Principles of Robot Autonomy I
Problem Set 4
Due Friday, December 2 at 11:59pm

Starter code for this problem set has been made available online through Google Drive; to get started download the code from https://drive.google.com/file/d/1vZ6gDqHZ5jOyJpI8qWwpz1Ri_6sAQ1SS/view?usp=share_link.

For your final submission, you will submit the python files that contain your work for the code items (denoted by the 🖥 symbol) and a pdf containing your write up for questions with the ✎ symbol.

IMPORTANT NOTE ON GROUP WORK: Problem 1 and 2 in this problem set involve interaction with ROS and may thus be completed in your final project groups (with identical writeups/code). Group submissions have been opened on Gradescope.

This problem set has no extra problem required for students taking AA 274A/CS 237A/EE 260A for 4 credits. Instead, Problem 3 will be offered as extra credit to all students. Problem 3 must be completed individually and submitted separately on Gradescope.

Problem 1: EKF Localization

In this problem we will use the linear feature extraction methodology developed in Problem Set 3 as the basis for a robot localization Extended Kalman Filter (EKF). Essentially, given a known map of linear features, we can correct the output of open-loop state propagation by measuring the difference between linear features perceived by the robot and the map features it expects to see. See pages 331–342 in SNS for a more detailed exposition of line-based EKF localization on which this problem is modeled.

Recall the differential drive model through which we represent the dynamics of our simulated Turtlebot:

$$\begin{aligned} \dot{x}(t) &= V(t)\cos(\theta(t)) \\ \dot{y}(t) &= V(t)\sin(\theta(t)) \\ \dot{\theta}(t) &= \omega(t) \end{aligned} \tag{1}$$

The continuous state variable is $\mathbf{x}(t) = [x(t), y(t), \theta(t)]^T$ and the instantaneous control is $\mathbf{u}(t) = [V(t), \omega(t)]^T$.

(i) ✎ As discussed in class, we can derive a discrete time model from these continuous dynamics by assuming a zero-order hold on the control inputs (i.e., hold $\mathbf{u}(t)$ constant over a time interval of length dt). That is, for suitable notions of $\mathbf{x}_t$ and $\mathbf{u}_t$ we may write $\mathbf{x}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t)$ and for small perturbations $(\tilde{\mathbf{x}}_{t-1}, \tilde{\mathbf{u}}_t)$ close to $(\mathbf{x}_{t-1}, \mathbf{u}_t)$ we may Taylor expand:

$$\tilde{\mathbf{x}}_t = g(\tilde{\mathbf{x}}_{t-1}, \tilde{\mathbf{u}}_t) \approx g(\mathbf{x}_{t-1}, \mathbf{u}_t) + G_x(\mathbf{x}_{t-1}, \mathbf{u}_t) \cdot (\tilde{\mathbf{x}}_{t-1} - \mathbf{x}_{t-1}) + G_u(\mathbf{x}_{t-1}, \mathbf{u}_t) \cdot (\tilde{\mathbf{u}}_t - \mathbf{u}_t)$$

where G_x and G_u are the Jacobians of g with respect to $\mathbf{x}$ and $\mathbf{u}$ respectively.[1]

[1] This is just the multivariate analogy of approximations like $f(\tilde{x}) \approx f(x) + (\tilde{x} - x)f'(x)$.

- Derive the transition model, $g(\mathbf{x}_{t-1}, \mathbf{u}_t)$, and the Jacobians, G_x and G_u. **Show your work for full credit.**
- In the case where $|\omega|$ is too small, $g(\mathbf{x}_{t-1}, \mathbf{u}_t)$, G_x, and G_u may suffer from numerical instability. Derive the limit of $g(\mathbf{x}_{t-1}, \mathbf{u}_t)$, G_x, and G_u as $\omega \to 0$

HINT: To compute the new state g, you will need to integrate the dynamics of x, y, and θ. Since θ is changing with time, try integrating x and y with respect to $d\theta$ instead of dt by introducing ω to perform a change of variables. When $|\omega|$ is small, assume that θ stays approximately constant ONLY for calculating the next x and y. In other words, $\theta_t \neq \theta_{t-1}$. The Jacobian with respect to ω is not 0.

