UNIVERSITY OF COLORADO - BOULDER
Robotics Program

ROBO 5302 (CSCI 4302/5302) - Advanced Robotics

Homework #4 (Assigned: Tue 11/4, Due: Fri 11/21 11:59pm on *Canvas*)
State Estimation

Instructions

- Submission will take the form of a link to your GitHub branch that contains the work for this assignment. This branch needs to contain all of your code and a pdf with any written answers.
- Your writeup should be well organized and must contain the relevant figures that your code generated (such as a screenshot of your final state estimate and any graphs of error vs time generated by the visualization python script) in addition to you documenting your approach to the problems, issues encountered and how you solved them. The writeup should also contain your answers to the specific prompts posed in each subsection listed under 'Deliverables'.
- The provided README.md in each zipped folder contains instructions for how to install the package for each portion of this assignment. For each portion, there is a .py file in which you will need to complete the functions that run the given filter. The docstrings for the functions that need to be completed contain instructions for how to complete the code. You are advised against changing the core functionality of the code that is provided (except for debugging purposes).

Part 1: Setup

First, you will need to install matplotlib, numpy, and scipy,.

```
pip install numpy
pip install matplotlib
pip install scipy
```

or

```
conda install numpy
conda install matplotlib
conda install scipy
```

As mentioned above, each filter implementation requires a separate package installation. Navigate to the relevant package directory (ie the top-level folder named `assignmentX_filtername` that contains a setup.py file) and run

```
pip install -e .
```

You should see a message that says "Successfully installed filtering_exercises_package_name" where package name is the name of the respective package. See the README.md files for more instructions on completing each filter and the command to run the visualization of filter functionality.

Part 2: Bayes Filter

In this section, you will implement and analyze a **Bayes filter** for robot localization in a discrete 2-D grid world environment. The goal is to estimate the robot's (x, y) position on a known map using a probabilistic model of motion and sensing.

1. Environment

The robot operates in a known occupancy grid map represented as a 2-D array:

- Each cell corresponds to a discrete position (i, j).
- **Free cells** are marked with `0` (the robot can occupy these).
- **Obstacle cells** are marked with `1` (the robot cannot occupy these).

The environment provides:

- `env.grid`: the grid map
- `env.size`: the grid dimensions $(\text{rows}, \text{cols})$
- `env.n_beams`: the number of range sensor beams

2. Robot State

The robot's true state consists of:

$$x_t = (i_t, j_t, \theta_t)$$

where:

- (i_t, j_t) = robot's position (row, column)
- $\theta_t \in \{0, 1, 2, 3\}$ = orientation (right, up, left, down)

Because orientation is not directly observed, the Bayes filter maintains a belief distribution only over positions:

$$bel(x_t) = P((i, j) \mid z_{1:t}, u_{1:t})$$

That is, the probability that the robot is in cell (i, j) given all past sensor readings $z_{1:t}$ and actions $u_{1:t}$.

The belief is represented as a 2-D array of probabilities, one for each free grid cell. Initially, the robot has no information about its position, so the belief is uniform over all free spaces.

3. Motion Model (Prediction Step)

At each time step, the robot executes one of three possible actions:

Action	Meaning
0	Move forward
1	Turn right
2	Turn left

The motion model is probabilistic:

- With probability $1 - p_{noise}$, the robot successfully moves forward into the next cell (if it is free).
- With probability p_{noise}, the robot fails to move and remains in the same cell.
- Turning actions do not change position (only heading).

Because the filter does not explicitly track heading, it assumes a uniform distribution over headings when propagating motion:

$$P(x_t \mid u_t = forward, x_{t-1}) = \begin{cases} (1 - p_{noise}), & \text{if move is valid} \\ p_{noise}, & \text{if move fails} \end{cases}$$

$$P(x_t \mid u_t = turn, x_{t-1}) = \begin{cases} 1, & \text{if } x_t = x_{t-1} \\ 0, & \text{if move fails} \end{cases}$$

4. Measurement Model (Correction Step)

The robot has a simple range sensor that measures the distance to the nearest obstacle in each direction (front, right, back, left). The true readings depend on both position and orientation.

For a hypothetical position (i, j) and heading θ, the expected sensor readings $\hat{z}$ are computed by "casting rays" outward in the grid until they hit an obstacle or the map boundary.

