E160 – Autonomous Mobile Robots

Lab 4

Particle Filter Localization

INTRODUCTION
Determining a robots position in a global coordinate frame is one of the most important and difficult problems to overcome in enabling mobile robots to navigate an environment and carry out tasks autonomously. In lab 2, you used odometry for localization and saw first hand how errors accumulate with distance travelled.

In this lab, you will implement a particle filter localization algorithm. At each iteration of the algorithm, odometry is used to propagate the robot motion of every particle. Then these particles are assigned weights based on how closely the current range sensor measurements match with expected range measurements. The expected range measurements are calculated using the propagated particle state and a map of the environment. The particle distribution is then resampled based on the particle weights.

BACKGROUND
There are several steps to implement a particle filter localization algorithm, and this will take quite a bit more time than previous labs. The algorithm is outlined in the slides for Lecture #15. See Sebastian Thrun's text *Probabilistic Robotics* for additional details.

For our code base, (new version for lab 4), the robots main control thread update function in E160_robot.py looks like:

```
def update(self, deltaT):

        # get sensor measurements
        self.encoder_measurements, self.range_measurements =
                                        self.update_sensor_measurements(deltaT)

        # update odometry
        delta_s, delta_theta = self.update_odometry(self.encoder_measurements)

        # update simulated real position, find ground truth for simulation
        self.state_odo = self.localize(self.state_sim, delta_s, delta_theta,
                                        self.range_measurements)

        # localize with particle filter
        self.state_est = self.PF.LocalizeEstWithParticleFilter(
                                self.encodermeasurements,self.range_measurements)

        # to out put the true location for display purposes only.
        self.state_draw = self.state_odo
```

A few important notes here:

- The variable state_odo is set to be that produced from the odometry (using either simulated or actual encoder measurements). This is only really useful for simulator mode.
- The variable state_est is the estimated state will be calculated with the particle Filter
- The variable state_draw is the state drawn on the GUI with the robot image.

A new addition to the code base is the file E160_PF.py. This contains all the particle filter code. You will add most of your code (if not all of your code) here. To start, notice how we have created a particle class at the bottom of the file:

```
class Particle:
	def __init__(self, x, y, heading, weight):
		self.x = x
		self.y = y
		self.heading = heading
		self.weight = weight

	def __str__(self):
		return str(self.x) + " " + str(self.y) + " " + str(self.heading) + " " +
		                          str(self.weight)
```

Each particle in the filter has an x, y, theta, and weight value. The key functions within the E160_PF.py file for which you will have to add code include:

```
def InitializeParticles(self)

def LocalizeEstWithParticleFilter(self, encodermeasurements, sensor_readings)

def Propagate(self, encodermeasurements, i)

def CalculateParticleWeight(self, sensor_readings, walls, particle)

def GetEstimatedPos(self)

def FindMinWallDistance(self, particle, walls, sensorT)

def FindWallDistance(self, particle, wall, sensorT)
```

Make sure you can find these in your file, because you will be adding to them shortly.

CODING AND EXPERIMENTS

First, let's get some geometry done:

1. Determine the distance to a wall

Using geometry, you need to calculate the distance to a wall using the laser range sensor. You will need to modify the function `FindWallDistance(self, particle, wall, sensorT)`. The `particle` variable includes a robot x, y position and heading with respect to the global coordinate frame. The variable `sensorT` is the angle of direction of the range sensor mounted on the robot, with respect to the robot's local coordinate frame.

In this function, you must calculate the expected range measurement *d* from a robot sensor to a `wall`. The `wall` is defined by a line segment with two endpoints `[X1, Y1, X2, Y2]` as defined at the top of E160_environment.py.

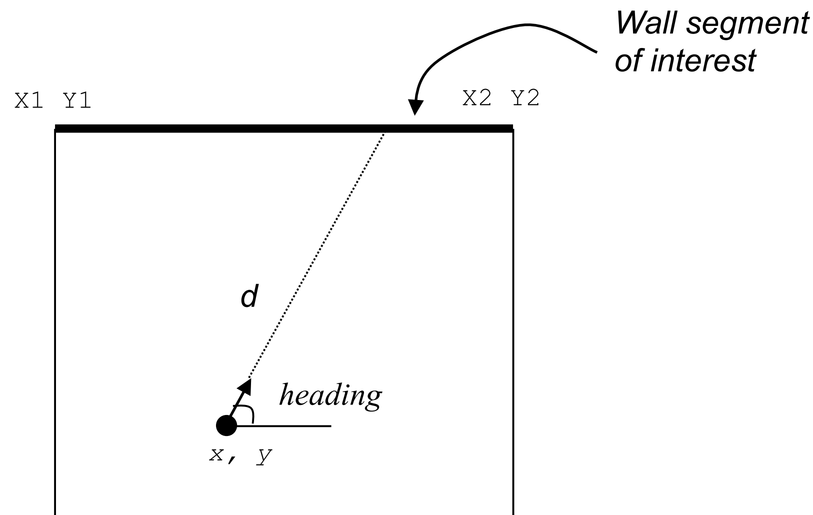

Figure 1: Calculating the distance to a wall segment

An easy way to calculate *d* is to calculate the point of intersection between the line segment and the ray cast by the range sensor transmission signal. This assumes the sensor transmits a linear signal with no energy dissipation.

