Particle filter localization, Due 11/10, 11AM step-by-step

You will implement particle filter localization as discussed in class. The steps are below.

1. World Map: You will read in a file that contains the following:

World_X_Dimension (cm) World_Y_Dimension (cm)
Obstacle_1_X Obstacle_2_Y
Obstacle_2_X Obstacle_2_
……..
Obstacle_N_X, Obstacle_N_Y

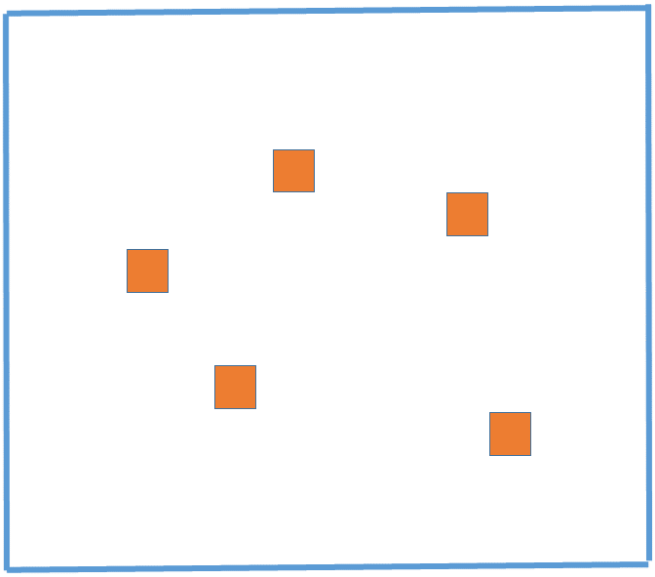
For example, the file:

200 300
20 20
40 40
150 200

Will be a world 200 by 300 cm, with obstacles centered at 20,20; 40,40; and 150,200. Each obstacle is an orange plastic cone, dimensions 11.4 x 11.4 cm, with the cone center at the obstacle X,Y coordinates. You can assume the square base of the cone is aligned with the coordinate axes. So if the cone center is at (20,30) the four

corners of the cone are at (20-5.7,30-5.7), (20+5.7,30-5.7), (20-5.7,30+5.7) coordinate in the file.

2. Draw your world and obstacles in a graphic interface.
You can draw your world and obstacles graphically using Python Turtle graphics: https://docs.python.org/3/library/turtle.html)



Sample World with obstacles

3. Implement a particle filter algorithm to localize the robot. You can use the code given in class as a guide. You will have to implement a virtual sensor to replicate the GoPiGo's ultrasound sensor for measuring distances to the obstacles. We will discuss how to do this in class.

4. **What your program should do:**

1. **Draw the initial world and obstacles, with particles shown as small filled blue circles, and a red circle representing the GoPiGo's initial**

location (you need to specify this). Because you have so many particles, only display every tenth sample.

2. **Run the algorithm and show the best localization after a reasonable number of updates. This will show the final particles (sample only a tenth of the final particles for graphical display), the final robot location, and the location of the mean X,Y of all the particles. Report the number of updates (movement + prediction) as well as the mean particle error (distance from robot to each particle), and the mean particle location (X,Y), mean orientation and actual GoPiGo location and orientation (you need to measure this in your world –use the blue measuring tape we gave you!).**
3. **Graphics is compute intensive, so you only need to show starting and final configurations. You may want to show intermediate configurations of the particle set for debugging, but it is slow.**
4. **We will discuss details and subtleties of this lab in class.**
5. **We will set up a test environment in CEPSR to allow you to test your code live during normal hours.**

Notes/Tips to implement this Lab:

1. Make sure you understand the sample particle filter program discussed in class: http://www.cs.columbia.edu/~allen/F16/NOTES/simple_particle_pka.txt
2. Use this program as a template if you want. Here are some of the differences you need to understand to implement the particle filter on the GoPiGo.
3. First, you need to create a movement (move function in class robot) that randomly moves the robot at each iteration of the algorithm.
    a. Set a suitable threshold which tells the robot that there is an obstacle in front. Make sure you use the ultrasonic senor to detect the obstacle. If there is no obstacle close movement is easy.
    b. For the movement each time, you can let the car perform random movement (go for/backward, turn left/right). However, due to the odometry errors caused by turning performance, we recommend to do fewer rotations in the car movement. A good way to do that is to keep the car moving forward without rotation (you can determine the step

size the car should move each time by setting the encoder values). A good choice is to use 10 encoder values and vary the step size each time by using a rule like a Gaussian or uniform Distribution. If an obstacle is encountered within the movement window, then make a random turn until there is no obstacles in front and then move forward again. That will reduce the rotations and odometry errors.

4. Each virtual particle needs to sample the world using a virtual ultrasound. This involves taking scans every few degrees between 0-180 degrees and seeing if there is an obstacle or a wall. The virtual ultrasound function should return a list of samples (say every 20 degrees) over the 0-180 degree range at each measurement cycle.
5. Once you have the virtual ultrasound scan, you need to compare it to the real robot scan. Simply find the absolute value of distances between the virtual and real scans and send that through a Gaussian probability function where the mean is the actual distances from the real scan. This probability becomes the importance weighting. Particle virtual scans whose distances are small from the real scan will be weighted higher than those further away.
6. Number of particles. Since you are sampling X,Y Theta with each particle, and you need a large number of particles to get convergence. Using 1500 or more particles is probably required, and this will be a drain on the Raspberry Pi's ability to process efficiently. At each cycle, you are doing 1) a movement

of N particles, 2) a virtual ultrasound scan and probability weighting of N particles, and 3) a resampling of N particles.

HINT: - get everything working OFF the robot in virtual space. Once you get the filter converging OFF the robot, then you can move to testing on real hardware.