(ii) Implement the computation of g, G_x, and G_u in the `compute_dynamics()` function in `turtlebot_model.py`. When $|\omega| <$ `EPSILON_OMEGA`, you will want to use the limits of $g(\mathbf{x}_{t-1}, \mathbf{u}_t)$, G_x, and G_u as $\omega \to 0$ derived in (i). Note that your implementation must accommodate arbitrary control durations dt.

Then call `compute_dynamics()` in the `transition_model()` method of the `EKFLocalization` class in `ekf.py`.

Run `validate_localization_transition_model()` from `validate_ekf.py` to check your work.

(iii) Let the belief state at time $t-1$ be distributed as $\mathcal{N}(\mathbf{x}_{t-1}, \Sigma_{t-1})$. We will model the uncertainty in our Turtlebot's dynamics propagation by additive continuous white noise $\nu \sim \mathcal{N}(\mathbf{0}, R)$ applied to the control input. Then over a time step dt the EKF prediction step is:

$$\bar{\mathbf{x}}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t)$$
$$\bar{\Sigma}_t = G_x \cdot \Sigma_{t-1} \cdot G_x^T + dt \cdot G_u \cdot R \cdot G_u^T.$$

Implement the dynamics transition update (i.e., prediction step) in the `transition_update()` method of the `Ekf` class in `ekf.py`.

Run `validate_ekf_transition_update()` from `validate_ekf.py` to check your work.

The discrete-time EKF developed in class assumes that all time steps are the same duration and every prediction step is followed immediately by a measurement correction. This yields arguably the cleanest presentation of the EKF, but is a bit at odds with how typical robotic systems act in practice. Measurement corrections (e.g. from scanner data, GPS data) often occur at rates $\sim$ 10Hz interspersed with controls at up to $\sim$ 1kHz. Moreover, these measurement updates often take time to process. Often, by the time the measurements are ready to be applied for filtering, they may no longer correspond to the most recent prediction step. Thus, instead of a nice, lockstep version of the EKF, we may equivalently think of the EKF as an object that tries to maintain a high-quality Gaussian belief state based on whatever information it's seen up to some time. That is, transition updates and measurement updates may stream in arbitrarily, but as long as they're processed in order and appropriately according to our mathematical dynamics/measurement/noise models, the lack of synchronization won't be a problem. Thus in the following parts let $\bar{\mathbf{x}}_t$ and $\bar{P}_t$ denote the most up-to-date belief mean and covariance at the time of measurement.

(iv) Our turtlebot is equipped with a depth sensor[2] that allows it to detect lines in the coordinate frame of its camera. In this problem we compare these exteroceptive location cues against known lines in a map M, a $2 \times J$ matrix with columns $\mathbf{m}^j = [\alpha^j, r^j]^T$ corresponding line parameters in the world frame (in the code this map is stored as `self.map_lines` for a `EkfLocalization` object). To compare the predicted and observed measurements we must convert the world-frame parameters for each map line into the camera frame.

Implement the coordinate change for a map entry between the world frame and camera frame in the `transform_line_to_scanner_frame()` function in `turtlebot_model.py`; see Figure 1 for a description of the relationship between these two coordinate frames. That is, you should compute the mean camera

[2] Which we've made quite noisy in the simulation portion of this problem for "dramatic effect."

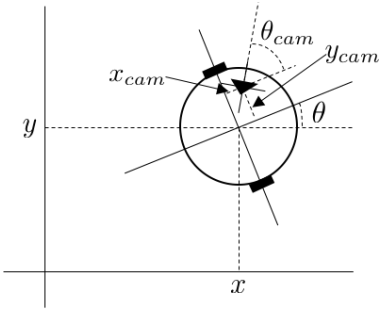

Figure 1: The robot's state defines the offset/yaw of the robot's base frame with respect to the world frame. The member variable `self.tf_base_to_camera` will contain the (constant) offset/yaw of the robot's camera frame with respect to its base frame.

frame parameters $\mathbf{h}_t$ (which will depend on $\bar{\mathbf{x}}_t$) for a single map entry $\mathbf{m}$, as well as its Jacobian H_t with respect to the belief state mean.

