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)
```

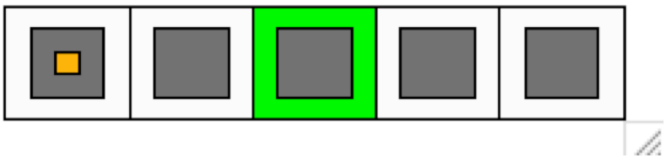
Go to your terminal window, and type

```
>>> 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:

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

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

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 a tuple of pairs, each of which is a possible next state (robot coordinates), 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):
    return (((clip(rx+ax,0,xmax-1),clip(ry+ay,0,ymax-1)), 1.0),)
```

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.

Question 8. Write the "east wind" motion model, in which, with probability 0.1, the robot always lands one square to the east 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.

Checkpoint: Tuesday 5:00 PM

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

Belief state update

The robot's *belief state* is a probability distribution over possible states of the world. In this world, there is one underlying world state for each square; each of these states has a probability value assigned to it, and these values sum to 1. In the simulator, the current belief state is shown by the gray squares, with darker values closer to zero. It is also printed out whenever you move; in fact it is printed twice: once after the transition update and again after the sensing update, as explained below. (If you get tired of seeing the printout, you can always type `w.verbose = False` to shut it up.)

Just as in the HMM, the belief state is updated in two steps, based on the state transition model and the observation model. Study the state update rules in the notes, and convince yourself that at the end of this, all of the entries in the belief state will sum to 1.

Practice

This stuff is hard to build up an intuition for. Let's start by practicing in a world without noise. In our set-up, `w51pp` is just such a perfect world. Either make a new instance of it or do `w51pp.reset()`, which will set the belief state to $(0.2, 0.2, 0.2, 0.2, 0.2)$. This is often called a *uniform* distribution. The robot has no idea where it is, and considers all states equally likely.

Calculate an answer to these questions before trying it in the simulator. If you get a different result than you expected, convince yourself either that there is a bug in the simulator or why it's right.

Question 9. First, what is the robot's prior belief $\mathtt{b}(\mathtt{s_i})$ for each of the states?

Question 10. If we were to tell the robot to go east, what would the belief state be after taking the state transition (but not the observation) into account? Start by computing $\mathsf{P}(\mathtt{s_i}|\mathtt{s_j}, \mathit{east})$ for all $\mathtt{i}, \mathtt{j}$. Now, use this to compute $\mathtt{b}_{new}(\mathtt{s_i})$ for all $\mathtt{i}$.

Question 11. Now, the robot will see "white" because it's moving to a white square and there's no noise. What will the belief state be after this?

- First, for each state, figure out $\mathsf{P}(\mathit{white}|\mathtt{s_j})$ for all states.
- Then compute $\mathsf{P}(\mathit{white}|\mathtt{s_j})\mathsf{P}(\mathtt{s_j})$ for each state. Remember that at this point the $\mathsf{P}(\mathtt{s_j})$ we are using is the belief state that resulted from the transition, not the original prior.
- Now, compute $\mathsf{P}(\mathit{white})$.
- Finally, compute $\mathsf{P}(\mathtt{s_i}|\mathit{white})$.

Question 12. Which action could the robot take at this point to make its location completely unambiguous?

Question 13. Now, make `w201pp` (un-comment it from the file, or create it from the command line). Try driving the robot toward the east. Don't solve the problem numerically, but be sure you can explain to yourself why the belief state is behaving the way it does.

Checkpoint: Thursday 2:45 PM

- Explain your answers to the previous sets of questions.

Noisy Practice

Now, let's try adding in sensor noise. We'll use `w51ns` (un-comment it from the file, or create it from the command line), which still has perfect motion but noisy sensors. Again, try to do these computations before running the simulation, using the schematic shown in the notes.

Question 14. If we were to tell the robot to go east, what would the belief state be after taking the state transition (but not the observation) into account?

Question 15. Now, what's the distribution over what the robot sees? That is, what is $\mathsf{P}(\mathsf{O}_1 = \mathtt{o}_1|\mathsf{A}_1 = \mathit{east})$, for all values of $\mathtt{o}_1$?

Question 16. Compute the next belief state if the robot sees "white". That is, what is $\mathsf{P}(\mathsf{S}_1 = \mathtt{s}_1|\mathsf{O}_1 = \mathit{white}, \mathsf{A}_1 = \mathit{east})$, for all values of $\mathtt{s}_1$?

Question 17. Compute the next belief state if the robot sees "green".

Question 18. Compute the next belief state if the robot sees "red".

Question 19. Go east again. (That's twice now). Explain why, in this particular run, in your simulation, you got the result you got.

Question 20. What (approximately) would happen if you did the `stay` action 10 times?

Now, let's try noisy actions only, using `w51na`.

Question 21. What is the belief state after telling the robot to go east, but before it gets the observation?

Question 22. What is the next belief state if it observes "white"?

Question 23. Move the robot to the green square. What is the belief state now?

Question 24. Can it ever get confused after moving some more? Why or why not?