The measurement model compares these expected readings $\hat{z}$ with the actual readings z received from the robot's sensors:

$$P(z_t \mid x_t) = \text{average over all headings} \prod_{k=1}^{n_{\text{beams}}} p(z_t^k \mid \hat{z}^k)$$

Each beam likelihood is modeled as:

$$p(z_t^k \mid \hat{z}^k) = \begin{cases} 1 - m_{noise}, & \text{if } |z_t^k - \hat{z}^k| < \text{tolerance} \\ m_{noise} \cdot \dfrac{1}{1 + |z_t^k - \hat{z}^k|}, & \text{otherwise} \end{cases}$$

This likelihood quantifies how consistent the current sensor readings are with being at cell (i, j).

5. Output and Interpretation

After each update, the filter maintains a full belief map over the grid. The most likely cell corresponds to:

$$\hat{x}_t = \arg\max_{(i,j)} bel(i, j)$$

which is the current best estimate of the robot's position.

Over time, as the robot moves and senses, the belief should converge to the true location, even starting from a uniform prior.

Your Task

You will complete the empty functions in the `bayes_filter.py` file and verify correct behavior using the simulation instructions detailed in the README file. See the docstring of a function for more detailed instructions on how to complete it. The empty functions are:

- predict
- update

Deliverables

- Complete the empty functions in the `bayes_filter.py` file following the instructions provided in the docstrings.
- Run the visualizer of the dynamics and estimate over time following the instructions in the README (this will use `test_bayes_filter_vis.py`). Provide a screenshot of the final estimate in the visualizer.

Part 3: Extended Kalman Filter for Range-Based Robot Localization

Overview

In this section, you will implement an **Extended Kalman Filter (EKF)** to estimate a robot's position and orientation in a 2D world using range measurements from a single fixed station. The robot follows a nonlinear unicycle motion model with position-dependent velocity scaling, and you will derive and implement both the motion and measurement Jacobians. To set up the package, navigate to the `assignment2_ekf` directory and follow the directions in the README.md file.

Environment Description

The environment is a two-dimensional world that defines how the robot moves and senses its surroundings. It provides nonlinear terrain effects, a fixed range station, and bounds on the robot's motion.

- The environment defines a map of size $(x_{\max}, y_{\max})$, typically (10×5) meters.
- A single **ranging station** is placed at a known position:

  $$\mathbf{s} = [x_s, y_s]^\top,$$

  which serves as the fixed point for range measurements.
- The terrain influences robot motion through a spatially varying **velocity scaling function**:

  $$v_{\text{eff}}(x, y) = v \cdot \text{velocity\_scaling}(x, y),$$

  where the scaling term depends on sinusoidal terrain parameters:

  $$\text{velocity\_scaling}(x, y) = 1 + 0.5 \sin(f_x x) \sin(f_y y),$$

  with f_x and f_y representing the terrain frequency in the x and y directions.
- The maximum forward velocity is defined by the environment parameter $v_{\max}$, and sensor noise for the range station is modeled by σ_{station}:

  $$z = \sqrt{(x - x_s)^2 + (y - y_s)^2} + \mathcal{N}(0, \sigma^2_{\text{station}}).$$

Robot State and Motion Model

The robot is modeled as a **unicycle** system with nonlinear dynamics. Its state vector is:

$$\mathbf{x} = \begin{bmatrix} x \\ y \\ \theta \end{bmatrix},$$

where:

- x, y are the Cartesian coordinates of the robot in the environment.
- θ is the heading angle (in radians), measured from the global x-axis.

The control input vector is:

$$\mathbf{u} = \begin{bmatrix} v \\ \omega \end{bmatrix},$$

where v is the commanded forward velocity and ω is the commanded angular velocity.

The nonlinear motion model is given by:

$$\begin{aligned} x_{t+1} &= x_t + v_{\text{eff}} \cos(\theta_t)\, \Delta t, \\ y_{t+1} &= y_t + v_{\text{eff}} \sin(\theta_t)\, \Delta t, \\ \theta_{t+1} &= \theta_t + \omega_{\text{eff}}\, \Delta t, \end{aligned}$$

where:

$$v_{\text{eff}} = v \cdot \text{velocity\_scaling}(x, y), \quad \omega_{\text{eff}} = \frac{\omega}{\text{speed\_factor}},$$

and the speed-dependent turning factor ensures stable motion:

$$\text{speed\_factor} = \text{clip}\left(\frac{|v_{\text{eff}}|}{v_{\max}}, 0.5, 1.5\right).$$

This model captures both nonlinear kinematics and environmental effects on motion, providing a realistic simulation setup for the Extended Kalman Filter.

Your Task

You will complete the following functions in the `extended_kalman_filter.py` file and verify correct behavior using the simulation instructions detailed in the README file. See the docstring of a given function for more detailed instructions on how to complete it.

