Assignment 3

(100 pts is the perfect score - 20% of the course credit)

Deadline: EoD (i.e., 11:59 PM) Monday, December 11

Submission Process: You are asked to upload a zip file on Canvas containing all necessary files of your submission. Only 1 student per team needs to upload a submission. *Aim for 2 students per team.*

You should create a top-level folder named with your NETID, (e.g., xyz007-abc001 for teams of 2 students or just xyz007 for students working independently). Include at least the following files by following this folder structure:

- xyz007-abc001/maps/landmark_[0-4].npy (5 maps in total for problem 1.1)
- xyz007-abc001/controls.py (for problem 1.2)
- xyz007-abc001/controls/controls_[0-4]_[1-2].npy (10 control sequences for problem 1.2)
- xyz007-abc001/simulate.py (for problem 2.1)
- xyz007-abc001/gts/gt_[0-4]_[1-2].npy (10 ground truth trajectories for problem 2.1)
- xyz007-abc001/readings/readings_[0-4]_[1-2]_[L-H].npy (20 sensor readings for problem 2.1, 10 for low noise level and 10 for high noise level)
- xyz007-abc001/dead_reckoning.py (for problem 2.2)
- xyz007-abc001/dr/dr_X_Y_Z.mp4 (at least 5 example visualizations of dead reckoning solutions - at least 1 per map, at least 2 for low or high noise)
- xyz007-abc001/particle_filter.py (for problem 3.1)
- xyz007-abc001/estim1/estim1_X_Y_Z_N.npy (40 example solutions of the particle filter algorithm, 2 for each one of the setups above for low and high number of particles N assuming *known* initial robot pose)
- xyz007-abc001/video1/particles_X_Y_Z_N.mp4 (at least 5 example particle visualizations for the above setups for the same problems you visualized the dead reckoning solution)
- xyz007-abc001/particle_filter_kidnapped.py (for problem 3.1)
- xyz007-abc001/estim2/estim2_X_Y_Z_N.npy (40 example solutions of the particle filter algorithm, 2 for each one of the setups above for low and high number of particles N assuming *unknown* initial robot pose)
- xyz007-abc001/video2/particles_X_Y_Z_N.mp4 (at least 5 example particle visualizations for the above setups for the same problems you visualized the dead reckoning solution)
- xyz007-abc001/evaluate.py (for problem 4)

- xyz007-abc001/eval1/eval1_X_Y_Z_N.mp4 (at least 5 example visualizations of ground truth robot poses versus estimates of the particle filter given *known* initial robot poses - for the same problems you visualized the dead reckoning solution)
- xyz007-abc001/eval2/eval2_X_Y_Z_N.mp4 (at least 5 example visualizations of ground truth robot poses versus estimates of the particle filter given *unknown* initial robot poses - for the same problems you visualized the dead reckoning solution)
- xyz007-abc001/others/* (if any, like .tex)

Put all required files as instructed and all other files under xyz007-abc001/others/. Please zip this xyz007-abc001 folder and submit the xyz007-abc001.zip on Canvas. You can also create a utils.py file that contains helper functions under the xyz007-abc001 folder. On the first page of the report's PDF indicate the name and Net ID of the students in the team (up to 2 students - aim for 2 students).

Late submissions will not be graded. Keep file size under 50 MB. **Failure to follow the rules will result in a lower grade (-10 pts) on the assignment.**

Extra Credit for LaTeX: You will receive 6% extra credit points if you submit your answers as a typeset PDF, i.e., one generated using LaTeX. For typeset reports in LaTeX, please also submit your .tex source file together with the report, e.g., by adding a report.tex file under the xyz007-abc001 folder. There will be a 3% bonus for electronically prepared answers (e.g., using MS Word, etc.) that are not typeset. Remember that these bonuses are computed as a percentage of your original grade, i.e., if you were to receive 50 points and you have typeset your report using LaTeX, then you get a 3-point bonus. If you want to submit a handwritten report, scan it or take photos of it and submit the corresponding PDF as part of the zip file on Canvas. You will not receive any bonus in this case. For your reports, do not submit Word documents, raw ASCII text files, or hardcopies etc. If you choose to submit scanned handwritten answers and we cannot read them, you will not be awarded any points for the unreadable part of the solution.

Requirements: Standard Python libraries are allowed. Additionally, Numpy, Scipy and Matplotlib can be used. Please use the following script to build your Python environment under conda.

Setup python environment

```
#!/bin/bash
conda create -n cs460 python=3.10.12 numpy=1.25.2 matplotlib=3.7.2 scipy=1.11.1 -y
conda activate cs460
# run your code from here
```

This Python environment will be used for all assignments. We expect to run your code on ilab. If we cannot run your code in the Python environment as described above, then 0 credits will be awarded for the programming tasks. You can also not use conda, as long as your Python and library versions match.

