = x_t + v\cos(\phi_t)\Delta t$$
$$y_{t+1} = y_t + v\sin(\phi_t)$$
$$\phi_{t+1} = \phi_t + \omega\Delta t$$

Expressed in the matrix form, we have

$$\mathbf{x}_{t+1} = \mathbf{F}\mathbf{x}_t + \mathbf{B}\mathbf{u}$$

where $\mathbf{x}_{t+1} = [x_{t+1}, y_{t+1}, \phi_{t+1}]^\mathsf{T}$, $u = [v, \omega]^\mathsf{T}$,$F = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$,$B = \begin{bmatrix} \Delta t\cos(\phi_t) & 0 \\ \Delta t\sin(\phi_t) & 0 \\ 0 & \Delta t \end{bmatrix}$

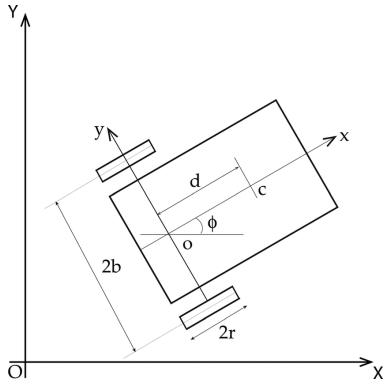


Figure 1: Two wheeled differential drive robot.

1.3 Assumptions and Simplifications

A number of simplifications and assumptions about the robot were made in order to simplify the simulation, which will not seriously affect the performance of particle filters.

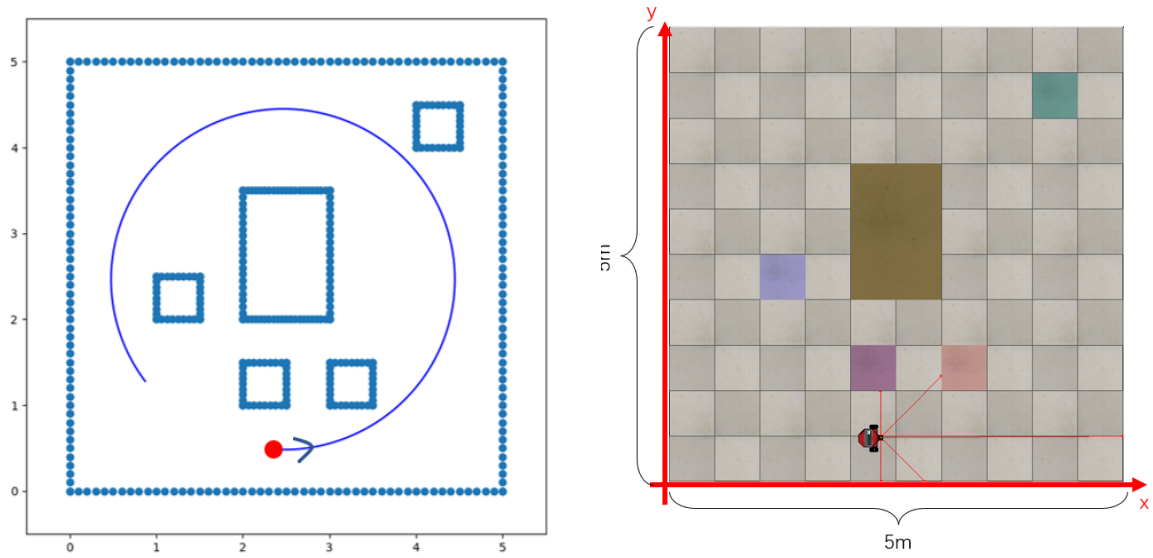
The robot is assumed to move in a square room with a size of $5m \times 5m$, as Figure 2 shows. A simple discrete map is created and the obstacles are placed in the corresponding locations in CoppeliaSim. For simplicity, we input a **fixed** control command $u = [v, \omega]^\mathsf{T} = [0.5, 0.25]^\mathsf{T}$ to the robot, so it will follow a trajectory of circle with a radius of 2.

For the purposes of the simulation, the error signals in the motion model are assumed to be gaussian and are added to the input control command.

During the update step of the particle filter, the weight of the particle is determined by its difference from the "true" reading and the expected reading. And this "true" reading is the actual value of the robot with added error, which is assumed to be a normal distribution.

The scanning angle of the LiDAR is set to be 180. For simplicity of computation, the number of lidar beams is set to 5. Each lidar reading is represented by the distance value in counterclockwise order. For example, in Figure 2(b), the lidar reading can be expressed as

$$\mathbf{data} = [0.4813, 0.6849, 2.6704, 0.9539, 0.5187]$$



(a) (b)

Figure 2: Map

1.4 The Challenges

The mobile robot localization problem is to determine the pose (orientation and position) of the robot given the map of the environment, sensor data, a sensor error model, movement data, and a movement error model. It is a very basic problem of robotics since most of robot tasks require the position of the robot. There are three types of localization problems in increasing order of difficulty.