1. _motion_model(self, state, forward_vel, angular_vel, dt)

Purpose: Predicts the next state $[x, y, \theta]$ based on control inputs and nonlinear unicycle kinematics.

What to do: Implement the nonlinear motion equations for the unicycle model:

$$\begin{aligned} x_{t+1} &= x_t + v_{\text{eff}} \cos(\theta_t)\, \Delta t, \\ y_{t+1} &= y_t + v_{\text{eff}} \sin(\theta_t)\, \Delta t, \\ \theta_{t+1} &= (\theta_t + \omega_{\text{eff}}\, \Delta t) \bmod 2\pi \end{aligned}$$

where

$$v_{\text{eff}} = v \cdot \text{velocity\_scaling}(x, y),$$

and ω_{eff} is adjusted using the speed-dependent turning factor.

2. _motion_jacobian(self, state, forward_vel, angular_vel, dt)

Purpose: Computes the **Jacobian matrix** $F = \frac{\partial f}{\partial x}$ of the motion model with respect to the state.

What to do: Derive and fill in all partial derivatives of the motion equations. The matrix should capture how small changes in x, y, θ affect the next state.

Use:

- Partial derivatives of the velocity scaling function $(dvs/dx, dvs/dy)$,
- Effective velocity v_{eff},
- Nonlinear dependence on heading via $\sin(\theta)$ and $\cos(\theta)$.

3. _measurement_jacobian(self, state)

Purpose: Computes the **Jacobian matrix** $H = \frac{\partial h}{\partial x}$ of the measurement model.

What to do: Linearize the measurement function:

$$z = h(x, y) = \sqrt{(x - x_s)^2 + (y - y_s)^2}$$

by deriving:

$$\frac{\partial z}{\partial x} = \frac{x - x_s}{r}, \quad \frac{\partial z}{\partial y} = \frac{y - y_s}{r}, \quad \frac{\partial z}{\partial \theta} = 0,$$

and handle the near-zero range case safely.

4. predict(self, forward_vel, angular_vel, dt)

Purpose: Implements the **EKF prediction step** using the motion model and its Jacobian.

What to do:

1. Call `_motion_model()` to predict the next state.
2. Compute F using `_motion_jacobian()`.
3. Update the covariance:
$$P = FPF^T + Q$$
4. Normalize the heading and store history.

5. update(self, measurements)

Purpose: Implements the **EKF update step** using a range measurement from the fixed station.

What to do:

1. Compute predicted measurement $\hat{z}$ with `_measurement_model()`.
2. Compute H using `_measurement_jacobian()`.
3. Compute Kalman gain:
$$K = PH^T(HPH^T + R)^{-1}$$

4. Update the state and covariance:

$$x = x + K(y - \hat{y}), \quad P = (I - KH)P$$

5. Clip large updates, normalize heading, and store results.

Deliverables

- Complete all of the above-listed functions in the `extended_kalman_filter.py` file following the instructions provided in the docstrings.
- Provide the derivations for the Jacobians for the F and H matrices.
- Run the visualizer of the dynamics and estimate over time following the instructions in the README.md (this will use `test_particle_filter_vis.py`). Provide a screenshot of the final estimate in the visualizer.
- Note that the particle filter solution may start with an accurate estimate and lose accuracy over time. This is okay with a single-range measurement, and normally multiple measurements are integrated to get an accurate EKF estimate. The estimate should have position errors of around 0.5m, though in some instances of the simulation they may become slightly larger.
- For more information on installing and running the EKF, see the README.md file in the `assignment2_ekf` directory.

Extra Credit

- Add an extra ranging station to the simulation at a new location of your choice. This will mean that every measurement vector includes two range measurements. Note that this will change the size of your H and P matrices and that your measurement function and measurement Jacobian function will have to be changed accordingly. You should see that the EKF estimate of the robot's position is now very accurate when you get this running.
- Submit screenshots of your single ranging station EKF behavior (described in the previous section) as well as screenshots of the final estimate and graphs of errors for the double ranging station EKF simulation. You do not need to submit working code for both examples, just for the 2-station simulation.

Part 4: Particle Filter (GRAD QUESTION)

Problem Setup and Environment

You are localizing a robot in a 2D environment (`MultiModalWorld`) that contains two visually similar hallways, making the belief distribution **multimodal** — at the start, the robot could plausibly be in either hallway.

