MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.01—Introduction to EECS I
Spring Semester, 2008

Design Lab 13, Issued Thursday, May 8

Do

```
athrun 6.01 update
```

to get the `~/Desktop/6.01/lab13/designLab` folder.

Note that problems 5 and 6 correspond to problems 14 and 15 from last week's design lab.

Where am I?

In this section of the lab, we will explore the robot's ability to figure out where it is in a world, if it knows the map of the world, but not where it is to start with.

We'll use the brain in `HGEstBrain.py`. The file `boxWorldSmall20.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 `boxWorldSmall` 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. You can tell the program where to look for its data files by editing the line below, replacing `yourPathNameHere` with your path name.

```
dataDirectory = "yourPathNameHere/Desktop/6.01/lab13/designLab/"
```

The `setup` method of the brain does a lot of work to initialize the state estimator and draw windows, and then creates a sequential machine with some instances of the `IPE` class. The `IPE` class is exactly like the one we used in lab 11, except that each time a macro action finishes executing, it does a state update based on the current sonar readings and the change in odometry since the last update. Then, instead of calling `cheatPose` to determine where it is located after each macro action, it asks the state estimator for the most likely state, and uses that as the starting state for planning.

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'll change this approach, because a 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.

A simulacrum of the real thing

Now, start up SoaR in simulation, select the `boxWorldSmall.py` from the `lab13/designLab` directory as the world, and `HGEstBrain.py` from the same directory as the brain. When it starts up, you'll see two new windows.

One window, labeled `Belief`, shows the outline of the obstacles in the world, and a grid of colored squares. Squares that are colored gray represent locations that partially or completely blocked by obstacles. For the purposes of this window, for each (x, y) location, we sum the probabilities over all the θ values for each location, then we draw a color that's related to the probability (bright red is most unlikely, bright green is most likely, black is in between). At the absolutely most likely pose (x, y, θ), the robot is drawn, with a nose, in gold.

The other 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 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)**, because it's possible, for example, to get a sonar reading that is unlikely to have been made at any location; we've scaled the colors to make them sort of similar to the colors in the belief window, but they aren't directly comparable.

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 sometimes 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: Change the code so that after the robot reaches the goal it does a dance, and then drives to `goalPose2`. (Note that we have changed `IPE` to accept an additional argument that specifies that the location of the goal needs to be reached but does not require a particular orientation.)

Question 4: Run this system on the robot in a pen that is configured to be the same shape as `boxWorldSmall`. We have roughly half as many robot pens as pairs of students in the biggest lab section, so you may have to share with one other group. Be sure you understand where the origin is, and how the box should be placed with respect to it.

Checkpoint 1: 60 minutes

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

A scanner brightly

These problems will build on your `lightBrain.py` *from lab 12.*

When the robot gets to one of the locations where a light could be, it should execute a behavior that scans for a light. If a light is detected, the robot should track the light until the desired brightness is reached. The robot should stop but the behavior should not terminate. If no light

is detected after a full scan, the behavior should terminate (so that subsequent behaviors can take over).

One simple possibility for scanning is to have the robot rotate in place, testing for a bright enough light; we can do this using one or more calls to `RotateTSM`. Another possibility is to scan the head (by turning through a series of intermediate positions), but because the head can turn a total of less than 180 degrees, we could miss the light if it is behind the robot. A combination of body and head scanning is also possible.

In constructing your scanning behavior, you should use the TSM combination classes, defined in `TSMFull.py`. In particular:

- `SequentialTSM(list_of_TSMs)` - takes a list of TSMs and executes them in order until each is `done`.
- `IfTSM(condition, sm1, sm2)` - takes a condition (a function) that is given the inputs and returns a boolean and executes `sm1` or `sm2` depending on whether the returned value is `True` or `False`. Note that this test happens at every step of the resulting machine, so the `IfTSM` machine will step one behavior or the other depending on the value of the condition. This machine is `done` when the currently selected machine is `done`.
- `RepeatTSM(sm, n)` - will execute `sm` until it is `done` a total of `n` times. If `n` is not given, it will execute it forever.

Each of these combinations produces a TSM so they can be combined arbitrarily.

We suggest a debugging strategy where you start, first, by running the robot in the simulator, but with the head hooked up to the laptop via the Nidaq. You can shine light on the robot's eyes and it should respond in the simulator.

Question 5: Define the top-level behavior in `lightBrain.py` (`robot.behavior`) to scan for a light and either terminate when there is no light or track (and not terminate) when there is one.

Question 6: Extend the behavior so that it moves in a square and scans for lights at each of the corners of the square. You might find `ForwardTSM` useful.

Checkpoint 2: 120 minutes

- Demonstrate your robot scanning for a light over a square and show two outcomes: (a) giving up because it does not find a light and (b) finding a light, tracking it and stopping when close enough.

Finding Light, with a Clue

Now, assume that a lamp is located at either `goalPose1` or `goalPose2`. Your job is to build a controller for the robot that localizes itself and drives to one goal location, rotates around to try to find the light, and if the light is bright enough, drives up to the light and stops. If it does not find

the light at the first goal location, it should proceed as directly as possible to the second location, and look for the light there.

In order to extend `HGEstBrain.py`, you will need to change `False` to `True` in the line below, in order to be able to read from and write to the nidaq box.

```
initIO(sonarDistances, pose,
       lambda f, r: discreteMotorOutput(f, r, stepLength),
       loadNidaq = False)   # Change to true to use nidaq
```

Then, you'll need to integrate in the appropriate parts of your head control code from the previous section of this lab.

Question 7: Define a sequential machine that will achieve this task. Implement it and verify that it works.

Checkpoint 3: End of Lab

- Demonstrate your robot planning and executing systematic motions between goals, searching effectively for the light at each location, and driving up to the light if it finds it.

Finding Light, without a Clue

If you've finished early and are looking for more of a challenge, you might try one or more of these things:

- Systematically explore the world, making sure that you visit every location, and look around for light at each location, tracking it if you find it, as before.
- Try your system in `boxWorld.py` (which is the size of two small worlds).
  Or, even, make your own new world (this requires defining a world file for soar, adding definitions of the obstacle locations to Python for the mapping function (in `GridMap.py`) and generating new stored sonar readings. To make a new `.dat` file, for a new set of boxes defining obstacles in your world, you need to load the file `WriteIdealReadings.py` into Idle, and then run a command of this form:

  ```
  writeReadings(Map(myNewBoxes),
                "/Users/lpk/lpkmac/C1/ps13/ps13code/distAngle/newBoxes20.dat",
                0.0, 3.05, 0.0, 3.05, 20)
  ```

  With your path name, of course. The last numbers are the dimensions of the world (it needs to be square) and the number of grid squares in each dimension (same for all dimensions).