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

Assignment 13, Issued: Tuesday, Nov. 27

To do this week

...in Tuesday software lab

1. Start answering the numbered questions (1–21) in the software lab. These mostly require you to derive results using the state estimation algorithm from lecture.

...before the start of lab on Thursday

1. Re-read the Week 12 lecture notes.
2. Read through the entire description of Thursday's lab.

...in Thursday robot lab

1. Do the nanoquiz at the start of the lab. It will be based on the material in the lecture notes on state estimation and the software lab exercises from last week and this week.
2. Answer the numbered questions in the robot lab and demonstrate the appropriate ones to your LA.

...before the start of lecture next Tuesday

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

On Athena machines make sure you do:
`athrun 6.01 update`
directory which has the files for the software lab.
- You need the new file `StateEstSMterm.py` in the `lab12` folder for the software lab.

For Thursday lab, you will also need the files in `Desktop/6.01/lab13`.

More on State Estimation

In this lab we'll continue our study of state estimation; 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.

Tuesday Software Lab

In this week's software lab we'll continue working with the non-deterministic grid-world simulator. You should work with your partner on this. We repeat the instructions from the previous lab here.

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.
Note that there is a new version of the `StateEstSM.py` file called `StateEstSMterm.py` which supports terminal input instead of buttons, in case you run into problems with the buttons. It also has some additions that you'll need for the software lab.

In the `lab12` folder you will find
`StateEstSMterm.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 folder

```
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