Robot State

$$x = [x,\, y,\, \theta]^T$$

- x, y: position
- θ: heading angle

Control Inputs

$$u = [v,\, \omega]^T$$

- v: commanded forward velocity
- ω: commanded angular velocity

Forward Model

At each time step, we apply the following update for every particle:

$$\begin{aligned} v' &= v + \varepsilon_v, & \varepsilon_v &\sim \mathcal{N}(0, \alpha_1^2 v^2), \\ \omega' &= \omega + \varepsilon_\omega, & \varepsilon_\omega &\sim \mathcal{N}(0, \alpha_2^2 \omega^2), \end{aligned}$$

where α_1 and α_2 (variance values) control the amount of noise added to the forward and angular velocities, respectively. This means that faster motions produce more uncertainty. *Hint: when randomly sampling noise for each particle, pay close attention to what inputs your random normal sampling function takes in (variance vs standard deviation).*

Each particle is then propagated through the (noisy) motion model:

$$\begin{aligned} x' &= x + v' \cos(\theta)\, \Delta t, \\ y' &= y + v' \sin(\theta)\, \Delta t, \\ \theta' &= \theta + \omega'\, \Delta t. \end{aligned}$$

Measurements

At each timestep, the robot receives beam distance measurements from a simulated range sensor (similar to a low-resolution lidar). Each beam measures the distance to the nearest wall in that direction. The true measurement model is nonlinear and subject to Gaussian noise.

If a particle moves into a wall or invalid region, it is reinitialized randomly in free space.

Update (Measurement Likelihood)

Each particle predicts what the robot *should have seen* from its pose by ray-casting in the environment:

$$\hat{z}^{[m]} = h(x^{[m]})$$

and compares this to the actual measurements z:

$$w^{[m]} \propto \prod_{i=1}^{N_{\text{beams}}} \exp\left[-\frac{1}{2}\left(\frac{z_i - \hat{z}_i^{[m]}}{\sigma_{\text{range}}}\right)^2\right]$$

Particles that produce measurements similar to the real sensor readings receive higher weights. Weights are normalized so that:

$$\sum_m w^{[m]} = 1$$

Resampling

Particles with higher weights are duplicated; those with low weights are dropped.

You will use a **multinomial resampling** strategy:

- Draw N new samples from the current set of particles with probability proportional to their weights.
- Add a small amount of random noise (`self.resampling_noise`) to maintain diversity.
- Reset all weights to uniform.

State Estimate

You still ultimately require a point estimate of the robot's state from all of your weighted particles. One way to do this is to take the most likely particle (particle with largest weight) as the point estimate. In the assignment you will take a weighted average over all particles.

After the update and resampling steps, the particle filter maintains a set of particles $\{\mathbf{x}^{[m]}, w^{[m]}\}_{m=1}^{N}$ representing the belief distribution over robot states. To obtain a single state estimate, we compute the weighted mean of the particles.

Each particle state is

$$\mathbf{x}^{[m]} = \begin{bmatrix} x^{[m]} \\ y^{[m]} \\ \theta^{[m]} \end{bmatrix}, \qquad \sum_m w^{[m]} = 1.$$

Weighted mean position. The estimated position is given by the weighted average of all particle positions:

$$\bar{x} = \sum_m w^{[m]} x^{[m]}, \qquad \bar{y} = \sum_m w^{[m]} y^{[m]}.$$

Weighted mean heading. Since the heading θ is an angle on the unit circle, it must be averaged using circular statistics. We first compute the weighted mean of the unit vectors $(\cos\theta^{[m]}, \sin\theta^{[m]})$ and then recover the mean angle using the two-argument arctangent:

$$\bar{\theta} = \operatorname{atan2}\left(\sum_m w^{[m]} \sin(\theta^{[m]}), \sum_m w^{[m]} \cos(\theta^{[m]})\right).$$

Final state estimate. The estimated robot state is therefore

$$\hat{\mathbf{x}} = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{\theta} \end{bmatrix}.$$

This estimate corresponds to the expected value of the belief distribution represented by the particle filter and provides a concise summary of the current localization estimate.

Your Task

You will complete the empty functions in the `particle_filter.py` file and verify correct behavior using the simulation instructions detailed in the README file. See the docstring of a function for more detailed instructions on how to complete it. The empty functions are:

- predict
- update
- resample
- estimate_state

Deliverables

- Complete all of the empty functions in the `particle_filter.py` file following the instructions provided in the docstrings.
- Run the visualizer of the dynamics and estimate over time following the instructions in the README (this will use `test_particle_filter_vis.py`). Provide a screenshot of the final estimate in the visualizer