

CPE 482 – Autonomous Mobile Robots

Lab 4

Particle Filter Localization

INTRODUCTION

Determining a robot's 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 #10. See Sebastian Thrun's text *Probabilistic Robotics* for additional details.

EXPERIMENTS

Note, download the most recent version of the base code for lab 4. All coding for steps 1 through 2 will occur within the file `Map.cpp`. Remaining steps require modification of the file `Robot.cpp`. You will need to cut and paste your odometry localization code and point tracking code from lab 3. You can leave out the trajectory tracking code for now.

The main control loop `Robot :: RunControlLoop(CWiRobotSDK* m_MOTSDK_rob)` calls three localization functions:

```
// Localize
MotionPrediction(m_MOTSDK_rob);
LocalizeRealStateWithOdometry(m_MOTSDK_rob);
LocalizeEstStateWithParticleFilter(m_MOTSDK_rob);
```

At this point of the course, we will use the function `LocalizeRealStateWithOdometry` to determine the actual state $[x \ y \ t]$ of the robot in simulations. Make sure that x , y , and t are set in this function.

To estimate what this actual position is, we use a particle filter localization algorithm. This will be implemented in the function `LocalizeEstStateWithParticleFilter`. Make sure that you set x_{est} , y_{est} , and t_{est} in this function.

First, let's get some simple geometry done within `Map.cpp`:

1. Determine the distance to a wall

Using geometry, you need to calculate the distance to a wall using the range sensors. You will need to modify the function `Map :: GetWallDistance(double x, double y, double t, int segment)`. The `x`, `y` variables are the position of the robot, and `t` is the orientation of the sensor wrt the global 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 $[x_1 \ y_1]$, $[x_2 \ y_2]$.

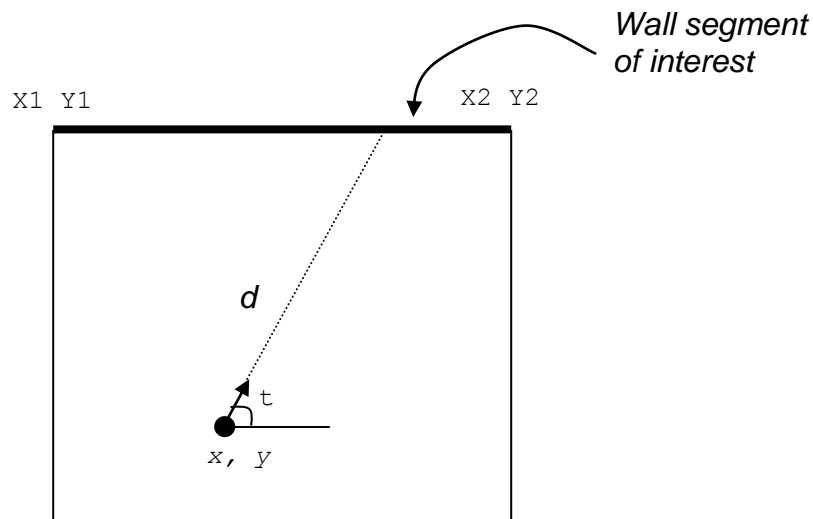


Figure 1: Calculating the distance to a wall segment

An easy way to do this 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.

Note that the variables `slopes`, `intercepts`, and `segmentSizes` for each segment are calculated for you in the `Map` constructor. These may be useful.

2. Determine the distance to closest wall

This function 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 `Map :: GetWallDistance(double x, double y, double t, int segment)`.

This function will be called from two other functions. First, the robot simulator will call `GetClosestWallDistance` 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.

Beware that this function returns the ranges from the center of the robot. The simulator will subtract off the distance between the sensor and the middle of the robot. You can set these distances `sonarRadius` and `IRRadius` in `Robot.h`. Also, the simulator will limit the measurements to ensure they are within the min/max values of the real sensors (e.g. for IR range sensors between 10 and 70 cm). These limits are also set in `Robot.h`.

Drive the robot around the simulated environment to ensure that the range measurements in your dialogue window make sense. Your code is linked to the text boxes on the dialogue window so that in simulator mode, they range measurements for each sensor should appear in real time.

3. Create an initial set of particles

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

```
SetRandomPos(i);  
//SetStartPos(i);
```

Within `void Robot :: SetRandomPos(int p)`, set the position of particle `p` to be some random location within the boundaries of the environment. Feel free to make use of the `rand()` function, and variables like `robotMap.maxX`, `robotMap.minX` that are set in the `Map` constructor. Note that in `Robot.h`, the number of particles is defined as `numParticles`.

4. Propagate particles

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

At each iteration of the loop, use the previous state of the particle [`particles[i].x` `particles[i].y` `particles[i].t`], the odometry [`wheelDistanceR` `wheelDistanceL`] along with some randomness in each wheel's distance travelled, to determine the new state of the particle [`propagatedParticles[i].x` `propagatedParticles[i].y` `propagatedParticles[i].t`]. Then call the function `CalculateWeight(int p)` to calculate the weight of the particle.

5. Weight the particles

Within the function `CalculateWeight(int p)`, compare any of the range measurements `[sonar1, sonar2, sonar3, IR1, IR2, IR3, IR4, IR5, IR6, IR7]` to expected range measurements given particle `p`'s state within the map. Use this comparison to calculate the weight of the particle, `propagatedParticles[p].w`. You will need the function `GetClosestWallDistance` that you created before.

6. Resample the particles

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

7. Calculate the state estimate

At the end of `LocalizeEstStateWithParticleFilter`, determine the state estimate `[x_est y_est t_est]` by taking a weighted average of all particle states.

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. Make sure your point tracker works.

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, you must demonstrate to the Professor that your PF localization algorithm is working properly. In both simulation and X80 mode, the OpenGL window should show the actual robot states and state estimates match.

Part of your 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 2-10 page report, 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.