Assignment 4: Particle Filter Localization
CS 393R: Robotics

Due Date: Thursday, October 22, 2009

Your task: Write the sensor update and re-sampling steps of a particle filter.

This assignment is to be done **individually**.

In this assignment, we will be using UTNaoTool, the tool developed by Austin Villa to develop and debug code on the humanoid Nao robots. You will be using UT's Nao codebase, and filling in missing parts of the particle filter. You will implement the sensor update and re-sampling steps of the particle filter and compare their predictions to those of the particle filter that the team uses.

I have tested that the code works on the department machines, therefore I would recommend that you work on them. It does NOT work on the machines in the lab. If you prefer to sit in the lab, you can always ssh into one of the department machines and run it that way.

Steps:
1. Copy the code from my folder to wherever you are going to use it: scp -r emos.cs.utexas.edu:/u/todd/cs393r/cs393r .
2. Compile the code for the first time. Go into the tools/UTNaoTool directory, and type make.
3. Take a look at the tool. In the tools/UTNaoTool directory, type ./UTNaoTool. Now open a log from the file menu. Now click on Vision or World. Here you can see what the robot saw and where it thinks it is.
4. In the world window, hit F2 to bring up a display of all the particles. The robot is shown at the weighted average of all the particles.
5. Now open core/localization/particleFilter/PFLocalization.cpp in an editor. In here you'll find two methods you need to fill in: resampleParticles and updateParticleFromObservations.
6. Once you've edited the file, you can re-compile the tool with your changes by typing make in the tools/UTNaoTool directory.
7. Then re-open the tool and open the log file. On the main window, you can toggle between 'view log' which will show the localization output saved in the log from our team's localization, and 'run core' which will show the output from your localization method. You can open the world window and toggle between these to compare them.

There are 4 logs for you to play with. log1 is a log of the robot walking back into its own goal. log2 is the same but I kidnapped the robot multiple times on its way. log3 shows the robot is chasing a ball, to show you how little it sees when doing this behavior. sim.log is a log from the webots simulator.

You will be implementing vision updates both for **distinct** and **ambiguous** observations. Distinct observations would be observations of the blue goal, the left blue post, etc, where there can only be one landmark that matches the observation. For these observations, you can simply update the particle's probability based on the expected and observed readings. Ambiguous observations are things like unknown blue post (where it might be the left or right post) or unknown intersection (where it could be any line intersection on the field). For these, you'll want to find the landmark with the highest likelihood for each particle and update based on it.

One thing to note is that you are working with 1000 particles. Due to processing constraints, when we run on the robot, we typically only use 100 particles, making some of these problems much harder.

Checklist:

[____] (2 points) Implement particle vision updates from known observations (i.e. goals, left post, right post).

[____] (2 points) Implement particle vision updates from ambiguous observations (i.e. unknown post, unknown intersection).

[____] (2 points) Implement and demonstrate particle re-sampling.

[____] (2 points) Demonstrate localization accuracy similar to what our particle filter achieved (in 'view log').

[____] (2 points) Clarity and quality of your memo. Turn it in at the time your assignment is graded.

[____] (1 points) **Extra Credit:** Implement updates from line observations **OR** Implement clustering or some other method to get the robot's pose instead of weighted averaging of the particles (in estimateRobotPose()).

Documentation:

- Particles
  - These are defined in core/localization/particleFilter/Particle.*
  - You can get the particle's probability, position, or orientation through: getProbability(), getPosition(), getAngle().
  - You can set the particle's probability, position, or orientation through: setProbability(double), setPosition(Point2D, Rect), setAngle(double).
  - You can also "degrade" the particle's probability, multiplying it by some factor, using: degradeProbability(double).
  - You can determine the expected bearing and distance to an object from a particular particle position using: getDistanceToPoint(Point2D) and getBearingToPoint(Point2D).
  - We have one non-traditional thing in the code: we randomly move each particle based on its probability. This causes the particles to spread out, as the lower probability particles will move out more. This is currently commented out, but you can put it back in by uncommenting the line randomWalkParticles().
- World Objects
  - These are defined in core/common/WorldObject.*
  - These objects describe where everything is on the field, as well as tell you if something was seen and where it was seen.
  - World Objects 11-24 are things that the robot might see (including ambiguous observations such as unknown goal post or unknown line intersection.
  - World Objects 11-16 and 25-38 are actual objects on the field (including all the specific line intersections that the ambiguous observation might match to).
  - What the robot sees:
    - If you get the world object, the seen boolean tells if the robot saw that object that frame.
    - visionDistance and visionBearing tell you the distance and bearing that the robot saw the object.
  - Actual objects
    - The object's loc variable (a Point2D) tells you where the object is located on the field.
- Geometry, Point2D
  - You may want to use some of the existing code in Point2D and Geometry. This is located in core/common/Geometry.*
- Parameters
  - You can set the parameters at the top of the file in core/lua/cfgpart.lua.
  - Some of these parameters won't be used in your code.
  - But others you can use to define how often resampling occurs, etc.
- Debug
  - You can add print statements that will show up in the tool's 'Log' window at the frame they were printed by using: memory->log(int, "something something"); where the int is the verbosity level from 0-100.
  - The log window will display all messages from that frame, and in the bottom left you can set the range of message levels you want to see.