Overview of Robot Localization Assignment 3

The assignment involves the following components (you build the simulator, and this simulator returns data to the robot, the robot uses this data to do localization):

1. The first component of your solution should setup (i) an obstacle-free environment with a rectangular boundary and recognizable point landmarks, as well as (ii) control sequences for a robot to move inside this environment. In particular, your script can randomly place the landmarks in the environment and pick a random initial pose for the robot. Your software should compute a sequence of piece-wise constant controls for the robot that allows it to move inside the environment without hitting the rectangular boundary of the environment assuming perfect execution of these controls. Note that the landmarks are not obstacles, i.e., the robot can go over them. You are asked to generate multiple test environments, i.e., different landmark maps and different robot control sequences. You should be able to work on this component early!

2. The second component executes the robot's control sequence given the corresponding map of recognizable landmarks and generates simulated odometry and landmark observation readings. The execution of the control sequence, however, is imperfect. Given a motion model that expresses the noise of the actuators, your simulation environment should integrate the control sequence with noise. As you simulate the robot's motion, you will generate:

   - the "ground truth" poses that the robot goes over when it executes the noisy controls, given some time discretization of the trajectory;
   - for the same time discretization, you should also generate a noisy estimation of how much the robot has moved between two poses on the trajectory given simulated, noisy odometry sensors; and
   - for the same time discretization, your should also generate noisy measurements of the distance and angle to the recognizable landmarks in the environment given simulated, noisy sensors.

   The odometry measurements and the landmark observations will correspond to the information available to a localization algorithm. The ground truth poses are generated only for evaluation purposes. You are asked to experiment for different levels of noise in terms of both the motion and the observation models.

3. The third component of your software solution should be able to read the noisy odometry and landmark measurements files and execute a robot localization algorithm based on the Particle Filter approach. The algorithm's objective is to estimate the true robot poses, i.e., position and orientation, that gave rise to the corresponding noisy measurements. You are asked to experiment for different algorithmic parameters, e.g., different number of particles, and also measure the computational overhead of the approach.

4. The final component will evaluate the performance of the localization algorithm. It will use the output generated as part of the third component (i.e., the robot pose estimates of the localization algorithm) and the ground truth poses of the robot generated during the second component to estimate the error in the algorithm's estimate.

Extra credit will be awarded for implementing an Extended Kalman Filter as an alternative to the Particle Filter approach and comparing the performance of the two.

For all animations, please make sure that are within one minute of duration. Do no use gif, please use mp4 for animation. Failure to follow the rules will lose 25% points for the problem.

1 Setup maps and generate controls (15 pts)

Your robot is a rigid body that can rotate and translate in 2D given the dynamical expression indicated below in Eqs. (1-3). Create a 2D rigid body with dimensions: $0.2 \times 0.1 m^2$. The reference point for the rectangle is its geometric center. Please check recitation 6 for a simplified version of a differential drive robot, which can be used as a guide for the implementation of the dynamics. Instead of reasoning for the individual velocity of each wheel, the controls are assumed to be the forward linear velocit and the angular velocity of the robot. When the short side of the rigid body points (heading) to the right, the robot has an orientation of $\theta = 0$. Counterclockwise rotation is in the positive direction. The range of heading should be in $[-pi, pi)$, where π and $-\pi$ represent the same robot state. Consider the following:

- $q(t)$ is the robot configuration at time t;
- u is the control input (the controls can change instantly), corresponding to:
  - v is the car's velocity along its current heading direction (valid range: $-0.5 \leq v \leq 0.5$);
  - ϕ is the angular velocity (valid range: $-0.9 \leq \phi \leq 0.9$);
- $L = 0.2$ is the wheelbase, i.e., the distance between the left and right axles;
- $dt = 0.1$ is the time step to update the robot's configuration, when you are integrating its dynamics.

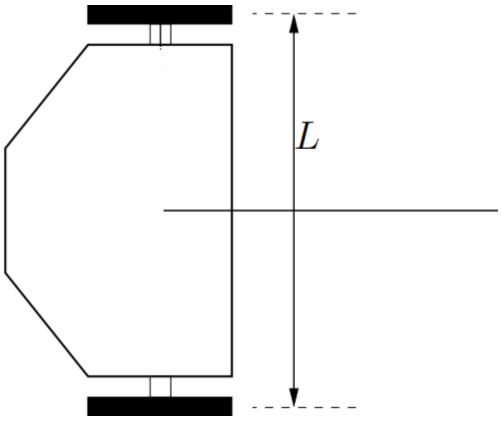

Figure 1: Example robot model. The long side of car should be $L = 0.2$, the short side of car should be 0.1. Note that these are different dimensions relative to those used in previous assignments. In your implementation, your car should be like a rectangle from Fig. 4

$$q(t) = \begin{bmatrix} x(t) \\ y(t) \\ \theta(t) \end{bmatrix}, \quad u = \begin{bmatrix} v \\ \phi \end{bmatrix} \tag{1}$$

$$\dot{q}(t) = \begin{bmatrix} v\cos(\theta(t)) \\ v\sin(\theta(t)) \\ \phi \end{bmatrix} \tag{2}$$

$$q(t + dt) = q(t) + \dot{q}(t)dt \tag{3}$$

The workspace limits are $[0, 2]$ for both x and y.

