COMPSCI 170 - Artificial Intelligence
Programming Project II
Robot Localization

Due date: April 25
(will be accepted w/o penalty up until May 9)

1 Description

In this programming project you will implement a simple particle filter for robot localization. We have done many things to make this easy for you, so the actual implementation of the particle filter shouldn't be more than a page or two of code.

In this version of the localization problem, everything is discrete to save you the trouble of dealing with Gaussians and continuous distributions. The robot's position is a discrete position in a grid, and the robot's sensors measure distance in discrete units in the 8 directions: N, NW, W, SW, S, SE, E, NE. (Check out the comments in the `sensor.c` file to see how the readings are mapped to physical directions. You will find some other helpful functions in this file as well.)

In the class directory, you will find code for reading in maps, generating ground truth data, and generating sensor logs. Your code will need to do the following:

1. Read in a map. (This is trivial because we have already provided code for this.)
2. Read sensor log files of the form generated by `runner.c` and perform localization using a particle filter.
3. Output the (weighted) mean and variance of the particle positions at each time step. To help with your debugging, you may want to write some additional code for visualizing the positions and weights of your particles on the map, but this is not required.
4. Compute the average error per time step of your particle filter.

The README file contains additional information about how to specify noise models and how to work with the distribution files. You should write you code to accept any noise file and any sensor log. We will provide you with some more sensor logs and noise models to help with your debugging and a specific set to use for your final submission.

2 Deliverables

To get full credit for this project, you will need to turn in the following:

- (60 points) A working particle filter implementation.
- (10 points) A "sanity check" demonstration that your particle filter tracks correctly when given a deterministic noise model and data generated from a deterministic noise model.

- (10 points) A demonstration that your particle filter produces reasonable average particle positions when run with noisy sensors and at least 10 particles.
- (10 points) A demonstration that your particle filter can figure out the robot's position fairly accurately in the "kidnapped robot" problem. To do this, you will need to start with a very large number of particles scattered uniformly over your map and then show that they converge to the correct position. You can do this by plotting your particles on the map, or by showing in text output how the mean approaches the correct position and how the variance reduces.
- (10 points) A graph showing accuracy as a function of particles for 1, 5, 10 and 100 particles. Demonstrate accuracy by computing the average error for each run (a run is a complete parse of the sensor data) and then average this over at least 10 runs. Include error bars on your graph to show variance. (Be sure to indicate what the error bars mean.)

3 Implementation details

3.1 Odometer

The odometer data are stored as increments from the previous position. For example, an odometry reading of $(+1, +1)$ indicates that X and Y have both increased by 1 since the last reading. You should treat the odometer reading as the center of the distribution of possible robot positions. For example, if the odometer noise model indicates 0 noise with probability 0.6, then your motion model should accept the odometer reading at face value with probability 0.6. If the odometer noise model subtracts 1 with probability 0.1, then with probability 0.1, you should subtract 1 from the odometer reading in your motion model.

3.2 laser

The "laser" in our simulation functions a bit more like sonar. It reads in 8 directions and produces noisy data. To compute the probability of a laser reading given a simulated robot position, you should compute the expected sensor reading given the robot position. (You can use our code to do this.) You should use the laser error model to compute the probability of the reading in comparison to what the reading *should* be given your robot position.

If a reading has larger noise than any possible laser error, then the observation should get weight 0. Note that this can cause problems in some cases, such as the kidnapped robot example. In such a case, it's possible that no particle will have the correct robot position. This can cause all of your particles to have weight 0 and can cause divide by 0 errors if you aren't careful about how you do your normalization/resampling. A common trick/cheat to avoid this is to avoid assigning 0 probability to any measurement and assign a very tiny positive probability to even the worst sensor reading. It's actually possible to come up with some creative theoretical justifications for this hack, but the main practical reason is that it prevents some degeneracies in your code.

Note that it's possible to get negative laser readings with some noise models. While these may not make much sense physically, you can still use them in your particle filter.

Finally, you may notice that it's possible to simulate robot positions that are physically impossible, e.g., a robot position that places the robot inside of a wall. Deal with this by assigning any measurement made from inside of wall probability 0.