MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.081—Introduction to EECS I
Spring Semester, 2007

Work for Lucky Week 13

- Software lab for Tuesday, May 8
- No pre-lab or tutor problems this week
- Robot lab for Thursday, May 10
- `No Post-lab problems`

Robot localization

In Simulation

Download the file `ps13-code.zip`. It should have the following files:

- `AvoidWanderTB.py`
- `GridMap.py`
- `GridStateEstimator.py`
- `KBest.py`
- `Map.py`
- `search.py`
- `Sequence.py`
- `utilities.py`
- `WanderEstBrain.py`
- `WriteIdealReadings.py`
- `XYDriver.py`
- `XYEstBrain.py`
- `XYGridPlanner.py`

Edit `WanderEstBrain.py` so that the variable `dataDirectory` is defined to be whatever path you unpacked the code file into.

Now, start up SoaR in simulation, using the `She` world, and use `WanderEstBrain.py` as the brain. When it starts up, you'll see two new windows.

The first window, labeled `Belief`, shows the outline of the obstacles in the world, and a grid of colored squares. Squares that are colored black represent locations that cannot be occupied by the robot. For the purposes of this window, for each $(\mathtt{x}, \mathtt{y})$ location, we find the θ^* value that is most likely, and then draw a color that's related to the probability that the robot is at pose $(\mathtt{x}, \mathtt{y}, \theta^*)$. The colors go in order from more to less likely: yellow, red, blue, gray. At the absolutely most likely pose, the robot is drawn, with a nose, in green.

The second window, labeled `P(O|S)`, shows, each time a sonar observation `o` is received,

$$\max_{\theta} \Pr(\mathtt{o}|\mathtt{x}, \mathtt{y}, \theta) \ ,$$

for each square $\mathtt{x}, \mathtt{y}$. That is, it draws a color, as above, that shows how likely the current observation was in each square, using the most likely possible orientation. Note, though, that the values drawn

in this window aren't normalized (they don't sum to 1); we've scaled the colors to make them sort of similar to the colors in the belief window, but they aren't directly comparable.

The brain `WanderEstBrain.py` just uses our standard avoid and wander program (from week 2!), but keeps the robot's belief state about its pose updated as it does so.

Run the simulation. Watch the colored boxes, and be sure they make sense to you. Try "kidnapping" the robot (dragging the simulated robot in the window) and see how well the belief state tracks the change. We recommend clicking the SoaR `stop` button, then dragging the robot, then clicking the `run` button. It makes it less likely that the robot will get stuck in some random place along the way.

Question 1. Explain the relationship between the two windows, and why they often start out similar and diverge over time.

Question 2. Why is it that, when you put the robot in a corner, all of the corners have high values in the $P(o|s)$ window?

Question 3. What happens when the robot is kidnapped?

Checkpoint: Tuesday 4:00 PM

- Find a staff member, and explain your answers to the previous set of questions.

The Code

Here is much of the code from `WanderEstBrain.py`, with an explanation of what's going on. It is similar, in high-level structure, to the planner brains we used before.

We start by telling this program where to look for its data files. You can do that by editing the `dataDirectory` line. The file `maxRange2.465/she20.dat` contains the ideal sensor readings at every x, y, θ pose on a $20 \times 20 \times 20$ grid, assuming that the walls of the *She World* in the simulator are fixed. It takes a long time to compute them, so it's better to do it off-line, and then just look them up when we're running the state update routine.

```
dataDirectory = "yourPathNameHere/ps13code/maxRange2.465/"
```

Next, we make an instance of the `GridStateEstimator` class, first specifying the size of the world, and then calling the initializer.

```
def setup():
    (xmin, xmax, ymin, ymax) = (0.0, 4.0, 0.0, 4.0)
    m = GridStateEstimator(Map(sheBoxes), xmin, xmax, ymin, ymax, 20,\
                           dataDirectory + "she20.dat",
                           numBestPoses = 5)
```

Now, we make two windows, one for displaying the belief state and one for displaying the perception probabilities. To display a belief state, we start by finding, for each grid value of x and y, the value of θ so that $P(x, y, \theta)$ is maximized. What does this mean? It's the orientation that would be most likely for the robot, if it were in that location. Now, we take all those values and draw the squares with the highest values in yellow, next highest in red, next highest in blue, and least high in gray. We also show the most likely pose (both the location and orientation) in green.

```
(xrange, yrange) = (xmax-xmin, ymax-ymin)
(wxmin, wxmax, wymin, wymax) = (xmin - 0.05*xrange, xmax + 0.05*xrange,
                                ymin - 0.05*yrange, ymax +
                                0.05*yrange)
beliefWindow = DrawingWindow(300, 300, wxmin, wxmax, wymin, wymax, \
                             "Belief")
percWindow = DrawingWindow(300, 300, wxmin, wxmax, wymin, wymax,"P(O | S)")
m.drawBelief(beliefWindow)
m.initPose(pose())
```