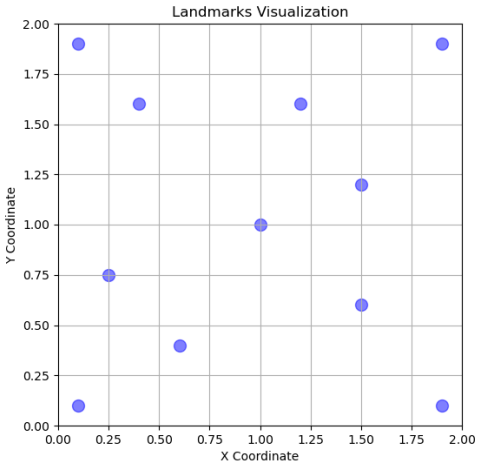

Figure 2: Landmark_0

1.1 Generate 4 maps with landmarks (5 pts)

Generate 4 maps with randomly placed point landmarks inside the range $\{[0, 2] \times [0, 2]\}$. Please name the corresponding maps as [landmark_1.npy, landmark_2.npy, ..., landmark_4.npy]. The first two maps should have 5 landmarks, the last two should have 12 landmarks. A map landmark_0.npy with 11 landmarks is provided to you as an example and visualized in Fig. 2. The corresponding file stores an $[11 \times 2]$ matrix, where 11 is the number of landmarks for this map, and the columns store the (x, y) coordinates of each landmark. Each landmarks' row index will be used as the landmark ID. Overall, you will have 5 maps to experiment with. So, for your first two maps, you should generate and store a matrix of dimensions $[5 \times 2]$ and for the last two you should generate a matrix of dimensions $[12 \times 2]$.

1.2 Generate robot control sequences (10 pts)

Create an executable Python file called controls.py. We will run your code as:

```
python controls.py
```

This script should read the landmark maps from the folder "map" and generate two control sequences for the robot for each map. Each control sequence will first indicate an initial robot pose and then a sequence of controls of 20 seconds total duration.

The initial robot pose is a random (x, y, θ) configuration that places the robot inside the rectangular region and at least a distance 0.2m from the rectangular boundary. Keep sampling initial robot poses until this condition is satisfied.

For the sequence of controls and since our integration step is $dt = 0.1sec$, each control sequence U will correspond to 200 control inputs $U = [u_0, u_1, ..., u_{199}]$, where $u_i = (v, \phi)$. You are asked to impose that the control input will remain constant for 2 seconds and then change. In other words, you will only have to sample 10 random control inputs total: the first control input is applied for the first 20 integration steps of duration 0.1 seconds each, then you sample another control input for the next 20 integration steps, until you hit a duration of 20 seconds and 200 steps, i.e., $u_{[0-19]}$ have the same control values, $u_{[20-39]}$ have the same values and so on. Note that your control inputs should not exceed the maximum linear or angular velocity limits mentioned above.

Furthermore, you are asked to generate control sequences that will keep the robot inside the rectangular environment and at least a distance of 0.2 from the rectangular boundary. As you

sample controls, integrate the robot's dynamics forward and compute the resulting robot poses given the corresponding controls. If a control moves the car closer to the boundary than a distance of 0.2m for any of the resulting poses, then this is not a valid control sequence. It is up to you to make sure that this condition is satisfied. You can treat this as a motion planning problem similar to the last assignment, where you treat as a collision every time that the robot gets to the boundary closer than 0.2m. Or you can just keep resampling controls every time that one of them results in your car get that close to the boundary.

You should generate 10 control sequences total, i.e., two control sequences for each of the 5 maps created by you and two control sequences for the "landmark_0.npy" map we provide. Then, there should be 10 plots visualized over the corresponding landmark map. These plots will visualize the 201 positions that the robot goes over (i.e., just the (x, y) coordinates of the robot), when the controls are integrated forward from the corresponding initial robot pose. Fig. 3 provides a visualization of a robot path for random controls and a random initial pose (but does not show the landmarks, which you should also visualize).