Next, use this coordinate change function to complete the `compute_predicted_measurements()` method of the `EkfLocalization` class in `ekf.py`.

Run `validate_localization_compute_predicted_measurements()` from `validate_ekf.py` to check your work.

Annotate Figure 1 with a line and its parameters (α, r) in the world and camera frame. Also annotate any intermediate reference frames you used in your implementation. This can be hand-drawn on top of Figure 1.

(v) In order to apply the Kalman filter update correctly, the observed lines must be associated with the map entries most likely to have produced them. To this end, we use the Mahalanobis distance between a predicted measurement $\mathbf{h}_t^j$ (i.e., a map entry $\mathbf{m}_t^j$ in the camera frame) and observation $\mathbf{z}_t^i$ (i.e., a line extracted from the scanner angle/depth data). With the *innovation*

$$\mathbf{v}_t^{ij} = \mathbf{z}_t^i - \mathbf{h}_t^j$$

as a measure of the difference between a predicted and observed measurement, and with the *innovation covariance*

$$S_t^{ij} = H_t^j \cdot \bar{\Sigma}_t \cdot {H_t^j}^T + Q_t^i$$

(where Q_t^i is the covariance[3] of the observation $\mathbf{z}_t^i$), we may calculate the Mahalanobis distance as

$$d_t^{ij} = {\mathbf{v}_t^{ij}}^T \cdot (S_t^{ij})^{-1} \cdot \mathbf{v}_t^{ij}.$$

For each observed scanner line we want to associate the most likely map entry (i.e., the entry with least Mahalanobis distance), but we want to make sure we don't fall prey to any corrupting measurements that do not correspond well to any entries in the map (i.e., unmapped changes in the environment). Thus we introduce a *validation gate* g and consider only associations that fall into this gate $d_t^{ij} < g^2$. The validation gate g may be found as `self.g` for a `EkfLocalization` object.[4]

Each observation $\mathbf{z}_t^i$ will be associated with at most one map entry $\mathbf{h}_t^j$; for each association $(i, j(i))$ the "measurement" that we actually use in the EKF update equations (see part vii below) will be the innovation $\mathbf{v}^{ij}$, with measurement covariance R^i and Jacobian with respect to the belief mean H_j^t.

[3] We didn't make you compute this covariance in last week's assignment, but rest assured that a Gaussian approximation of the uncertainty in (α, r) that arises from Gaussian noise on the scanner measurements $(\theta, \rho)_k$ may be derived using linearization and Jacobians (the common theme of this whole EKF business).

[4] Note that this g is distinct from the discrete transition function $g(x, u)$; the validate gate is a multivariate Gaussian analog of z-score (and is a scalar)!

Write an algorithm, in pseudocode, of how you will implement `compute_innovations()` method of the `EkfLocalization` class in `ekf.py`.

Implement the measurement association process in `compute_innovations()`.

Run `validate_localization_compute_innovations()` from `validate_ekf.py` to check your work.

(vi) We are now in a position to implement the EKF measurement correction. We may stack the innovation vectors for all line associations into one big measurement

$$\mathbf{z}_t = \begin{bmatrix} \mathbf{v}_t^1 \\ \vdots \\ \mathbf{v}_t^K \end{bmatrix}$$

with covariance

$$Q_t = \begin{bmatrix} Q_t^1 & & 0 \\ & \ddots & \\ 0 & & Q_t^K \end{bmatrix}$$

and Jacobian with respect to $\mathbf{x}$

$$H_t = \begin{bmatrix} H_t^1 \\ \vdots \\ H_t^K \end{bmatrix}$$

(note: the indexing on Q's and H's here comes from `compute_innovations()` and should correspond to the ordering of the $\mathbf{v}$'s).

Implement the assembly of this joint measurement in the `measurement_model()` method of the `EkfLocalization` class in `ekf.py`.