In all the previous labs, when we issued a motor command to the robot, it would continue moving with those velocities until it got the next command. In this lab, we're going to do it differently, because doing belief state update can sometimes take a long time to compute, and if we go for a long time without giving the robot a new command, it could run into the wall before we have a chance to give it a stop command. So, this time we are going to run the robot in *discrete motor mode*, where it moves for a tenth of a second at the commanded velocities and then stops until it gets another command.

```
def discreteMotor(trans, rot):
    discreteMotorOutput(trans, rot, 0.1)
```

Because the belief update is so expensive, we don't want to do it on every primitive step. But we're going to need to make a function that can be executed on every primitive step. So, here, we define a function `makeBeliefUpdateEveryN` that takes `n` as an argument, allocates a counter that will keep track of how long it has been since the last update. Then, we return a function that refers to that counter: it checks to see whether it has been `n` steps since the last update. If so, it resets the counter and updates the belief state based on the current sonar readings and the current pose and redraws the windows.

Why are we handing the current pose into the belief state update, when the robot doesn't really know where it is in the world? The answer is that the belief state update needs to know the action we just took, since we need to use the combination of a previous belief state, action and observation to update the probability of a state. We can interpret the action as 'whatever change in pose happened over the past N steps'. So, that is what we use the pose for: to compute the change in pose over the last N steps.

```
def makeBeliefUpdateEveryN(n):
    updateCount = [n]
    def beliefUpdateEveryN():
        if updateCount[0] == n:
            updateCount[0] = 0
            m.update(sonarDistances(), pose())
            m.drawObsP(percWindow)
            m.drawBelief(beliefWindow)
        updateCount[0] += 1
    return beliefUpdateEveryN
```

Finally, we make the driver. It is an instance of a new class, `TBParallelWithFun`, defined in our new `Sequence.py`, which takes a terminating behavior and a function at initialization time, and makes a new terminating behavior that does whatever the original TB did, but also calls the specified function on every step.

In this case, we do our old favorite *avoid and wander* behavior, in parallel with a function that updates the belief state every 10 steps.

```
    robot.driver = TBParallelWithFun(AvoidWanderTB(sonarDistances,
                                                   discreteMotor),
                                     makeBeliefUpdateEveryN(10))
    robot.driver.start()
```

Whew. That was all the initialization. Now, every time we're asked to do a step, we ask the driver to take a step.

```
def step():
    robot.driver.step()
```

Quality of Localization

We might want to know how well our localization method is working, in order to evaluate how changes in various aspects of our model affect localization performance. So, first, we have to decide what would be a good measure of how well the localization is working, and then implement a method that computes it. Luckily, in the simulator at least, we can find out the true pose; but we still have to think of a way to measure the quality of the current belief state, given the true pose.

In the simulator, you can "cheat" and find the true pose of the robot. Inside a brain, you can call `cheatPose()` to get the current true pose.

You can find the logarithm of the probability associated with a real-valued pose by doing

```
m.getBeliefAtPose(cheatPose())
```

Or, if you want to get the log probability associated with a set of grid indices, you can do

```
m.getBeliefAtIndices((2, 2, 20))
```

To convert these numbers to probability, use `math.exp`. If you want to use it, you'll have to put `import math` at the top of your file. You can also convert a set of grid indices into a pose using `m.indicesToPose`, or a pose into a set of grid indices using `m.poseToIndices`.

The robot's opinion of the grid square with the highest probability (i.e., the tuple `(ix, iy, ith)` with the highest probability of containing the robot) can be found with:

```
m.bestPoseIndices
```

Another method that you might find useful is

```
m.kBestPoses
```

It returns a tuple of the `k` most likely poses, in the format `((lp1, (x1, y1, th1)), (lp2, (x2, y2, th2)), ...)`, where `lp1` is the log of the probability of the robot being in the pose with *indices* `(x1, y1, th1)`. By default, it keeps the 5 best poses, but you can change this value by changing the argument in the initializer for `GridStateEstimator`.

As of Tuesday, our version of SoaR doesn't do logging correctly; so skip this for now In order to do repeatable experiments, it will be useful to gather data logs from simulated or real robot runs. To gather a log of the sensor data that the robot gathers as it's moving around, add `writeLog("robot.log")` to your brain. This will keep track of all the sensor and odometry readings that your robot got. Now, you can run your program again, but replace that line with

`readLog("robot.log")`; the robot won't move, and instead it will "hallucinate" the sensor readings that it had on the previous run whenever you call `sonarDistances()` or `pose()`. This allows you to sit quietly at your desk and print out values, or stare at the graphics windows to really understand what was going on, rather than trying to figure it out as you chase your robot around the lab. When you are replaying a log, you'll have to choose a simulated world, and the robot will run around, insensate, banging into things. Just don't watch.