Beyond the visualizations, your script should also store each control sequence in a separate output file. Store these files inside a folder "controls" using the filenames "controls_X_Y.npy", where:

- X is the id of the landmark map (from 0 to 4) for which the sequence is built for and
- Y is the id of the control sequence (from 1 to 2).

Each of these files will have 201 rows, where the first row will indicate the random initial robot pose (i.e., (x, y, θ) coordinates) and the remaining 200 rows will indicate the controls you have sampled (i.e., v, ϕ values).

In your report, please include visualizations of the 10 control sequences over the corresponding landmark maps you adopted for the rest of the assignment. Please also describe how did you go about generating the control sequences and the corresponding visualizations.

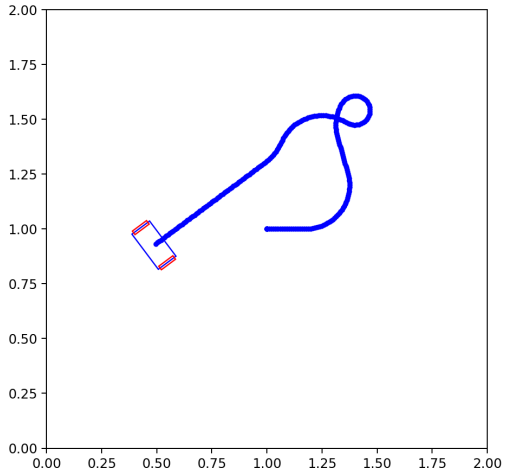

Figure 3: For section 1 of the assignment, you are asked to visualize the path that the robot would follow if the planned controls are executed perfectly. Here you can see an example drawing of the robot's planned path for random controls. A landmark map is not shown here but you are asked to visualize a control sequence in the corresponding landmark map.

2 Integrate Noisy Dynamics and Collect Sensor Readings (30 pts)

When a robot executes controls there are errors that arise in the process due to inaccuracies and uncertainty. You are asked to model the corresponding noise given an actuation model. You should integrate forward a noisy version of the robot's dynamics to generate the "ground truth" poses that a robot will go over at the time resolution of the integration step, i.e., every 0.1 seconds. For the same set of poses, you are also asked to simulate (i) a noisy odometry sensor that measures how much the robot has moved between two consecutive poses, and a (ii) noisy landmark detection sensor that measures the distance and angle to landmarks in the environment given the robot's pose. For each one of the 10 control sequences you generated during part 1 and the corresponding landmark map, you are asked to output the list of "ground truth" poses as well as the simulated measurements in files that will be used in the next components of the assignment.

2.1 Simulate Robot's Motion and Measurements (20 pts)

Create an executable Python file called simulate.py. We will run your code as:

```
python simulate.py --plan controls/controls\_X\_Y.npy
                   --map maps/landmark\_X.npy
                   --execution gts/gt\_X\_Y.npy
                   --sensing readings/readings\_X\_Y\_Z.npy
```

The script will load a control sequence "controls/controls_X_Y.npy" and the corresponding map of landmarks "maps/landmark_X.npy", for which the control sequence has been generated for. It will then store:

1. The ground truth poses that the robot will go over in the output file "gts/gt_X_Y.npy ". This requires integrating the robot dynamics by starting from the initial robot pose indicated in the planned control sequence and using a noisy version of the controls as specified in Section 2.1.1. The number of poses inside the files "gts/gt_X_Y.npy" should be 201, i.e., you start from the initial robot pose of "controls_X_Y.npy" and then you perform the integration of the noisy dynamics.

2. The measurements collected by the robot during this process will be stored in the output file "readings/readings_X_Y_Z.npy". The parameter Z corresponds either the letter "L", for low level of noise in the observation models, or the letter "H", for high level of noise in the observation models. This file will contain 401 rows of information, which correspond to the following:

- The first row corresponds to the initial robot pose in "controls/controls_X_Y.npy" i.e., the robot coordinates (x, y, θ) (this is also the same first row in "gts/gt_X_Y.npy").
- Then, the script alternates between 200 pairs of rows that report:
  - First, an odometry measurement of the form u_i^{sensed}:

    $$\{v_i^{sensed}, \phi_i^{sensed}\}$$

    indicating a noisy estimate of how much the robot has moved between poses $i-1$ and i. See Section 2.1.2 for details on how to generate the odometry measurements;
  - Next row corresponds to landmark observation measurements of the form $l_i =$

    $$d_i^0, \alpha_i^0, \ldots, d_i^N, \alpha_i^N$$

    indicating noisy distance d_i^j and angular α_i^j measurements of the j^{th} landmark from the i^{th} pose of the robot during the execution of the corresponding trajectory. Each such row reports noisy measurements for N landmarks, where N is either 5, 11 or 12 landmarks depending on the map you are loading. See Section 2.1.3 for details on how to generate the landmark observation measurements.

