Albert-Ludwigs-Universität Freiburg
Lecture: Introduction to Mobile Robotics
Summer term 2015

Institut für Informatik

Prof. Dr. Wolfram Burgard
Dr. Gian Diego Tipaldi
Dr. Barbara Frank

Sheet 6

Topic: Discrete Filter, Particle Filter

Submission deadline: June 18, 2015
Submit to: `mobilerobotics@informatik.uni-freiburg.de`

Exercise 1: Discrete Filter

In this exercise you will be implementing a discrete Bayes filter accounting for the motion of a robot on a 1-D constrained world.
Assume that the robot lives in a world with 20 cells and is positioned on the 10th cell. The world is bounded, so the robot cannot move to outside of the specified area. Assume further that at each time step the robot can execute either a *move forward or a move backward* command. Unfortunately, the motion of the robot is subject to error, so if the robot executes an action it will sometimes fail. When the robot moves forward we know that the following might happen:

1. With a 25% chance the robot will not move
2. With a 50% chance the robot will move to the next cell
3. With a 25% chance the robot will move two cells forward
4. There is a 0% chance of the robot either moving in the wrong direction or more than two cells forwards

Assume the same model also when moving backward, just in the opposite direction.
Since the robot is living on a bounded world it is constrained by its limits, this changes the motion probabilities on the boundary cells, namely:

1. If the robot is located at the last cell and tries to move forward, it will stay at the same cell with a chance of 100%
2. If the robot is located at the second to last cell and tries to move forward, it will stay at the same cell with a chance of 25%, while it will move to the next cell with a chance of 75%

Again, assume the same model when moving backward, just in the opposite direction.
Implement in Python a discrete Bayes filter and estimate the final belief on the position of the robot after having executed 9 consecutive *move forward* commands and 3 consecutive *move backward* commands. Plot the resulting belief on the position of the robot.

Hints: Start from an initial belief of:

$$bel = numpy.hstack\left((numpy.zeros(10), 1, numpy.zeros(9)\right))$$

You can check if your implementation has bugs by noting that the belief needs to sum to one (within a very small error, due to the limited precision of the computer). If it doesnt there is a mistake.
Careful about the bounds in the world, those need to be handled ad-hoc.

General Notice

In the following exercises you will implement a complete particle filter. A framework containing the motion model and the measurement model is provided to you, so you can concentrate on the implementation of the filter itself. The following folders are contained in the tarball:

data This folder contains files representing the world definition and sensor readings used by the filter.

code This folder contains the particle filter framework with stubs for you to complete.

To run the particle filter, just run in the terminal `python particle_filter_framework.py sensor.dat world.dat N` , where N is the number of the particles. *Note: You first have to complete the functions resample_particles, get_mean_position and measurement_prob_range in order to get the filter working correctly.* We are not using any external library for visualizing the particle cloud. The stub for plotting the particles is included in the framework. Once you have completed the particle filter, you should be able to see a simple matplotlib figure popped up for visualization. Some implementation tips:

- Turn off the visualization to speed up the computation by commenting the lines in the function `get_mean_position`.
- To read in the sensor and world data, we have used dictionaries. Dictionaries provide an easier way to access data structs based on single or multiple keys. The functions `read_sensor_data` and `read_world_data` read in the data from the files and build a dictionary for each of them with timestamps as the primary keys. To access the sensor data from the data_dict, you can use

  ```
  data_dict[timestamp,'sensor']['id']
  data_dict[timestamp,'sensor']['range']
  data_dict[timestamp,'sensor']['bearing']
  ```

  and for odometry you can access the dictionary as `data_dict[timestamp,'odom']['r1']`

  ```
  data_dict[timestamp,'odom']['t']
  data_dict[timestamp,'odom']['r2']
  ```

  To access the positions of the landmarks from world_dict, you can simply use

  `world_dict[id]`

Exercise 2: Theoretical Considerations

(a) Particle filters use a set of weighted state hypotheses, which are called particles, to approximate the true state x_t of the robot at every time step t. Think of three different techniques to obtain a single state estimate $\bar{x}_t$ given a set of N weighted samples $S_t = \{\langle x_t^{[i]}, w_t^{[i]} \rangle \mid i = 1, \dots, N\}$.

(b) How does the computational cost of the particle filter scale with the number of particles and the number of dimensions in the state vector of the particles? Why can a large dimensionality be a problem for particle filters in practice?

Exercise 3: Measurement Model and Resampling

A particle filter consists of three steps listed in the following:

(a) Sample new particle poses using the motion model.

(b) Compute weights for the new particles using the sensor model.

(c) Compute the new belief by sampling particles proportional to their weight with replacement.

The motion model (a) is already implemented in the provided framework. In this exercise you are asked to implement both (b) and (c).

(b) Complete the function `measurement_prob_range`. This function should implement the update step of a particle filter, using a *range-only* sensor. It takes as input landmarks as world_dict and landmark observations as ids and range values.

   It returns the weight for each particle. See slide 15 of the particle filter lecture for the definition of the weight w. Instead of computing a probability, it is sufficient to compute the likelihood $p(z|x, l)$. The measurement standard deviation is $\sigma_r = 0.2$.

(c) Complete the function file `resample_particles` by implementing stochastic universal sampling.

   It takes as an input a set of particles and the corresponding weights, and returns a sampled set of particles.