Question 4. Think of a good way to measure how well the localization is working, and augment `WanderEstBrain.py` to compute that measure and print it out. Talk over your planned measure with your LA.

Question 5. Implement your measure and arrange it so that your brain prints it out every time the belief state is updated. You might want to print average value of your measure over time, as well. In order to compare localization performance carefully, we should try different methods on the same data. To do this, let the simulated robot drive around, while you are grabbing a log. Kidnap the robot a time or two when you do this (but try to remember where the simulated robot really was, and something about the kidnappings, or you'll have trouble making sense of subsequent runs.)

Question 6. Try changing the frequency with which the belief state is updated. How does that change the quality of the localization, as measured on the log file you grabbed above?

Question 7. Try using grids of different sizes. You can do this by changing `she20.dat` in `WanderEstBrain.py` to `she30.dat`. You'll also have to change the `20` in the call to `GridStateEstimator` to `30`, as appropriate. How does that affect the localization performance? Is your localization metric comparable when the grid size changes? If not, how could you change it so it would be?

Question 8. We've prepared two other environments for you. Try using `shannon20.dat` instead of `she20.dat`. You can do that by commenting out the two lines for `she` world, and uncommenting ones for `shannon`. You'll have to choose `shannon.py` in SoaR when you specify what simulator to use. Try using `playpen20.dat` in your program and `empty.py` in the simulator. Which of these worlds is easier for the robot to localize in? Why? Could you fix it with better sensors?

Checkpoint: Tuesday 5:00 PM

- Implementation of a metric.

Checkpoint: Thursday 3:00 PM

- Changing update frequency
- Changing granularity
- Other environments

On the robot

Remember to remove all calls to `cheatPose` *when you're running on the real robot. It doesn't know how to cheat!*

Using a robot and the available playpens (we'll have one corresponding to Shannon's World, which can be converted to Empty world by removing the box in the middle), gather some logs of sensor data as the robot moves around in the playpen. Take notes while the robot is moving, and draw a rough picture of the trajectory it followed. Now, replay the logs and watch the state estimation windows. Have the code print out the most likely state at each state estimation step. Do they correspond to the actual positions of the robot, according to your notes?

If you find the computational overhead of doing the state estimation is too annoying while gathering your robot logs, you could actually comment out the state-estimation code while running on the real robot; since this is a passive part of the program and doesn't affect the robot's actions, you'll get the same data if you just run the basic avoid and wander behavior. Of course, this is only true because we have set a switch in SoaR that makes the robot move a fixed amount of time and then stop on each step, rather than moving continuously until the next move command is received.

Sensor noise We picked the standard deviation used in the sensor model pretty arbitrarily to be 0.3. Inside the file `GridStateEstimator.py` you will find a statement

```
sigma = 0.3
```

How should we set this value? There are actually two sources of variability that are being accounted for, here. The first is the usual one of how much variability there is in a sonar reading given an actual distance to the wall. In the simulator, there is currently no noise on the sensor readings, so we aren't getting much variability from that source; but of course there is more of this kind of noise on the real robot sensors. The other one is due to the discretization: we are treating all the poses in one cube of the pose space as if they were the same, and there may be significant variability of the sensor readings from the different poses in that cube.

The right strategy for setting the value would be to sample a bunch of sensor readings, and then do some statistical analysis to estimate the mean and variance. Instead of doing that, you might try changing the variance in the model and seeing what effect that has on the quality of the estimates you get (in simulation, of course).

Question 9. Does changing the sensor noise model make the estimation work better? How did you change it?

Question 10. Is $\texttt{sigma} = 0.3$ reasonable for the grid quares we're using? What would we need to do to our model if there were no bubble wrap?

Checkpoint: Thursday 4:00 PM

- Tests on the robot, with different sensor noise model.

Using the planner

Switch to using `XYEstBrain.py`. Instead of running the avoid and wander behavior, it runs the planning system from lab 11. But how does it know its current pose? Of course, it doesn't really; but it takes the most likely pose out of the state estimator, assumes it's the correct pose, makes a plan, executes the first step, and then re-estimates the pose using our state estimator.

Question 11. Run it (in simulation) three or four times in She world. What happens?

Question 12. Try it on the robot. Does it seem to behave differently there?

Question 13. If there are cases when the robot does not reach the desired goal (either in simulation or on the real robot), describe one, explain why you think it happened, and describe a strategy for fixing it (but you don't have to actually do it, unless you have spare time).

Checkpoint: Thursday 4:45 PM

- Answers to questions above.

No Written Post-Lab Due

Concepts covered in this assignment

Here are the important points covered in this assignment:

- Robot localization can be done robustly, even with noisy sensors and effectors
- A variety of modeling tricks, including discretization and independence assumptions, are necessary to make the system practical
- Making formal models of real problems is difficult and important
- Programming practice