Execute your script for all 10 control sequences you generated during part 1 of this assignment and store the resulting ground truth poses and measurements as part of your submission. Below are details on how to generate the corresponding files and what probabilistic models of noise to adopt in your implementation.

2.1.1 Actuation model

At each step, given a control input indicated in the "controls/controls_X_Y.npy" file, your simulated robot will execute a noisy version of that command. This can be due to friction and other unmodeled factors. For example, if you ask the robot to move forward with $v = 0.1$, with added noise, the robot may move forward only with $v = 0.095$.

Eq. 4 indicates the nature of the noise introduced relative to Eq. 1 for the execution of the robot's motion. This is the noise for each control u_i of the control sequence that is passed as input. The resulting noisy control $u_i^{executed}$ is used to compute the next pose. You can use the res t of Eqs. (1-3) to integrate the dynamics forward in order to generate the ground truth poses.

The noise in linear and angular velocity will be applied separately. Furthermore, the noise is assumed to follow an independent zero-mean Gaussian distribution $\epsilon \sim N(0, \sigma^2)$ with standard deviation σ. For the linear velocity noise use a standard deviation of $\sigma_v = 0.075$. For the angular velocity noise use a standard deviation of $\sigma_\phi = 0.2$.

$$u_i^{executed} = \begin{bmatrix} v_i^{planned} + \epsilon_v \\ \phi_i^{planned} + \epsilon_\phi \end{bmatrix} \tag{4}$$

Please clamp the max speed after adding noise within the specified valid bounds of Part 1. The noise should be sampled at every time step. In your previous control sequences, $u_{[0-19]}$ have the same value, but now with added noise, all $u \in u_{[0-19]}$ will end up not having the same value.

2.1.2 Odometry model

The robot is making use of wheel encoders to estimate how far it has traveled during a time step, i.e., again every 0.1 seconds. In order to generate the noisy odometry readings $u_i^{sensed} = \{v_i^{sensed}, \phi_i^{sensed}\}$ use Eq. 5 below. The noise model of the odometry sensor is similar to the actuation noise but the odometry measurements u_i^{sensed} are different than the executed controls $u_i^{executed}$.

$$u_i^{sensed} = \begin{bmatrix} v_i^{executed} + \epsilon_{ev} \\ \phi_i^{executed} + \epsilon_{e\phi} \end{bmatrix} \tag{5}$$

We will refer to the noise parameter of the odometry model as σ_{ev} and $\sigma_{e\phi}$. You are asked to generate two versions of sensor measurements for each control sequence, one for low level of assumed sensor noise and one for a higher level:

- For the output file "readings/readings_X_Y_Z.npy", where "Z=L", i.e., the low level noise model, set $\sigma_{ev} = 0.05$ and $\sigma_{e\phi} = 0.1$.
- For the output file "readings/readings_X_Y_Z.npy", where "Z=H", i.e., the high level noise model, set $\sigma_{ev} = 0.1$ and $\sigma_{e\phi} = 0.3$.

See an example effect of the above noise model in Fig. 4.

If the planned controls are 0, then your actuation and odometry should still be 0 regardless of the noise.

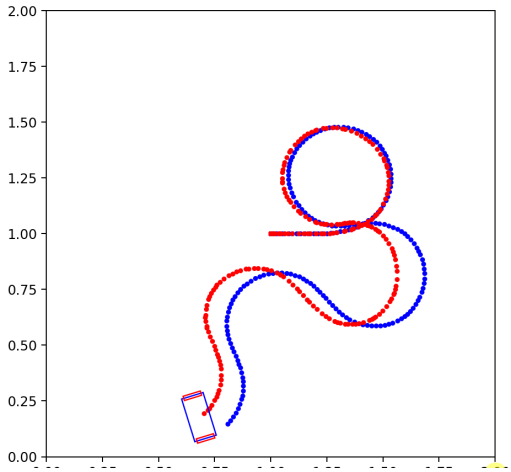

Figure 4: Example effect of noise in the motion of the robot. Blue indicates “ground truth”.

2.1.3 Observation Model

It is assumed that the robot can measure a distance and an angle to a visible and identifiable landmark in the environment. Landmarks are considered as points, so you do not need to worry about collisions. We assume that the robot can observe the landmarks anywhere inside the environment.

You should implement a landmark observation function, which takes as input the current robot' state and the landmarks' global coordinates and outputs the distance d_i^j to landmark j at time step i and the angular measurement α_i^j for the same landmark.

```
def landmark_sensor(ground_truth_x, ground_truth_y, ground_truth_theta, landmarks):
    ### your code
    return landmarks_local
```

Consider Fig. 5 as the setup where there are 3 landmarks in the map and the robot is actually pointing up at the corresponding pose. In this case, your function should return 6 values:

$$[[0.45, \frac{\pi}{6}], [0.7, \frac{4\pi}{6}], [0.3, \frac{-\pi}{2}]]$$

corresponding to the distance and angular measurements for landmarks 1, 2 and 3 in that order. In the figure, we used degrees to report the angular measurements. But your output should be in radians as shown in the list above. Pay attention that these angular measurements are in the robot's local coordinate system.