(vii) Finally, we can implement the standard EKF measurement correction:

$$\begin{aligned} S_t &= H_t \cdot \bar{\Sigma}_t \cdot H_t^T + Q_t \\ K_t &= \bar{\Sigma}_t \cdot H_t^T \cdot S_t^{-1} \\ x_t &= \bar{x}_t + K_t \cdot \mathbf{z}_t \\ \Sigma_t &= \bar{\Sigma}_t - K_t \cdot S_t \cdot K_t^T \end{aligned}$$

Implement the measurement update (i.e., correction step) in the `measurement_update()` method of the `Ekf` class in `ekf.py`.

Run `validate_ekf_localization()` from `validate_ekf.py` to check your work.

(viii) Time to test your localization EKF on a real (simulated) robot! On Genbu, update (`git pull`) your `asl_turtlebot` package and copy the contents of the `HW4` folder within the `AA274A_HW4` repo (i.e., the code you've been editing) into `asl_turtlebot/scripts/HW4`. Now run the `hw4_maze.launch` launch file to load the simulation environment (gazebo) and state estimate visualization (rviz). You can visualize the environment from the TurtleBot's perspective through the camera in RViz. Click the "Add → By Topic" button on the bottom left and investigate more!

Getting the right ROS plumbing together for feeding control/measurement information into your EKF (including keeping track of timestamps) is a bit tricky[5] so we've provided a script `localization.py` for you (where would this file live?). Note that `hw4_maze.launch` starts up this localization node automatically so don't worry about manually `rosrun`-ing it.

[5] If you have a particular interest in learning the ins and outs of ROS we strongly encourage you to write this script yourself!

Start up a keyboard teleoperation controller for your simulated TurtleBot. You can use the one from the `turtlebot3_teleop` package (`roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch`), or the rosnode in the `asl_turtlebot` package (`rosrun asl_turtlebot keyboard_teleop.py`).[6]

Edit `localization.py` to inject some extra uncertainty into the system (e.g., perturb the initial position/heading, add Gaussian noise to the control/scanner measurements, routinely drop half your scan points, etc. — sometimes simulations are a little bit too perfect to provide really compelling results!). Depending on what uncertainty you add, it may also make sense to change some of the variance/covariance parameters in `maze_sim_parameters.py` as well.

Submit a list of uncertainties you injected and any differences you noticed in performance.

(ix)  Drive your TurtleBot around the maze, and observe how the open loop and EKF state estimate changes. By driving it around, explain what type of motions cause those estimates to diverge from each other. Take three screenshots of RViz (1) the initial state, (2) when the TurtleBot has moved far from the initial state, and (3) when the state estimates diverge, and include it in your write up.

[6] You may note that if you stop with only one wall in view, your EKF estimate may have drifted parallel to the direction of that wall. Turn about yourself with `j` or `l` to see that drift corrected!

Problem 2: EKF SLAM

Note: This problem builds upon Problem 1, so you should finish that problem before moving on to this one.

Localizing ourselves with respect to an a priori known map may make sense in structured settings (e.g., a factory floor which a robot will repeatedly traverse), but in general robots must autonomously build their own maps from scratch (accounting for uncertain localization) while exploring an environment. In this problem we'll explore a middle ground between these two extremes — where the robot has a fuzzy knowledge of the map that it must correct during its operation.

(i) EKF Simultaneous Localization and Mapping (SLAM) follows the same general structure of EKF localization; the main difference is that the EKF belief state is augmented with a probabilistic representation of the map:

$$\mathbf{x}(t) = [x(t), y(t), \theta(t), \alpha^1, r^1, \ldots, \alpha^J, r^J]^T.$$

The map features are assumed to be static in the world frame so that their dynamics are

$$\dot{\alpha}^j = 0 \qquad \forall j$$
$$\dot{r}^j = 0 \qquad \forall j.$$

Similar to Problem 1 (and likely reusing some code) we may consider the state transition function $\mathbf{x}_t = g(\mathbf{x}_{t-1}, \mathbf{u}_t)$ and its derivatives $G_x = \frac{\partial g(\mathbf{x}_{t-1}, \mathbf{u}_t)}{\partial \mathbf{x}}$ and $G_u = \frac{\partial g(\mathbf{x}_{t-1}, \mathbf{u}_t)}{\partial \mathbf{u}}$.

