Robotics Project 3

Group Members: Sina Hazeghi, Timothy Liu (skh79, tyl19)
(all visualizations and animations can be found in respective folders in project directory as described by assignment writeup pages 1-2)

Part 1: Setup maps and generate controls

1.1 Generate 4 maps with landmarks

- To generate the maps I use the script gen_landmarks.py which takes in keyword arguments <number of landmarks to generate> and <filename to save to>
- I call a function that uniformly samples points in $[0, 2]\mathrm{x}[0, 2]$ and asserts that all sampled points are distinct

1.2 Generate Robot Control Sequences

- First I created a simple model of a differential drive robot within diff_drive.py, this model allows us to control the robot through keyboard presses and also dynamically control the robot's state using input controls.
- Within controls.py I first use a function within_bounds() which takes in a point and returns True if it is in $[0.2, 1.8]\mathrm{x}[0.2, 1.8]$
- I sample an initial pose and use within bounds to make sure it is at least 0.2 from the boundary
- In genControls() I start at my initial pose and while I don't have 10 controls I sample random controls within the bounds of my actuator constraints and integrate forward for 20 timesteps. If all the integrated positions are within bounds using my previous function this control is kept and I update my position to be the final position that I integrated to. Otherwise I keep sampling until I generate the full control sequence
- In show_visualization() I take my control sequence and initial pose and plot all visited points. At the final pose I draw a picture of a rotated rectangle using my differential drive model

Part 2: Simulate Robot Motion and Measurements

2.1 Simulate Robot's motion and measurements

- To implement the actuation model I sampled noise at every timestep independently for linear and angular controls according to the standard deviations given in the assignment and applied it to the planned controls if they are non-zero. I then clip the values so that they're within the actuator constraints
- To implement the odometry model I used a very similar process as actuation model but just using different standard deviations and applying the noises to executed controls instead of planned controls. I also don't clip the values here
- To find the distance and angle from every landmark to the current robot position, we first calculated the relative position of the landmark in the robot's local coordinates by subtracting the robot's position, and rotated the relative position based on the robot's orientation to align it with the robot heading. Using these new coordinates, we find the normal vector, which would be our calculated distance, and the inverse tan, which would be our angle. After that, we added some random noise to our measurements.

2.2 Dead Reckoning

- In dead reckoning I generate the animations of the ground truths from the saved poses while simultaneously integrating the observed controls and showing their poses in red. I then use the integrated poses from the sensed controls to generate estimates of where I think the landmarks are (these estimates are generated the same as we did with the landmark sensor but instead we don't add any extra noise here)
- We observed that the sensed controls with lower accuracy 'L' tend to deviate much more from the ground truth poses

Part 3: Implement Particle Filter for localization

To implement the particle filter, we first initialize the particle and weight arrays.

- Each particle represents a robot state. In 3.1, every particle is initially set to the initial robot orientation, while in 3.2, they are set to uniformly random values. All weights are initialized to 1.
- First, we integrate every particle by the control that was given in the reading file, with some added noise using the standard deviations in the actuation model of the robot.
- Once we've moved all of our particles we correct their weights by checking the difference in distances between the landmark readings and the distances the particles are from the landmarks, we also compare the angular distance of the readings compared to what we are getting for each particle.
- We compute the probability of getting the euclidean distances by computing the gaussian PDF value of the difference, with mean = 0, standard deviation set to 0.4. We set this standard deviation even though it is not what is in our observation model because we felt that a standard deviation of 0.02 was too precise and resulted in our particles converging too closely, making our filter both harder to visualize and slightly less accurate because it didn't account for any outliers.
- We did the same thing with the angular differences except we used a PDF from a vonmises distribution rather than a regular gaussian (we used the same standard deviation of 0.4) . A vonmises distribution is the gaussian equivalent but for circles which have a different topology than something like linear distance.
- We update the weights array by multiplying the PDF's of the two distributions together (since they are assumed to be independent), then normalize it.
- Once we normalize it, we partition the weights using the cumulative sum, then sample a number between 0 and 1 and see which partition it falls under. We do this for every particle and store the partitions that we get, then we set each particle to the index that the partition represents. In short, we resample the particles based on the weight that was given to the old particles.
- Once we resample the weights are reset to 1. The particles would eventually converge around the same area with enough iterations. The accuracy of that area compared to the car is dependent on the number of particles we use, standard deviation used and the accuracy of our sensor readings.

Part 5: Evaluation and Experiments

- In evaluation.py I generate animations similar to how I did in controls.py, except this time I use estimated poses rather than sensed controls
- To create the graphs that visualize error I simply compute the euclidean distance between estimated poses and ground truth poses at each timestep using np.linalg.norm and the difference of the xy coordinates of the poses
- To compute angular distance I used an approach where I compute the sin and cosine of the differences between the angles and then computed the arctan2 of that, effectively mapping angular difference to a linear ratio in order to respect angular topology.

Part 6: Kalman Filter

To implement the Kalman Filter to estimate the position of the landmarks, we first defined the state transition and measurement models. The state transition model represents how the robot's state evolves over time. For us, we set it as a vector containing the x, y, theta, velocity and phi. The measurement model is just the integration of the controls onto the robot position.Next, we initialized the state estimate to $[0, 0, 0]$ and the error covariance matrix. We then predicted the next state estimate and error covariance by calculating the Jacobian matrices of the state and measurement models. After that, we put it in the Kalman gain equation to get our new state estimate and covariance. Unfortunately, due to time, we were unable to finish integrating the algorithm into Python, but we put what we had in kalman_filter.py.