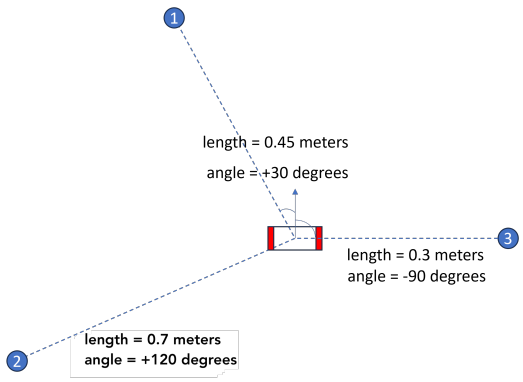

Figure 5: Example for the landmark sensor. The robot can detect all landmarks in the scene.

The above function returns the true distance and direction to the landmarks given the robot's ground truth pose. For what is stored in the readings file, i.e., the sequence $d_i^0, \alpha_i^0, \ldots, d_i^N, \alpha_i^N$, you are asked to add noise to the landmark sensor using again a Gaussian assumption, where we set $\sigma_l = 0.02$ for the distance estimate and $\sigma_\alpha = 0.02$ for the direction estimate. For example, the ground truth observation may detect the landmark 1 at a (distance,direction) pair of $(0.45, 30)$. The actual, noisy observation, however, may return the landmark 1 with values $(0.44, 29)$. The noise is applied independently between distance and direction measurements, i.e., independent zero-mean Gaussian noise is used for each measurement. Again, you should use radians in your program for direction measurements.

Beyond describing in your report how you implemented your landmark sensor, please complete the visualization described in problem 2.2 below to get full points for the implementation of the observation model. (i.e., 50% of the points will depend on the output of your visualization).

2.2 Visualize the dead reckoning solution (10 pts)

Create an executable Python file called dead_reckoning.py. We will run your code as:

```
python dead_reckoning.py --map maps/landmark\_X.npy
                         --execution gts/gt\_X\_Y.npy
                         --sensing readings/readings_X_Y_Z.npy
```

Your program should pop up a plot that will show: (a) the true landmark locations according to the "landmark_X.npy" file, together with (b) an animation of the ground truth motion of the robot according to the "gts/gt_X_Y.npy" file and (c) a synchronous animation of the dead reckoning solution given the readings file "readings_X_Y_Z". The dead reckoning solution integrates the odometry measurements to update the estimated robot pose. For each estimated robot pose it should also visualize the landmark observation.

The X, Y, and Z arguments in the files follow the same rules as above, i.e., X is the id of the landmark map, Y is the id of the control sequence, and Z is the level of noise (L or H).

Your program should animate the robot for a duration of 20 seconds given the information available in the input files in a way similar to Fig. 4, i.e., simultaneously show the ground truth pose and the one estimated given the odometry readings.

For the ground truth motion, the program should just visualize for each simulation step the robot poses as specified in the ground truth input file. For the dead reckoning solution, you start from the same initial pose as the ground truth solution but each new robot pose is an integration of the odometry reading, i.e., you simulate forward the robot's dynamics from the last estimated robot pose $q(t)$ given the u_i^{sensed} reported in the readings file at each simulation step to get $q(t + dt)$.

You should also visualize the ground truth of landmark as static points during the animation as filled blue dots.

You are also asked to visualize the landmark observations at each simulation step by using the robot pose estimates provided by dead reckoning. In particular, for each estimated robot pose q_i from dead reckoning and for each landmark j observation pair $[d_i^j, \alpha_i^j]$ draw a line segment originating from the robot coordinates according to q_i with length d_i^j towards direction α_i^j relative to the robot's local coordinate system given q_i. You can use the line segments shown in Fig. 5 as an example of what you can visualize. Only in your case because both the dead reckoning pose as well as the landmark observations are noisy, the line segments will not be necessarily ending on a ground truth landmark point.

You should also be visualizing the measurement of landmarks as static points during the animation, similar to how they are shown in Fig. 2 as filled red 'x'.

Your dead_reckoning.py script should be able to operate over any "readings_X_Y_Z.npy" files that follow the corresponding format. For your submission, save 5 example animation videos for 5 of the "readings_X_Y_Z" files you had generated before under a folder "dr" using the filename "dr_X_Y_Z.mp4". Choose at least one animation for each map and choose at least a couple of animations for each level of noise. Describe in your report your implementation process for these animations as well as your observations in terms of the impact of noise.