1) **Local Position Tracking** The initial pose of the robot is assumed to be known. Since the uncertainties are confined to region near the actual pose, this is considered to be a local problem.

2) **Global Localization** In contrast to local position tracking, global localization assumes no knowledge of initial pose. It subsumes the local problem since it uses knowledge gained during the process to keep tracking the position.

3) **Kidnapped Robot Problem** The kidnapped robot problem arises from the movement of a successfully localized robot to a different unknown position in the environment to see if it can globally localize. Thus it is more difficult than global localization problem since the robot has a strong but wrong belief in where it is.

The first two problem could be solved by MCL (Monte Carlo Localization) algorithm, and the third problem could be solved by AMCL (Augmented Monte Carlo Localization) algorithm. In this homework, the score consists of the following parts:

1. Basic tasks: The main part of this homework.

- If you solve the **Local Position Tracking problem**, you will get **90 points**.
- If you solve the **Global Localization problem**, you will get the remaining **10 points**.

2. Bonus tasks: You can choose one of the following two tasks for bonus points. Bonus points will not accumulate. That is, complete both of the two tasks will also get 10 bonus points.

- In the **Global Localization problem**, calculate the time spent each time for robot localization, if you meet the requirements of real-time (up to 1 second interval per iteration), you will get **extra 10 points** for bonus.
- If you solve the **Kidnapped Robot Problem**, you will get **extra 10 points** for bonus.

1.5 Provided File List

- 1-HW3_assignment.pdf: The introduction and description of homework3.
- 2-MCL_local.py: The code framework of local localization. You need to supplement your code on this framework.
- 3-MCL_global.py: The code framework of global localization. Almost the same as the previous file, only a boolean variable named **Nearby_flag** is changed to False to generate the particles on the whole map.
- 4-AMCL.py: The code framework of AMCL to solve the Kidnapped Robot Problem.
- 5-pf.ttt: The scene file in CoppeliaSim for homework 3.
- 6-Report_Template: A latex template for report.

1.6 Submission File List

- MCL_local.py: The completed code for MCL algorithm to solve the **Local Position Tracking** problem.
- MCL_local.mp4: The video of MCL algorithm. We need to see the process that the particles gradually converge to the real position of the robot.
- MCL_global.py: The completed code for MCL algorithm to solve the **Global Localization Problem**.
- MCL_global.mp4: The video of MCL algorithm for global localization. We want to see the process that the particles sprinkle randomly on the whole map at the beginning, and finally converge to the real position of the robot.
- MCL_global_realtime.py (optional): The completed code for MCL algorithm of global localization with real-time requirements (up to 1 second interval per iteration).
- MCL_global_realtime.mp4 (optional): The video of MCL algorithm for global localization. We want to see the time spent for each iteration in the video.
- AMCL.py (optional): The completed code for AMCL algorithm to solve the **Kidnapped Robot Problem**.
- AMCL.mp4 (optional): After the particle correctly converges to the actual position of the robot, drag the robot in CoppeliaSim with the mouse to simulate the kidnapping (move at least 2m), show that the particles can converge to the position after kidnapping.
- HW3_report.pdf: Report for homework 3.

That is, for basic taks, you need to submit at least 5 documents. If you want to get the bonus 10 points, you need to submit 7 documents.

2 Basic Tasks: Monte Carlo Localization Algorithm[90 + 10 points]

2.1 Description

Monte Carlo Localization is an algorithm that begins with a set of random hypotheses about where the robot might be all over the map and in any heading. Each of these hypotheses is called a particle, as this technique derives from the general technique called Particle Filtering. In this section, you will implement the MCL algorithm for the autonomous robot DR20. The map and the simulated environment is given. You can update the particle's probability based on the expected and the observed readings.

You can run the Python file **MCL_local.py** with CoppeliaSim opened, and you will see the particles (signed as green dots) are uniformly scattered around the robot, and the robot walks in a circular path (signed as blue lines). The main part of the homework is in the function named **pf_localization**. In this function, three steps should be performed: Prediction, Update and Resample.

The following is a description of some variables in the code file:

- **NP**: The number of particles. You can change this value, but the bigger the number, the longer the iteration time. However, in the two basic tasks, we have no requirement on the iteration time.
- **Nearby_flag**: A boolean variable in function **generate_particles**. If it is set to True, then it's a local position tracking problem. Particles will only be scattered around the robot in the process of initializing particles. If you solved the problem, you will get **90 points**. If you set the variable to False, and finish the global localization, you will get the remaining **10 points**.
- **DT**: The time interval between robot movements. The value must be the same as the one in CoppeliaSim. In this task, DT is set to 0.1 s (100 ms in CoppeliaSim).
- **Q, R**: LiDAR range error and the input error. You may use these two variables in Prediction steps.
- **Q_sim, R_sim**: The simulation parameters. You cannot change them in your code.

