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 (x, 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 (x, y, θ^*). 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,

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

for each square x, 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 $x, y, 0$ 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 0 so that $P(x, y, 0)$ 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.