Visualization styles:

- The “ground truth” motion in blue
- The odometry motion in red
- The ground truth position landmark in blue (blue dot)
- The noisy observation position of landmark in red (red ’x’)

3 Implement Particle Filter for localization (40 pts)

A particle filter approach represents the robot’s state distribution at each point in time via a set of samples/particles. Refer to the corresponding lectures for a detailed explanation. Below is a summary of the approach given the notation of the assignment.

Each particle is an estimate of the robot’s state. For each discrete time step i, the particle filter takes as input the set of N particles from the previous time $P_{i-1} =< p^1_{i-1}, \ldots, p^N_{i-1} >$. Each particle is a robot pose estimate and a weight. The algorithm resamples a new set of particles according to these weights and sets the weights of the new particles to 1. See the implementation of the re-sampling procedure from the lectures and recitations.

It then uses the noisy odometry measurements u_i^{sensed} and the odometry model of Section 2.1.2 (i.e., either for a low or a high noise level) to propagate these particles forward and generate new pose estimate for each particle. In particular, given the pose of the j-th particle p_i^j after resampling, sample a motion for the robot for step i according to a Gaussian distribution centered at u_i^{sensed} that follows the given odometry model. Then define the new pose of the particle p_i^j by integrating the robot’s dynamics according to the sampled motion.

Each particle is then weighted based on the likelihood of all observed measurements. You can assume that the landmark observations are conditionally independent one from each other given an estimate of the robot’s state. Employ the observation model defined in Section 2.1.3 to define the likelihood of an individual landmark observation. These particles are used as the estimates for the state distribution at the new time step i. Use the weighted mean value of these particles as the best estimate at each step i.

3.1 Given known initial robot pose (35 pts)

You are going to use the "readings" files that you generated in Section 2.1 in order to generate robot pose estimates at each time step from the particle filter algorithm.

Create an executable Python file called particle_filter.py. We will run your code as:

```
python particle_filter.py --map maps/landmark\_X.npy
                          --sensing readings/readings\_X\_Y\_Z.npy
                          --num_particles N
                          --estimates estim1/estim1\_X\_Y\_Z\_N.npy
```

The readings file includes the initial (ground truth) robot pose, which will be used to initialize all of the particles at the same pose. The number of particles used by the algorithm is set equal to the value of the input parameter "num_particles". Consider at least one low value of particles (e.g., 200 or 500) and a high number of particles (e.g., 1000 or 2000) in your experimentation process. Make the choice of the number of particles depending on how computationally efficient you solution is, try to go on the higher side of the suggested number of particles if possible.

Your program has two outputs. It should generate an animation that visualizes the particles at each iteration of the algorithm, i.e., it should visualize at least the (x, y) location of the robot according to each particle as a point. If you can plot a short line segment per particle that indicates the robot’s orientation, this may be more informative but it is not required. You need to plot all particles on the map so you can see how particles move over consecutive iterations of the

algorithm. This visualization may help you with debugging your implementation. Given that you start from a known robot location, the hope is that the output of the particle filter will be close to the ground truth poses.

Notice that the particle filter algorithm does not make use of the “ground truth” robot poses to compute its estimates. It does need, however, the map of landmarks as input in order to compute the likelihood of landmark observations from different robot pose estimates.

The other output of the script is a file "estim1/estim1_X_Y_Z_N.npy" that stores at each iteration of the algorithm (i.e., for the 201 time steps that we are executing it) the mean particle estimate for the (x, y, θ) coordinates of the robot. The format of this file is similar to that of the ground truth pose files.

Your particle_filter.py script should be able to operate over any "readings_X_Y_Z.npy" files that follow the corresponding format. Make sure to execute the script for all combinations of landmark maps (x5), control sequences (x2), level of noise (x2) and number of particles (x2), i.e., you should generate 40 different output files for all these conditions that you will use to evaluate the performance of your algorithms.

In terms of animation videos for your submission, save 5 example animation videos for the same choices that you made during the generation of the dead reckoning solution (check page 1 for naming). Describe in your report your implementation process for these animations as well as your observations in terms of the impact of noise and the number of particles chosen.

Visualization styles:

- The odometry motion in red
- estimated robot’s pose in black
- for all particles, please plot them with a small dot.
- The ground truth position landmark in blue (blue dot)
- The noisy observation position of landmark in red (red ’x’)

3.2 Assume Unknown Initial Robot Pose (5 pts)

Repeat the process of Section 3.1 with the only change that the algorithm is now not allowed to use the robot’s initial pose stored in the input readings file. Instead, initialize the particle population from a uniform distribution.

