AA 274: Principles of Robotic Autonomy
Problem Set 3: Filtering, Localization, and Mapping
Due Wednesday, March 6

Starter code for this problem set has been made available online through github; to get started download the code by running `git clone https://github.com/StanfordASL/AA274_HW3.git` in a terminal window.

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

Before you start, you will have to `git pull` from the `asl_turtlebot` repo. To make sure you do not have any merge conflicts, make sure that the following files have not been modified. You can type `git status` to check. If you have modified them, type `git checkout <filename>`, and then `git pull`.

- `launch/turtlebot3_gmapping.launch`
- `launch/turtlebot3_gmapping_sim.launch`

Problems 0 and 4 to come.

Problem 1

In this problem we will use the linear feature extraction methodology developed in Problem Set 2 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 input (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 $f(\tilde{x}) \approx f(x) + (\tilde{x} - x)f'(x)$.

Implement the computation of g, G_x, and G_u in the `transition_model` method of the `Localization_EKF` class in `ekf.py`. Note that your implementation must accommodate arbitrary control durations dt.

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

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

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

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

Run `validate_localization_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., scanner data, GPS data) often occur at rates $\sim$ 10Hz interspersed with controls streaming in at up to thousands of updates per second (e.g., modern MEMS inertial measurement units). Moreover, these measurement updates often take time to process so that by the time they're ready to be applied for filtering they may no longer be relevant to the most recent prediction step. Thus instead of the nice, lockstep mathematical 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.

(iii) 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 `Localization_EKF` object). To compare the predicted and observed measurements we must convert the world-frame parameters for each map line into the camera frame.

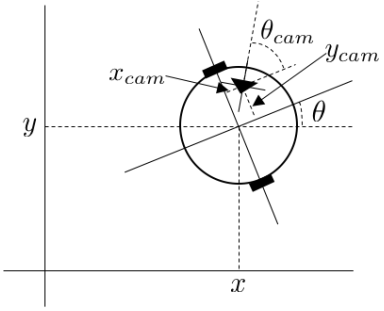

Figure 1: The robot's state defines the offset/yaw of the robot's base frame with respect to the world frame. The member variable `self.tf_base_to_camera` will contain the (constant) offset/yaw of the robot's camera frame with respect to its base frame.

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

Implement the coordinate change for a map entry between the world frame and camera frame in the `map_line_to_predicted_measurement` method of the `Localization_EKF` class in `ekf.py`; see Figure 1 for a description of the relationship between these two coordinate frames. That is, you should compute the mean camera 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.

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

(iv) 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*

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

(where R_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 `Localization_EKF` 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 5 below) will be the innovation $\mathbf{v}^{ij}$, with measurement covariance R^i and Jacobian with respect to the belief mean H_j^t.

Implement the measurement association process in the `associate_measurements` method of the `Localization_EKF` class in `ekf.py`.

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

(v) 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

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

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

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

[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)!

(note: the indexing on R's and H's here comes from `associate_measurements` 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 `Localization_EKF` class in `ekf.py`.

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

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

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

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

(vii) Time to test your localization EKF on a real (simulated) robot! Update (`git pull`) your `asl_turtlebot` package and run the `turtlebot3_maze` launch file to load the simulation environment (gazebo) and state estimate visualization (rviz). The default is `gui:=false`. You can still visualize the environment through the camera in RViz. Click the "Add" 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?). Copy `ekf.py`, `ExtractLines.py`, and `maze_sim_parameters.py` into the `scripts` folder and run the `localization` node.

Run the `teleop` launch file. You can use the one from homework 2, or the rosnode in the `asl_turtlebot` package. [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.

(viii) 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.

Problem 2

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.

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

[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!

(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 `SLAM_EKF` class in `ekf.py`. Note that your implementation must accommodate arbitrary control durations dt.

(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` for the `SLAM_EKF` class based on your implementation for `Localization_EKF` 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_SLAM_EKF()` from `validate_ekf.py` to check your work.