2.2 Result Visualization and Analysis

Result Visualization: You are required to implement the particle filter algorithm in **MCL_local.py** and **MCL_global.py**, and complete the visualization of the results according to the following requirements:

1. Plot the real robot path and the estimated path.
2. Plot the LiDAR points on the map.
3. Plot the particles, and show the weight of each particles in any way you like (e.g. radius represents weight).

You need to record the screen to show the process of localization in **MCL_local.mp4** and **MCL_global.mp4**

Result Analysis: You need to explain how you complete your algorithm, including the strategy of resampling, the update method of particle weight and so on.

3 Bonus Tasks [Extra 10 Points]

There are two bonus tasks, you can **choose one of them to get the extra 10 points**. In these two tasks, we have no restrictions on code changes, that is, you can modify the existing framework, rewrite the existing functions, or use other libraries, etc. Bonus points will not accumulate. That is, complete both of the two tasks will also get 10 bonus points.

3.1 Speed Up The Global Localization!

3.1.1 Description

The goal of the MCL algorithm introduced in this assignment is to perform global localization originally. However, due to the exist of Python Global Interpreter Lock or GIL, Python allows only one thread to hold the control of the interpreter. So if you use too many particles, the iteration time is long, and maybe you could try **multiprocess** or use other algorithms to speed up the localization process. If your algorithm is able to meet the requirements of real-time (up to 1 second interval per iteration), you will get extra 10 points for bonus.

3.1.2 Result Visualization

You are required to implement the particle filter algorithm in **MCL_global_realtime.py** and record a video named **MCL_global_realtime.mp4** to show that the iteration time meets the real-time requirements.

3.2 Solving The Kidnapped Robot Problem!

3.2.1 Description

The original MCL algorithm does not have the ability to recover from kidnapping. To prevent the localization algorithm from failing by getting stuck on a wrong global localization – or recovering after a robot kidnapping, AMCL (Augmented Monte Carlo Localization) incorporates two recovery parameters α_{slow} and α_{fast}. The higher the α_{slow} and α_{fast} values, the more likely the algorithm is to include random particles. The alpha parameters multiply a divergence of the weighted average of the measurement model and the short- and long-term average of the measurement likelihood, ω_{fast} and ω_{slow}, respectively, resulting in an update in the last two averages values. The rate of the short-term and long-term determines the probability of adding particles in random poses, in the form of $\max(0.0, 1 - \omega_{fast}/\omega_{slow})$.

You can modify the MCL algorithm to deal with the kidnapped robot problem. The pseudocode of AMCL algorithm is shown in the Figure. 3. In this task, you have no limitation of the number of particles or iteration time.

3.2.2 Result Visualization

You are required to complete the AMCL algorithm in **AMCL.py** file and record a video: Drag the robot in CoppeliaSim with the mouse to simulate the kidnapping (move at least 2 meters), show that the particles can converge to the position after kidnapping.

Algorithm Augmented_MCL($\mathcal{X}_{t-1}, u_t, z_t, m$)**:**

static $w_{\text{slow}},\ w_{\text{fast}}$

$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$

for $m = 1$ to M do

  $x_t^{[m]} =$ **sample_motion_model**$(u_t, x_{t-1}^{[m]})$

  $w_t^{[m]} =$ **measurement_model**$(z_t, x_t^{[m]}, m)$

  $\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$

  $w_{\text{avg}} = w_{\text{avg}} + \frac{1}{M}\ w_t^{[m]}$

endfor

$w_{\text{slow}} = w_{\text{slow}} + \alpha_{\text{slow}}(w_{\text{avg}} - w_{\text{slow}})$

$w_{\text{fast}} = w_{\text{fast}} + \alpha_{\text{fast}}(w_{\text{avg}} - w_{\text{fast}})$

for $m = 1$ to M do

  with probability $\max(0.0,\ 1.0 - w_{\text{fast}}/w_{\text{slow}})$ do

    add random pose to $\mathcal{X}_t$

  else

    draw $i \in \{1, \ldots, N\}$ with probability $\propto w_t^{[i]}$

    add $x_t^{[i]}$ to $\mathcal{X}_t$

  endwith

endfor

return $\mathcal{X}_t$

Figure 3: The pseudocode of AMCL algorithm.