Implement the computation of g, G_x, and G_u in the `transition_model()` method of the `EkfSlam` class in `ekf.py`.

What is the dimension of your state vector? Leave it in terms of a variable number of map elements.

(ii) The main work of EKF SLAM is done in the measurement update. To do EKF SLAM "right", this measurement update should even include a procedure for expanding the state with new line features as they're encountered (and confidently identified as new, as opposed to a mis-extracted or noisy measurement of an existing line), but as mentioned above we're going to "cheat" and assume we have prior knowledge of the number of line features and a noisy idea of the parameters for each one.

First we note that without a fixed world frame only the relative position and heading of the robot with respect to the observable landmarks can be estimated. That is, for any fixed scenario there are degrees of freedom in $\mathbf{x}$ corresponding to translating or rotating both the robot and line features by the same amount simultaneously. To eliminate these degrees of freedom we will fix as certain the parameters of the first two landmarks in the state vector. You may imagine these as the measured parameters for the first two non-parallel line features the robot sees together in the same scanner measurement, and saying the robot state (x, y, θ) at that moment is $(0, 0, 0)$).

Reimplement `measurement_model()`, `compute_innovations()`, and `compute_predicted_measurements()` for the `EkfSlam` class based on your implementation for `EkfLocalization` in `ekf.py`. Not much should actually be different; the most notable change should be seen in the construction of $H_t^j = \frac{\partial \mathbf{h}_t^j(\mathbf{x}_t)}{\partial \mathbf{x}}$, minding the size increase of $\mathbf{x}$.

Run `validate_ekf_slam()` from `validate_ekf.py` to check your work.

(iii) Time to test! Make sure that the `ekf.py`, `ExtractLines.py`, `maze_sim_parameters.py`, and `turtlebot_model.py` in the `scripts/HW4` directory of `asl_turtlebot` are updated with your changes from parts (1) and (2). Run the `hw4_arena.launch` launch file to load the simulation environment (gazebo) and state, including map, estimate visualization (rviz). This launch file also runs an EKF SLAM node implemented in `ekf_slam.py`. Finally, start up a teleoperation node to drive the robot around.

Drive your TurtleBot around the arena[7], and observe how your map estimate changes. In particular, you should notice that over time, your map estimate should converge to the to the true estimate. By driving your TurtleBot around, explain what you can do to make your map estimate to converge to the right one. To show this, take screenshots of RViz at (1) the initial state, (2) when the TurtleBot has moved away from the initial state and the map estimate has changed, and (3) when the map estimates have converged and include them in your write up.

Further, investigate what type of motions may cause your EKF and ground truth estimates to diverge. Give some examples of commands you tried and the resulting behavior of the EKF.

[7] We've restricted the field of view of the simulated LIDAR scan so that the robot can't see the entire arena at once.

Extra Credit Problem: Monte Carlo Localization

In this problem, we will implement Monte Carlo Localization (MCL), a popular localization algorithm that uses particle filters instead of Kalman filters to localize the robot with respect to a known map. While Kalman filters are limited to unimodal distributions of the belief state (represented with a Gaussian), particle filters are non-parametric and can approximate any distribution with enough particles. This property allows MCL to perform global localization, while standard EKF Localization requires that the initial pose is known.

Most of the code should look very similar to Problem 1, except the computations need to be done for each particle. For extra extra credit, try to avoid for loops in your implementation wherever possible by using Numpy vector operations instead. Python for loops carry significant computational overhead, while Numpy operations use *highly* optimized C/Fortran code. Our solution contains no Python for loops, and is ~ 100 times faster as a result. This means we can comfortably handle 1000 particles in real-time, while a naive implementation might only handle 10. This gives you a sense of how important engineering quality is for localization and mapping algorithms.