(iii) Time to test! Run the `turtlebot3_arena.launch` launch file to load the simulation environment (gazebo) and state, including map, estimate visualization (rviz). The default is `gui:=false`. You can still visualize the environment through the camera in RViz. Click the "Add" button on the bottom left and investigate more!

Make sure that the `ekf.py`, `ExtractLines.py`, and `maze_sim_parameters.py` are in the `scripts` directory of `asl_turtlebot` are updated with your changes from parts (1) and (2). Run the `map_fixing` node to start up the EKF SLAM.

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.

[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.

Problem 3

As you should have noticed, so far our Turtlebot software stack has been relying on an external localization solution in order to operate. Specifically, we have been using Gazebo to tell us exactly where the robot is in the world (through the `/gazebo/model_states` topic). Unless we were providing you with a motion capture system, this would not work on the real Turtlebot. However, now that we know how to localize our robot using inertial and range measurement, we can address that limitation.

We are providing you with new simulation files that run the package `gmapping` for you. This package runs a lidar-based SLAM algorithm that generates a map and the position of the robot on that map. That position can be accessed through the transform tree (`tf`). More on this later.

First, make sure you have the new simulation files using git:

```
$ cd ~/catkin_ws/src/asl_turtlebot
$ git pull
```

Before we begin, this problem will go through two very important tools in ROS is the `tf` package, and RViz. These tools were briefly introduced in the Homework 2 Problem 0 and 5. This problem will involve using these tools, so be sure to read up the documentation about it. Here is a short introduction to `tf`.

Using the tf package

`tf` is a ros package that keeps track of transformation between different reference frames used by your robot. The transform tree can be updated at any time by any source, and can keep track of how old certain transforms are. It's an incredibly useful tool you should read more about http://wiki.ros.org/tf. In our case, the transform tree contains a transformation between a static map and the Turtlebot. This transform gets updated by `gmapping` at some fixed rate using lidar measurements. We can then lookup that transform using a `TransformListener`. In the constructor of your node, you can initialize one using:

```
self.trans_listener = tf.TransformListener()
```

Now whenever you want to update your current pose estimate (for example at the beginning of the `get_ctrl_output` function or the `loop` function), you can do so using the `lookupTransform` function:

```
try:
        (translation,rotation) =
    self.trans_listener.lookupTransform('/map', '/base_footprint', rospy.Time(0))
except (tf.LookupException, tf.ConnectivityException, tf.ExtrapolationException):
        pass
```

Note the names of the two frames that we are looking up. The tf package also provides a few useful helper functions. In particular, you might find the function `tf.transformations.euler_from_quaternion` useful to convert a quaternion (as returned by the `lookupTransform` function) to an euler angle (used in our controller i.e. `self.theta`). `tf` becomes very important when you are dealing with multiple frames, for example, the lidar frame, the camera frame, and the TurtleBot frame. In many of the ROS geometry messages which at stamped (http://docs.ros.org/lunar/api/geometry_msgs/html/index-msg.html), they include a `frame_id` which specifies which frame that message is in. There are handy functions that allows you to automatically convert messages between frames.

Now let's take a look at the two places we rely on the position provided by Gazebo so that our stack works on the real robot too. Those files are `pose_controller.py` and `supervisor.py` (from problem set 2).

(i)  Take a look at `pose_controller.py`. Notice that there is a `use_gazebo` parameter[8]. What does

[8] https://github.com/StanfordASL/asl_turtlebot/blob/master/scripts/pose_controller.py#L29.

this variable do and what happens when this is set to false? Specifically, how is the TurtleBot's position updated? What is a `rosparam`? How would you set or check the value of a rosparam in the terminal? Read http://wiki.ros.org/rosparam to find out more. The `pose_controller.py` was first used in the homework 2 demo (problem 0). In particular, it was called using `hw2_control_demo.launch`. How does this launch file set rosparams? Can your `supervisor.py` script accommodate cases where Gazebo is not used?