Next calculate the distance between the point of intersection and the robot. Make sure that a **point of intersection actually exists**.

2. Determine the distance to closest wall

The function `FindMinWallDistance(self, particle, walls, sensor)` will return the shortest measurement that a range sensor should receive given the environment map. Within this function, loop through all wall segments to find the closest segment by calling the function `FindWallDistance`.

This function will be called from two other functions. First, the robot simulator in the `update_sensor_measurements` function of E160_robot.py will call `FindMinWallDistance` to determine what the sensor measurements would be, given the current state and the environment. Second, the function will be used by your particle filter to determine expected measurements for particular particle locations.

Drive the robot around the simulated environment to ensure that the range measurements make sense, (top left of GUI).

3. Create an initial set of particles

When the object Robot is constructed, it calls a function `InitializeParticles()`. This function will iterate on all particles and initialize their states using either of the two functions:

```
self.SetRandomStartPos(i)
#self.SetKnownStartPos(i)
```

Within `SetRandomStartPos`, set the position of particle `i` to be some random location within the boundaries of the environment. Feel free to make use of the random.random() function, and variables like `self.map_minX, self.map_maxX` that are set in the E160_PF constructor. Note that in E160_PF the number of particles is defined as `numParticles`.

4. Propagate particles

The prediction step within the particle filter is accomplished by propagating particles forward based on odometry. Within `LocalizeEstWithParticleFilter`, create a loop that iterates on all particles.

At each iteration of the loop, call the `Propogate`, function to update particle `i`'s position and heading using odometry along with some randomness added to EACH wheel's distance travelled, to determine the new predicted state of the particle. Be sure to store this predicted state in `self.particles[i]`. Then, for each iteration of the loop, call the function `CalculateWeight` to set `self.particles[i].weight`.

5. Weight the particles

Within the function `CalculateWeight`, compare any of the range measurements in `sensor_readings` to expected range measurements for the `particle`'s state within the map. You will use the function `FindMinWallDistance` that you created before. Use this comparison between range measurements and expected range measurements to calculate the weight that is returned at the end of the `CalculateWeight` function.

6. Resample the particles

Once the set of propagated particles have been created and weighted, you can create the corrected set of particles `particles`. This should be done after the prediction step in `LocalizeEstWithParticleFilter`. This resampling can be accomplished by randomly selecting particles from the predicted particles with increased likelihood of selection given to those particles with high weights.

7. Calculate the state estimate

At the end of `LocalizeEstWithParticleFilter`, calculate and return the state estimate using a function `self.GetEstimatedPos()` that you must write. Within `self.GetEstimatedPos()`, you must take the average of all particle states to calculate the estimate.

8. Known start position simulations

Use the `InitializeParticles()` function to initialize all particles at the known start position at [0 0 0]. Drive the robot around the simulated environment. Tune your Particle Filter parameters so that the estimated state matches the actual robot state.

9. Unknown start position simulations

Use the `InitializeParticles()` function to initialize all particles at random start positions. Drive the robot around the simulated environment. Tune your Particle Filter parameters so that the estimated state converges to the actual robot state.

10. Hardware experiment

Develop a maze using hardware experiment that demonstrates how well your particle filter localization algorithm works. You can use whatever configuration of walls you wish, along with your choice of paths to follow.

DELIVERABLES

1. Demonstration

Before the end of the final day of this lab, Monday March 26th at midnight, you must demonstrate to the TA or instructor that your PF localization algorithm is working properly. In both simulation and hardware mode, the display window should show the actual robot states and state estimates match.

Part of your grade will be based on performance: How well do state estimates match actual robot position? How stable is the controller when using the PF state estimates for feedback? Is the kidnapped robot problem solved?

2. Submissions

In a 5-10 page report, (similar format to lab 3), present your methods for PF localization. Discuss any decisions you made in your algorithm design, E.g. your sampling strategy, how you propagated states, what sensors you used for weight calculations, how you picked your environment, etc. Provide plots and data tables that demonstrate the performance of your algorithm in simulator mode. The report is due via email or hardcopy to the instructor by midnight on Monday. April 2nd.

You must demonstrate your PF in simulation and on the real robot by Monday March 26th at midnight.