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

Assignment 12, Issued: Tuesday, Nov. 20

To do this week

...in Tuesday software lab

1. Start writing code and test cases for the numbered questions in the software lab. Paste all your code, including your test cases, into the box provided in the "Software Lab" problem on the on-line Tutor. What you submit to the tutor will not be graded; what you hand in next Tuesday will.

...before the start of lecture next Tuesday

1. Do the lab writeup, providing written answers (including code and test cases) for **every** numbered question in this handout.

On Athena machines make sure you do:
`athrun 6.01 update`
so that you can get the `Desktop/6.01/lab12` directory which has the files mentioned in this handout. You need the whole lab12 distribution for the software lab; use one of our laptops for the software lab.

State estimation

In the next two software and two robot 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 we'll move on to using the real robots.

This week we'll start by working with a non-deterministic grid-world simulator. You should work with your partner on this.

Grid World Simulator

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.

In the `lab12` folder you will find
`StateEstSM.py`.
Open this file in Emacs, Idle or whatever editor you like. `If you use Idle as an editor, do not try to evaluate the python commands in the Idle Python Shell.`

Then, open a Terminal window (if you're on Athena, remember to do `add -f 6.01`), then connect to the lab12 director

```
cd ~/Desktop/6.01/lab12
```

and type `python`.

```
> python
Python 2.5 (r25:51918, Sep 19 2006, 08:49:13)
[GCC 4.0.1 (Apple Computer, Inc. build 5341)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

You can type commands to Python here. In this Python, type

```
>>> from StateEstSM import *
```

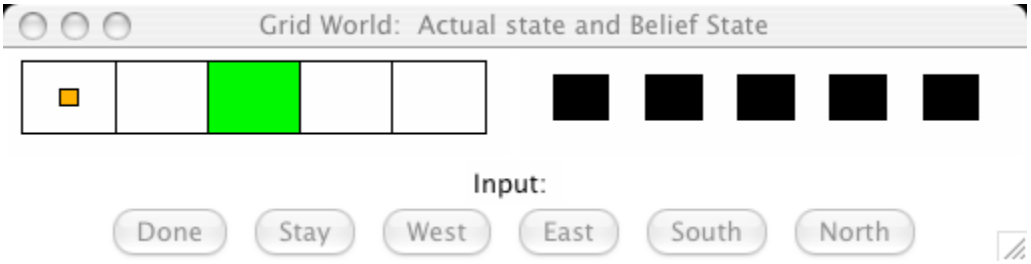
As the lab goes along, if you edit `StateEstSM.py`, then you'll need to go back to this window and type

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

You'll see a command at the end of the `StateEstSM.py` file, that says:

```
tw = makeGridSim(5, 1, [[],[],[[2,0]],[]], [0,0], perfectSensorModel, \
                 perfectMotionModel)
```

Execute this command in Python (on your Terminal window). 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 colored square (on the left). The possible colors of the states 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. On the right are five squares that start out being black; the color in those squares represents how likely the robot thinks it is that it's in that square. This is called the *belief state.* The colors illustrating the belief state are: black, when the value is what the uniform distribution would assign (0.2 for 5 states), shades of green when the probability is higher than the uniform, and shades of red when it is below the uniform value.

The arguments to the `makeGridSim` function are:

- The dimension of the world in x
- The dimension of the world in y
- Four lists 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.
- 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 grid 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:

```
tw.run()
```

and clicking on the buttons in the simulator window. The buttons correspond to the five available actions: Stay, West, East, South and North. Clicking on the Done button returns control to Python. The window will remain after you click Done. **Do not kill the window until you are completely done with it.** You can type:

```
tw.reset()
tw.run()
```

to start interacting with the window again after you've typed Done.

Question 1. Just to get the idea of how this works, move the robot east and west a few times, and write down 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. You should be able to relate this, at least roughly, to the HMM example in the notes.

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.

The sensor model, generally speaking, is supposed to specify a probability distribution over what the robot sees given what state it is in: $\mathsf{P}(\mathsf{O}_t = \mathsf{o}_t | \mathsf{S}_t = \mathsf{s}_t)$. 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 $\mathsf{P}(\mathit{observedColor}|\mathit{actualColor})$, and then we can find the probability of observing each color in each state, just by knowing the actual color of each state:

$$\mathsf{P}(\mathsf{O}_t = \mathit{observedColor}|\mathsf{S}_t = \mathsf{s}_t) = \mathsf{P}(\mathsf{O}_t = \mathit{observedColor}|\mathit{actualColor}(\mathsf{s}_t)) \ .$$

In our program, we specify the observation model by a function which takes as input the `trueColor` and returns an instance of the `Dist` class. The `Dist` class, which we saw earlier in the term, specifies a probability distribution over a set of values as a list of `[value, probability]` pairs. The probabilities must sum to one. Any possible value that is not mentioned in the distribution is assumed to have probability equal to 0.0.

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

```
def perfectSensorModel(trueColor):
    return Dist([[trueColor, 1.0]])
```

Note that, since there is no error, there is only one entry in the distribution, whose probability is 1.0.

The input to the sensor model function is an integer denoting the color. The mapping between integers to color are the following dictionaries defined in `StateEstSm.py`:

```
colors = {0: 'white', 1: 'black', 2: 'red', 3:'green', 4:'blue'}
inverseColors = {'white':0, 'black':1, 'red':2, 'green':3, 'blue':4}
```

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. Modify the test world `tw` so that it has some blue and red squares. Try out your sensor models above and see what happens to the belief state. Which one of these models is more informative? Justify your answer based on your experiments.

Question 5. Define a sensor model (call it `noisySensorModel`) in which the `trueColor` has probability of 0.8 and there is a probability of 0.2 of seeing one of the remaining colors (equally likely which other color you see).

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 assume that the kinds of errors the robot makes when it tries to move in a given direction don't depend on where the robot actually is (except if it is at the edge of the world), and we'll further assume that the transition probabilities to most next states are zero (there's no chance of the robot teleporting to the other side of the room, for example).

In this simulator, a motion model is a function that takes the robot's current x, y coordinates, an action, which is specified as a set of x, y offsets (so the action to move north would be $0, 1$, for example), and the dimensions of the world. It returns an instance of `Dist` that specifies a probability distribution over the possible next states, which is specified by lists of pairs of robot locations and the probability of moving to that state.

Here is a motion model with no noise. There is only one possible resulting state, in which coordinates of the action are added to those of the robot, and then the result is clipped to be sure it stays within the confines of the world.

```
def perfectMotionModel((rx, ry), (ax, ay), (xmax, ymax)):
    def cx(a): return clip(a, 0, xmax-1)
    def cy(a): return clip(a, 0, ymax-1)
    return Dist([[[cx(rx+ax),cy(ry+ay)], 1.0]])
```

Note that this is written for the general case of a two-dimensional grid even though our example right now is one dimensional.

Question 7. Write a motion model that makes the world a torus (that is, a donut, where there's no "edge" of the world, it just wraps around to the other side), but where motion is deterministic. Test it out by moving the robot around. Make sure that you do this so that it works on two-dimensional worlds.

Question 8. Write the "east wind" motion model, in which, with probability 0.1, the robot always lands one square to the **west** of where it should have. You can do this in the original model, or add it to the torus model. Test it out by moving the robot around. Make sure that you do this so that it works on two-dimensional worlds.

Question 9. Define your own noisy two-dimensional motion model, implement it and test it. Make sure you also provide an English description of the behavior.

Question 10. Try out your models on the 5 x 5 model, `s55`, defined in `StateEstSM.py`.

Go to the on-line Tutor at `http://sicp.csail.mit.edu/6.01/fall07`, choose PS12, and paste the code that you wrote during lab, including your test cases, into the box provided in the "Software Lab" problem. Do this even if you have not finished everything. Your completed answers to these questions are to be handed in with the rest of your writeup for Tuesday, November 27.
What to turn in: For each of these questions, include any code you wrote, as well as demonstrations on a set of test cases that you think demonstrates your code is correct. You will be graded on your selection of test cases, as well as on the correctness of your code.

Concepts covered in this assignment

Here are the important points covered in this assignment:

- Uncertainty is everywhere! And probability is a good way to model it.