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

Work for Week 12

- Software lab for Tuesday, May 1
- Pre-lab tutor problems due Thursday, May 3
- Robot lab for Thursday, May 3
- Post-lab written problems due Tuesday, May 8

State estimation

In the next two labs, we'll use basic probabilistic modeling to build a system that estimates the robot's pose, based on noisy sonar and odometry readings. We'll start by building up your intuition for these ideas in a simple simulated world, then next week we'll move on to using the real robots.

There will be two major parts of the work for this week: working with the grid-world simulator and writing your own state-estimation code. The plan is this:

Tuesday: Work with your partner to start the grid-world simulation lab

Thursday: Work with your partner to finish the grid-world simulation lab; when that's done,

Thursday: Work on the state-estimation code on your own.

Grid World Simulator

Download the code for this lab, in `ps12.zip` from the calendar page. In it, you will find `StateEstimationNoisyNN.py` and `StateEstLab.py`. *Don't use the Idle Python Shell for this lab. You can use the Idle editor, but you cannot run the code from inside Idle.* If you are a SoaR afficionado, you can do this all from within SoaR. Start SoaR, open an editor window onto `StateEstLab.py`, choose **Run in Interpreter** from the **SoaR** menu, and give commands by typing in the buffer at the bottom of the command window. If not, you can go to a terminal window and type `python`. You can type commands to Python here.

```
> python
Python 2.4.4 (#1, Oct 18 2006, 10:34:39)
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Open `StateEstLab.py` in Idle or your favorite other editor. You'll see a command at the end of that file, that says:

```
w51pp = World(5, 1, (), (), ((2,0),), (), (0,0), perfectSensorModel, \
              perfectMotionModel)
```

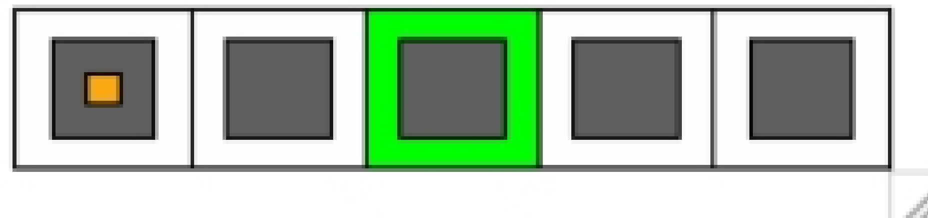
Go to your terminal window, and type

```
>>> from StateEstLab import *
```

As the lab goes along, if you edit `StateEstLab.py`, then you'll need to go back to this window and type

```
>>> reload(StateEstLab)
>>> from StateEstLab import *
```

When you evaluate it, you should see a window that looks like this:



This is a world with 5 possible "states", each of which is represented as a square with a colored border. The possible colors of the borders are white, black, red, green, and blue. In this example, four squares are white and one is green. There is a small orange rectangle representing the square that our simulated robot is actually occupying. The inner part of each square is gray; as we'll discuss in detail later, how light the square is represents how likely the robot thinks it is that it's in that square.

The arguments to the `World` initialization function are:

- The dimension of the world in x
- The dimension of the world in y
- Four tuples of coordinates, each specifying the location of colored squares. The first list gives the locations of black squares, the second red, the third green, the last blue. Squares unspecified in any of those lists are white.[1]
- A pair of indices specifying the robot's initial location
- A model of how the sensors work
- A model of how the actions work

So, this world is 5-by-1, with one green square, the robot initially at location (0,0), and perfect sensor and motion models.

You can issue commands to the robot by typing, if `w` is a world,

```
w.north()
w.south()
w.east()
w.west()
w.stay()
```

> **Question 1.** Just to get the idea of how this works, move the robot east and west a few times, and just take a look at (but don't be worried if you don't understand) what is written on the screen. It shows the actual numeric values associated with the robot's belief that it is in each of the squares.

This model is a slight variation on a hidden Markov models (HMM) that we saw during lecture. The only difference is that the robot can select different actions (like trying to move north or trying to move south), and those different actions will cause different state-transition distributions.

[1] The expression `((2,0),)` makes a tuple that contains a single element, which is itself a tuple that contains 2 and 0. The reason for the extra comma character is so that the outer parentheses are treated as a tuple constructor, not just as grouping parentheses.

The sensor model, generally speaking, is supposed to specify a probability distribution over what the robot sees given what state it is in: $P(O_t = o_t | S_t = s_t)$. In our case o_t ranges over *white*, *black*, *red*, *green*, and *blue*, and s_t ranges over all the possible states of the robot (the different grid squares).

We have made a simplifying assumption, which is that the robot's observation only depends on the color of the square it's standing in; that is, all white squares have the same distribution over possible observations. So, we really only need to specify $P(\mathit{observedColor}|\mathit{actualColor})$, and then we can find the probability of observing each color in each state, just by knowing the actual color of each state:

$$P(O_t = \mathit{observedColor}|S_t = s_t) = P(O_t = \mathit{observedColor}|\mathit{actualColor}(s_t)) \enspace .$$

In our program, this set of distributions is specified as a tuple of tuples, where each row corresponds to a different actual color. So, for example `m[actual][observed]` would give the probability of observing `observed` in a square that was really colored `actual`.

Here's the model for perfect observations, with no probability of error:

```
perfectSensorModel = ((1.0, 0.0, 0.0, 0.0, 0.0),
                      (0.0, 1.0, 0.0, 0.0, 0.0),
                      (0.0, 0.0, 1.0, 0.0, 0.0),
                      (0.0, 0.0, 0.0, 1.0, 0.0),
                      (0.0, 0.0, 0.0, 0.0, 1.0))
```

Question 2. Give a sensor model in which red and blue are indistinguishable (that is, the robot is just as likely to see red as to see blue when it is in a red square or a blue square).

Question 3. Give a sensor model in which red always looks blue, and blue always looks red.

Question 4. Which one of these models is more informative?

Question 5. Do the rows always have to sum to 1? The columns? Why?

Question 6. Given an example situation in which our assumption (that all squares of a given color have the same error model) is unwarranted.

In a real-world system we probably wouldn't ever want to have zeros anywhere in the sensor model: it is important to concede the possibility of error.

The state-transition model specifies a probability distribution over the state at time $t+1$, given the state at time t and the selected action a. That is, $P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t)$. This is a slight generalization of the HMM model we saw in class, since it assumes that there is an "agent" in the world that is choosing actions; the effects of the action are modeled by essentially selecting a different transition model depending on the action. So, the next state of the system depends both on where it was before and the action that was taken. If you were to write this model out as a matrix, it would be very big: mn^2, where n is the number of states of the world and m is the number of actions.

Often, the transition model can be described more sparsely or systematically. In this particular world, the robot can try to move north, south, east, or west, or to stay in its current location. We'll