(ii) **Testing your modified stack**

If `use_gazebo` is set to false, your stack should now be completely free of any dependency to the `/gazebo/model_states` topic. Can you see how rosparams can be useful when switching between simulation and the real-world testing?

We provided you with a new launch file that runs a similar environment as that in problem set 2, but with a few more features to help with the localization part of SLAM. As mentioned earlier, the launch file also runs `gmapping`, which is a package that uses a line-based particle filter in order to generate a map of the environment and localize the robot on that map. It is one of the most widely used SLAM packages.

Before you run the simulation, take a look at `turtlebot3_gmapping_sim.launch`. Can you identify what each part does? Given each section in the launch file, write down what each section is doing. For example, the first section (lines 2-6) sets parameters for for Gazebo.

You will be expected to write your own launch file for the final project so take the time to read through this (and future) launch files. Fill in the rosparams so that you (1) do not rely on Gazebo's model_states but instead rely on your map you create via `gmapping`, and (2) will use RViz to specify the desired goal poses. Take a look at `pose_controller.py` and `supervisor.py` to see how these rosparams are being used.

Run the simulation and your software stack and make sure that the TurtleBot is still able to go the the target while stopping at the stop sign. If you did things correctly, everything should work just like in problem set 2 (even though under the hood algorithms that took decades of robotics research to develop are making the whole thing much more transferable to the real world).

Launch `turtlebot3_gmapping_sim.launch`, and your `supervisor.py` from homework 2 (The default is `gui:=false`.)

RViz should open up and you can see the `tf` frames (`map, odom, base_footprint`). You can also see the lidar points that are reflected off the two large boxes and the stop sign. To help you visualize the map that the TurtleBot is creating (via `gmapping`), in the Displays window, add display markers to visualize the map and camera images (the By Topic tab shows you which topics you can currently visualize).You can also visualize the scenario form different frames[9]. You can save the RViz file so you do not have to add all the display markers again the next time you open it.

What happens to your frames when you intentionally drive into an obstacle?

(iii) Visualizing the TurtleBot via the tiny `tf` frame is can be difficult. It can be confusing to keep track which direction it is pointing, or that since the TurtleBot is not a point mass, it is unclear from RViz if it is colliding with an object. As such, visualization markers can help! Read http://wiki.ros.org/rviz/DisplayTypes/Marker to find out what type of markers are at your disposal. Write your own visualization code/rosnode to visualize the TurtleBot (as a circle, rather than a frame), and the desired pose. Feel free to add additional markers (e.g., velocity of the TurtleBot). At a high level, you need to specify the relevant entries of a visualization marker, especially the `frame_id` and `timestamp`, and then publish it so that it will be visible to RViz. You are free to write it however you like (there is plenty of documentation online or you can use existing code as a template). Having good visualization is generally good practice because it allows you to see better what ROS (or the robot) is thinking, and can potentially help you spot any bugs/unexpected behavior (a better way to see what

[9]This instruction is intentionally left vague — to encourage you to explore what RViz has to offer!

is happening than rostopic echo). We suggest you write it as a separate file so that it can be used for future homeworks and the final project. The Turtlebot is roughly 15cm in diameter.

Include a screenshot of RViz, with the map and camera displayed, and your own visualization markers. If you added additional markers other than the TurtleBot and command pose, give a description of what you have added. Explain how you would use your visualization code within a launch file. (Could you have used your visualization node in problems 1 and 2?) Include code snippets if necessary.

Problem 4: Group Effort

By now, you should be familiar with:

- Writing a subscriber and publisher
- Writing a rosnode that uses multiple subscribers and/or subscribers
- rosbag and rostopic tools
- Running launch files and rosnodes
- tf frames
- RViz

As a team, your job is to write your own rosnode that tracks a moving object using what ever method of your choosing. You can take implement one from scratch, or repurpose an existing package (but make sure to give a reference to this package).

You are given lidar measurements (3D pointcloud data), and state and control information of the TurtleBot. As a bare minimum, your implementation should visualize your pose estimate of the moving object, and the TurtleBot.