4 Code, Demo Video and Report

Code: You can only edit your code between "### START CODE HERE ###" and "### END CODE HERE ###" in **MCL_local.py**, **MCL_glabal.py**. Please ***DO NOT*** revise other parts of the code. The code block to be completed is described below. For bonus tasks, there are no restrictions on code changes, that is, you can edit your code anywhere.

```
###  START CODE HERE  ###
# You can tune the hyper-parameter, and define your utility function or class in this block
                                                    if necessary.

# Particle filter parameter
NP = 400  # Number of Particle
NTh = NP / 2.0  # Number of particle for re-sampling

# Set the random seed to ensure the repeatability.
seed=1
np.random.seed(seed)

#  Estimation parameter of PF, you may use them in the PF algorithm. You can use the
                                                    recommended values as follows.
Q = np.diag([0.15]) ** 2  # range error
R = np.diag([0.1, np.deg2rad(10)]) ** 2  # input error

###  END CODE HERE  ###
```

```
def pf_localization(px,pw,data,u):
    """
```

```
    Localization with Particle filter. In this function, you need to:

    (1) Prediction step: Each particle predicts its new location based on the actuation
                                                command given.
    (2) Update step:     Update the weight of each particle. That is, particles consistent
                                                with sensor readings will have higher
                                                weight.
    (3) Resample step:   Generate a set of new particles, with most of them generated around
                                                 the previous particles with more weight.
                         You need to decide when to resample the particles and how to
                                                                    resample the
                                                                    particles.

    Argument:
    px -- A 3*NP matrix, each column represents the status of a particle.
    pw -- A 1*NP matrix, each column represents the weight value of correspoding particle.
    data -- A List contains the output of the LiDAR sensor. It is represented by a distance
                                                value in counterclockwise order.
    u -- A 2*1 matrix indicating the control input. Data format: [[v] [w]]

    Return:
    x_est -- A 3*1 matrix, indicating the estimated state after particle filtering.
    px -- A 3*NP matrix. The predicted state of the next time.
    pw -- A 1*NP matrix. The updated weight of each particle.
    """

    ### START CODE HERE ###
    # You can define your utility function and class if necessary .
    for ip in range(NP):
        #  Prediction step: Predict with random input sampling
        pass

        #  Update steps: Calculate importance weight
        pass

    pw = pw / pw.sum()  # normalize
    x_est = px.dot(pw.T)

    # Resample step: Resample the particles.
    pass

    ###  END CODE HERE  ###
    return x_est,px, pw
```

```
def re_sampling(px, pw):
    """
    Robot generates a set of new particles, with most of them generated around the previous
                                                particles with more weight.

    Argument:
    px -- The state of all particles befor resampling.
    pw -- The weight of all particles befor resampling.

    Return:
    px -- The state of all particles after resampling.
    pw -- The weight of all particles after resampling.
    """
    ### START CODE HERE ###


    ###  END CODE HERE  ###
    return px, pw
```

```
###  START CODE HERE  ###
# Visualization
if True:
    plt.cla()
```

```
    # for stopping simulation with the esc key.
    plt.gcf().canvas.mpl_connect(
        'key_release_event',
        lambda event: [controller.stop_simulation() if event.key == 'escape' else None])
    x, y = zip(*room)
    plt.scatter(x, y)
    plt.plot(np.array(h_x_true[0, :]).flatten(),
             np.array(h_x_true[1, :]).flatten(), "-b")

    # Plot the particles
    plt.scatter(px[0,:],px[1,:],color = 'g',s=5)

    plt.axis("equal")
    plt.grid(True)
    plt.xlim(-0.5,5.5)
    plt.ylim(-0.5,5.5)
    plt.pause(0.01)
###  END CODE HERE  ###
```

Demo video: Please record your screen to show the results of your localization. The videos should be in mp4 format and a 10 MB max in total (you can speed up and compress the videos), named as MCL_local.mp4, MCL_global.mp4, MCL_global_realtime.mp4 (optional) and AMCL.mp4 (optional).

Report: Summarize the process and results of the homework, including but not limited to:

- Your understanding and analysis of the problem.
- The description of the implementation of your localization algorithms.
- The parameter you used in your algorithms.

5 Discussion and Question

You are encouraged to discuss your ideas, and ask and answer questions about homework 3. A new post for this assignment "Homework 3 Discussion" is opened in the Discussion Forum on Canvas. If you encounter any difficulty with the assignment, try to post your problem for help. The classmates and the course staff will try to reply. `https://oc.sjtu.edu.cn/courses/35320/discussion_topics`

6 Submission Instructions

1. For documents to be submitted, refer to the Submission File List in Section 1.6. Zip your python code files, demo videos and report files to a zip file named as **HW3_Name1_ID1_Name2_ID2.zip** for a group, and **HW3_Name1_ID1.zip** for individually. If you are working as a group of two, write both your names and student IDs in the name of zip file and on the cover of the report, and submit once is okay.
2. Upload the file to "Homework 3: Particle Filter" on Canvas:
   `https://oc.sjtu.edu.cn/courses/35320/assignments`
   Due: Dec. 15th 18:00 p.m.