Create an executable Python file called particle_filter_kidnapped.py. We will run your code as:

```
python particle_filter_kidnapped.py --map maps/landmark\_X.npy
                                    --sensing readings/readings\_X\_Y\_Z.npy
                                    --num_particles N
                                    --estimates estim2/estim2\_X\_Y\_Z\_N.npy
```

The hope is that while your particle population is initially uninformed regarding the robot location, it is able to quickly converge around the ground truth robot pose. You may need a higher number of particles for this version of the problem, especially at initial iterations of the algorithm. Experiment with what ends up working for you given the computational resources you have access to.

As in the previous section, you should execute the script for all combinations of landmark maps (x5), control sequences (x2), level of noise (x2) and number of particles (x2), i.e., you should generate 40 different output files for all these conditions that you will use to evaluate the performance of your algorithms.

But you are only asked to save 5 example animation outputs for the same choices that you made during the generation of the dead reckoning solution (check page 1 for naming).

4 Evaluation and Experiments (20 pts)

Create an executable Python file called evaluation.py. We will run your code as:

```
python evaluation.py --map maps/landmark\_X.npy
                     --execution gts/gt\_X\_Y.npy
                     --estimates estim1/estim1\_X\_Y\_Z\_N.npy
```

Your program should generate an animation of your robot moving around according both to the ground truth poses as well as the estimates of the particle filter algorithm. Check page 2 for naming of the output animation videos to store in your report.

Visualization styles for video:

- The ground truth motion in blue;
- The estimated robot's pose in black;
- The ground truth position landmark in blue (blue dot).

Replace the estim1_X_Y_Z_N.npy files with the estim2_X_Y_Z_N.npy files to evaluate solutions for the case of a "kidnapped" robot of Section 3.2, i.e., for an unknown initial robot pose.

Your program should also compute the error between the estimated robot pose and the ground truth robot pose at each iteration and generate a plot that visualizes this error as a function of time step. Please include error plots in your report and discuss the impact of the following problem or algorithm parameters to the accuracy and runtime of the approach relative to the dead reckoning solution:

- Level of noise in readings: Low vs. high noise.
- Number of particles

Try to be comprehensive in terms of your evaluation in your report. As in the previous sections, you should execute the script for all combinations of landmark maps (x5), control sequences (x2), level of noise (x2) and number of particles (x2) and use the output of this process to generate your plots in terms of accuracy and discuss impact on runtime as well.

Visualization styles for your plots: Use simple line charts (with two subplots) `https://matplotlib.org/stable/gallery/subplots_axes_and_figures/subplot.html#sphx-glr-gallery-subplots-axes-ar`
Subplot 1:
The X-axis should correspond to the time step.
The Y-axis should report the translational error relative to the ground truth pose (Euclidean distance);
Subplot 2:
The X-axis should correspond to the time step.
The Y-axis should report the rotational error relative to the ground truth pose (respect the circular topology);

5 Extra Credit Assignment - Kalman filter (25 pts)

Create an executable Python file called kalman_filter.py. We will run your code as:

```
python kalman_filter.py --map maps/landmark\_X.npy
                        --sensing readings/readings\_X\_Y\_Z.npy
                        --estimates estim1/kalman_estim1\_X\_Y\_Z\_N.npy
```

Focus on solving the version of Section 3.1, i.e., given a known initial robot pose. You should again store the Kalman filter estimation for all 20 experimental setups. There is no algorithmic parameter here to experiment over (i.e., such as the number of particles like in the particle filter case). Save at least 5 example animations as "kalman/kalman_X_Y_Z.mp4" for the same 5 setups

you have selected for the particle filter, where you show both the mean estimate for the x, y coordinates of the robot as well as the uncertainty of the filter using an ellipsoid in the x, y space.

Compare the results with particle filter in terms of accuracy and computation time. Similar to problem 4 generate the corresponding plots and discuss in your report.

Collusion, Plagiarism, etc.: Each team must prepare their solutions independently from others, i.e., without using common code, notes or worksheets with other students. You can discuss material about the class that relates to the assignment but you should not be provided the answers by other students and you must code your own solutions as well as prepare your own reports separately.

Furthermore, you must indicate at the end of your report any external sources you have used in the preparation of your solution. This includes both discussions with other students and online sources. Unless explicitly allowed by the assignment, do not plagiarize online sources and in general make sure you do not violate any of the academic standards of the course, the department or the university.

Online sources include the use of ChatGPT or other Large Language Models and Generative AI tools. While the use of such tools is allowed as supportive instruments for programming, the teams need to report the level of their use and the part of the code that was generated by them. Students have to be careful that blindly copying significant pieces of code from such tools - which falls under the definition of plagiarism - may result on submissions that do not accurately answer the assignment.

Failure to follow these rules may result in failure in the course.