It's very important to note that, because this is a simulated world, the exact same noise distributions are being used to generate the state transitions and observations as are being used to update the belief state. Of course, in the real world, we can never know the real world's transition and observation models exactly, so we have to estimate them. More about this next week.

Checkpoint: Thursday, 3:30 PM

- Explain your answers to the previous sets of questions.

Exploration: Try making bigger worlds with different patterns of colors and different error models. See if it behaves the way you expect.

Is there any way to make an environment that has a fundamental ambiguity that can never be resolved, even with perfect sensing and action?

State estimator

Implement a completely generic state estimator as a Python class. The initialization function should take as arguments:

- An initial state distribution, which is a function that consumes a state s and returns $P(S_0 = s)$.
- A transition model, which is a function that consumes a state s_t, an action a, and another state s_{t+1}, and returns $P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t)$
- An observation model, which is a function that consumes a state s_t and an observation o_t, and returns $P(O_t = o_t | S_t = s_t)$.

Your class should have methods that:

- update the belief state based on an action
- update the belief state based on an observation
- print out the belief state
- take a list of actions, $a_0, \ldots, a_t$, and observations, $o_1, \ldots, o_t$, and return

$$P(S_t = s_t | A_0 = a_0, \ldots, A_t = a_t, O_1 = o_1, \ldots, O_t = o_t)\ ;.$$

Debug your code by working through the first two steps of the copy-machine diagnosis problem from class. Include a printout showing that it works.

Extending the copy-machine model so that there are two actions, *print* and *maintain*. Assume that the model we specified was for the *print* action. Provide a model for the *maintain* action, which will tend to improve the state of the printer. Make up a couple of interesting action and observation sequences and show the belief state that results afterwards. Argue whether it's reasonable.

Post Lab

State-estimation Code

Finish up your state-estimation code if you didn't have a chance to do that already.

Diagnosing Pneumonia

Systems that estimate the hidden state of a process are used in a wide variety of applications. One interesting one is in the management of ventilators (active breathing systems) for intensive-care patients. One aspect of that problem is that such patients are susceptible to pneumonia, but it is difficult to diagnose based on a single temporal snapshot of the patient's vital signs, blood gasses, etc. Of course, the real process, and the models used, are very complicated. But let's consider a wildly oversimplified version below.[2]

Let the possible states of the patient be: **free** of bacteria, **colonized** by bacteria, or **pneumonia** from bacteria (includes being colonized). In this model, there are no actions; it is used just for diagnosis, but not planning. And let the observations (symptoms) be described by the following observation variables:

- abnormal temperature (yes, no)
- abnormal blood gas (yes, no)
- bacteria in sputum (yes, no)

(There are a huge number of other variables that could be relevant: many other signs and symptoms, age and general health of the patient, which particular hospital they're in, how long they've been on a ventilator, what drugs they are on, why they're in the ICU, etc. A big part of building such a model is deciding which of these variables are likely to be important).

[2]This problem was inspired by the paper "A Dynamic Bayesian Network for Diagnosing Ventilator-Associated Pneumonia in ICU Patients," by Charitos, van der Gaag, Visscher, Schurink, and Lucas, 2005; but the authors should in no way be held responsible for the ridiculous simplifications we've made in the following.

Question 25. Invent a state transition model (we know you probably don't know anything about medicine; just write down in English what your assumptions are, and then provide numbers that are consistent with your explanation.)

Question 26. How many possible observations are there in this domain?

Question 27. Invent a sensor model. Both pneumonia and simply being on a ventilator increase the likelihood of having abnormal blood gas readings. Pneumonia increases the likelihood of abnormal temp. Having bacteria in the sputum is a very strong indication of colonization (but there's always a chance the test sample was contaminated, for example.)

Question 28. Use your state estimator from the previous section to do state estimation in this model. Start with a belief state in which it's certain that the patient is **free** of bacteria. Try it with a sequence of observations that you think might be reflective of an actual infection. Try it again with a sequence of observations that has some abnormal readings, but which are probably not reflective of an actual infection. Show printouts of the state updates at each step. Explain why they go the way they do.

Question 29. In fact, this model was constructed so that ICU personnel could decide whether to administer antibiotic therapy and, if so, against which particular organisms (each of which will have a different dynamics and observation model). Speculate a little bit on how having a probabilistic model of the underlying state of a patient could help decide upon a therapy.

What to turn in

- Answers to all lab questions
- Generic state-estimation code, nicely documented, with test cases
- Pneumonia model and sample run using state-estimation 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.
- Even with noisy effectors and weak sensors, it's often possible to diagnose the underlying state of a system
- Programming practice