Any correct implementation of the following problems will get extra credit. However, we will give even more extra credit for each nested for/while loop that you can eliminate (please detail such efforts in your writeup.).

(i)  In MCL, we represent the belief state with a set of M particles, each particle represented by a state estimate and its weight $(x_t^{(m)}, w_t^{(m)})$.

Implement the transition update step of MCL by completing the `transition_model()` method in the `MonteCarloLocalization` class and the `transition_update()` method in the `ParticleFilter` class in `particle_filter.py`. The transition update needs to be performed for each of the M particles.

Run `validate_transition_model()` from `validate_particle_filter.py` to check your work.

(ii)  Now implement the measurement update step by completing the `measurement_update()`, `measurement_model()`, `compute_innovations()`, and `compute_predicted_measurements()` methods for the `MonteCarloLocalization` class in `particle_filter.py`. Although particle filtering can perform measurement updates with raw observations (point clouds), we choose to use extracted line features to provide a direct comparison to EKF Localization. Unlike EKF Localization, there is no need to filter out corrupting measurements using the validation gate g, since particles with low weights will be naturally filtered out in the resampling step. The Mahalanobis distance can be computed as

$$d_t^{ij} = \mathbf{v}_t^{ij^T} \cdot (Q_t^i)^{-1} \cdot \mathbf{v}_t^{ij}.$$

Run `validate_predicted_measurements()` and `validate_compute_innovations()` from `validate_particle_filter.py` to check your work.

(iii) After updating the particle weights in the measurement update step, the particles should be resampled with importance sampling. One problem with resampling is that it decreases the diversity of the particles, since some particles will most likely be duplicated and others lost. To avoid this, we can perform the following low variance sampling algorithm.

function RESAMPLE($(x^{(0)}, w^{(0)}), \ldots, (x^{(M-1)}, w^{(M-1)})$)
  $r \leftarrow$ UNIFORMSAMPLE$(0, \frac{1}{M})$
  $i \leftarrow 0$
  $c \leftarrow w^{(0)}$
  $X \leftarrow \varnothing$
  for $m \leftarrow 0 \ldots M-1$ **do**
    $u \leftarrow \left(\sum_j w^{(j)}\right)\left(r + \frac{m}{M}\right)$
    while $c < u$ **do**
      $i \leftarrow i + 1$
      $c \leftarrow c + w^{(i)}$
    end while
    $X \leftarrow X \cup (x^{(i)}, \frac{1}{M})$
  end for
  return X
end function

This algorithm works by selecting a single random number at the beginning, and then cycling through the weights in increments of $\frac{1}{M}$. If the distribution of weights is uniform, then this resampling method will return the same set of weights. Even if the distribution isn't uniform, the probability that a particle is sampled is still proportional to its weight!

Implement this algorithm in the `resample()` method for the `ParticleFilter` class in `particle_filter.py`.

Run `validate_resample()` and `validate_mc_localization()` from `validate_particle_filter.py` to check your work.

(iv) Time to test! First update `particle_filter.py` in the `asl_turtlebot/scripts/HW4` folder.

Run the `hw4_maze.launch` launch file to load the simulation environment (gazebo) and state, including map, estimate visualization (rviz). To enable the usage of your particle filter set the arguments `mc` and `num_particles`, i.e., `roslaunch asl_turtlebot hw4_maze.launch mc:=true num_particles:=100`. You can visualize the particles by adding the `particle_filter` topic in RViz. You may have to zoom in to see. Finally, start up a teleoperation node to drive the robot around.

Depending on how fast your implementation is, you may have to lower the number particles to get reasonable performance. See if you can handle up to 1000 particles.

Drive your TurtleBot around the maze, and observe how the open loop and MCL state estimate changes. Take three screenshots of RViz (with the particle view enabled) with (1) the initial state, (2) when the TurtleBot has moved far from the initial state, and (3) when the state estimates diverge, and include it in your write up. Explain what type of motions may cause your MCL and ground truth estimates to diverge. Does this change with the number of particles you use?