1. Representing Robot Pose Uncertainty

Implement the sample method for representing the pose PDF of the robot. At a minimum, this means defining a class with the following functions and data:

   a) A set of poses that represent the x,y,θ pose of the robot in the global workspace. Usually we’ll work with 500 to 5K poses; it’s nice to be able to initialize the pose set to an arbitrary number of poses.
   
   b) A function to initialize the poses to a uniform distribution within a given area, and another function to initialize them to a gaussian distribution with mean and variance in each of the three pose dimensions.
   
   c) A function to move the poses according to the robot motion. This function should take as arguments the robot error characteristics, along with the robot initial and final <x,y,th> coordinates from odometry. It should update every pose in the pose set.
   
   d) A drawing function for showing the poses as a set of dots in a graphics window.

To test your functions, try the following two scenarios. (1) Initialize the pose set to a uniform one within a small position window, but a random angle (i.e., a uniform set of angles from 0 to 360 degrees). Move the robot one meter forward, turn around 180 degrees, and move it one meter back. (2) Initialize the pose set to a single point, then move the robot forward 4 steps of 1 meter. How does the pose cloud evolve in each case?

This part can be done in simulation.

2. Robot Pose Updating with a Map

Connect your sample functions to robot movement. Write a program that will update the point cloud according to the robot motion. Use a map, and eliminate samples that violate the map constraints. Some things to note:

   a) Don’t update every cycle, since it can be expensive. Wait until the robot has moved or turned enough, e.g., 300 mm and/or 10 degrees.
   
   b) You can figure out how far the robot has moved and turned, as well as the direction it has moved, by comparing the current robot pose to its pose at the end of the last update.
   
   c) Use an independent thread to do the heavy computation of the update.

This needs to be tested on the robot, to make sure that the error model is realistic.

3. Markov Localization using a Particle Filter

This is the key part of the project. Here the update part of ML must be programmed. First, you must construct a sensor model for the laser to compute \( p(s|d) \) efficiently. You can use the correlation method in Konolige and Chou, or a ray-tracing method. Renormalize the sample set and draw it.

Finally, integrate the update part of ML with the robot motion part, keeping track of both the dead-reckoned and real-world position of the robot. One problem: how to determine the current position of the robot, based on the sample set. Make sure you can get an estimate of the RW pose of the robot at any time!

To test this whole scheme, start with the robot well-localized in some part of the map (use a Gaussian initial distribution), and see if it stays localized as you move. Use your goal-directed obstacle avoidance to move the robot to different positions in the map.

Extra Credit: See if you can integrate the global compass (from the first project) into the localization filter.
4. **Doing without Odometry**

   ML is robust to many types of error. You should be able to run ML even without good odometry, or any odometry at all. Turn off odometry, and modify your method appropriately to maintain localization. Run the robot using a joystick or the Wander program.

   For this part, you need to think carefully about the parameters you are using – laser sensor model (the uniform distribution ratio), number of samples, etc.

5. **Globalization**

   This is the ultimate test of ML. Start with a uniform sample set over the space of the map, or some reasonable portion of the map. Move the robot around, periodically sampling the lasers and performing ML, until the robot becomes localized. You can move the robot by hand here.

   The initial sample set should be quite large to cover the required space. As the robot becomes more localized, the sample set size should adapt, going from (say) 10K points to 200 when it is well-localized. You can use any of the methods discussed (likelihood, KLD) to adapt the sample size.

   Something to try: use the sensor model and map to seed the samples at likely locations, rather than initializing them randomly over the whole space.