```
>>> import StateEstSMterm
>>> from StateEstSMterm import *
```

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

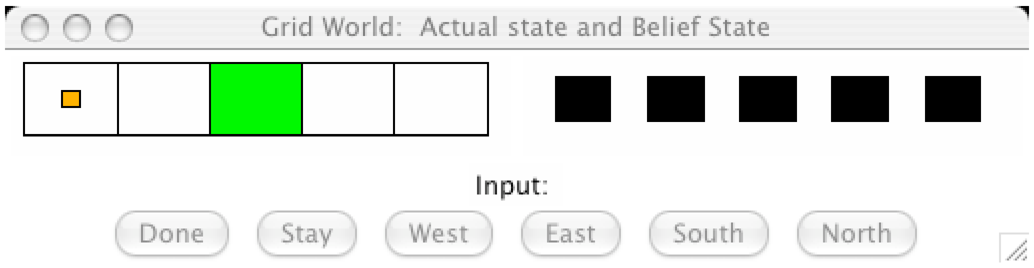
```
>>> reload(StateEstSMterm)p
>>> from StateEstSMterm import *
```

If you have trouble with the window buttons, use `makeGridSimNoButtons` instead of `makeGridSim`. The interaction is done by typing text in the Python window instead of clicking buttons on the display window.

You'll see a command at the end of the `StateEstSMterm.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.

If you have trouble with the window buttons, use `makeGridSimNoButtons` instead of `makeGridSim`. The interaction is done by typing text in the Python window instead of clicking buttons on the display window.

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.

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 squares on the right, with redder values closer to zero and greener values closer to one. 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 `tw.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. We'll ask you to work through the state estimation equations for some simple examples, so you can come to understand in detail how things work.

Let's start by practicing in a world without noise. In our set-up, `tw` is just such a perfect world. Either make a new instance of it or do `tw.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 by hand 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. Show your computation in detail when you hand in answers.

Question 1. First, what is the robot's prior belief $\mathtt{b}(\mathtt{s_i})$ for each of the states?

Question 2. When you do `tw.run()`, it automatically does a `stay` action. What is the belief state be after taking the state transition (but not the observation) into account? Start by computing $\mathtt{P}(\mathtt{s_i}|\mathtt{s_j}, \mathit{stay})$ for all $\mathtt{i}, \mathtt{j}$. Now, use this to compute $\mathtt{b}_{new}(\mathtt{s_i})$ for all $\mathtt{i}$.

Question 3. Now, the robot will see "white" because it's in a white square and there's no noise. What will the belief state be after this?

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

Question 4. 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 5. Now, the robot will see "white" again because it's moving to a white square and there's no noise. What will the belief state be after this?

Question 6. Which action could the robot take at this point to make its location completely unambiguous?

Question 7. Now, make `s201pp` (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 why the belief state is behaving the way it does.

Noisy Practice

Now, let's try adding in sensor noise (look at the `noisySensorModel` in `StateEstSMterm.py`). We'll use `s51ns` (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 8. After the initial stay action, but before the observation, what would the belief state be?

Question 9. Now, what's the distribution over what the robot sees? That is, what is $\mathsf{P}(\mathsf{O}_1 = \mathsf{o}_1|\mathsf{A}_1 = \textit{stay})$, for all values of o_1?

Question 10. Compute the next belief state if the robot sees "white". That is, what is $\mathsf{P}(\mathsf{S}_1 = \mathsf{s}_1|\mathsf{O}_1 = \textit{white}, \mathsf{A}_1 = \textit{stay})$, for all values of s_1?

Question 11. 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 12. Now, what's the distribution over what the robot sees? That is, what is $\mathsf{P}(\mathsf{O}_2 = \mathsf{o}_2|\mathsf{A}_2 = \textit{east})$, for all values of o_2?

Question 13. Compute the next belief state if the robot sees "white". That is, what is $\mathsf{P}(\mathsf{S}_2 = \mathsf{s}_2|\mathsf{O}_2 = \textit{white}, \mathsf{A}_2 = \textit{east})$, for all values of s_2?

Question 14. Compute the next belief state if the robot sees "green." That is, what is $\mathsf{P}(\mathsf{S}_2 = \mathsf{s}_2|\mathsf{O}_2 = \textit{green}, \mathsf{A}_2 = \textit{east})$, for all values of s_2?

Question 15. Compute the next belief state if the robot sees "red." That is, what is $\mathsf{P}(\mathsf{S}_2 = \mathsf{s}_2|\mathsf{O}_2 = \textit{red}, \mathsf{A}_2 = \textit{east})$, for all values of s_2?

Question 16. Go east again. (That's twice now). Explain why, in this particular run, in your simulation, you got the result you got.

Question 17. What (approximately) would happen if you then did the `stay` action 10 times? Explain.

Now, let's try noisy actions only, using `s51na`, starting from the initial state. Show your computation in detail when you hand in answers.

Question 18. When you do `s51na.run()`, it automatically does a `stay` action. What is the belief state after the stay action, but before it gets the observation?

Question 19. What is the next belief state if it observes "white"?

Question 20. Move the robot to the green square. What is the belief state now?

Question 21. 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 later.

Your completed answers to these questions are to be handed in with the rest of your writeup for Tuesday, December 4.
What to turn in: For each of these questions, include your computations and explain your reasoning.

Robot Lab for Thursday Nov. 29

Today, you will implement a generic version of state estimation and, then, start familiarizing yourself with an implementation of state estimation and planning on the robot.

The two parts of this lab can be done in either order. Read the whole lab and decide on the order in which you want to do them.

State estimator

You have been using the state estimation for grid worlds implemented in the class `GridStateEstimator` in the file `StateEstSMterm.py`. Now, implement a completely generic state estimator (not just for two-dimensional grids) as a Python class.

The initialization function for the class should take as arguments:

- A list of the possible states in your domain.
- 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 a list of observations, $o_1, \ldots, o_t$, and return

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

Question 22. Describe your implementation of state estimation.

Question 23. 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.

Question 24. Extend the copy-machine model so that there are two actions, *print* and *maintain*. Assume that the model we specified was the *print* action. Provide a model for the *maintain* action, which will tend to improve the state of the printer.

Question 25. Make up a couple of interesting action and observation sequences and show the belief state that results afterwards. Argue whether it's reasonable.

Checkpoint: Thursday 11:30 PM

- Find a staff member, and demonstrate your state-estimation code on the extended copy-machine model.

State Estimation on the Robot

The `~/Desktop/6.01/lab13` folder should have the following files:

- `AvoidWanderTB.py`
- `boxWorld.py`
- `emptySquare.py`
- `GridMap.py`
- `GridStateEstimator.py`
- `KBest.py`
- `Map.py`
- `probcolors.py`
- `search.py`
- `Sequence.py`
- `utilities.py`
- `WanderEstBrain.py`
- `WriteIdealReadings.py`
- `XYThDriver.py`
- `XYThEstBrain.py`
- `XYThGridPlanner.py`

and, also, the `maxRange2.465` subfolder, which contains a set of `.dat` files.

Edit `WanderEstBrain.py` so that the variable `dataDirectory` is defined to be whatever path you unpacked the sonar data files into (see "The Code" below).

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.

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, θ 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/Desktop/6.01/lab13/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 θ so that $P(x, y, \theta)$ 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 shades from red to green, where red is unlikely and green is likely. 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.

```
    robot.driver = TBParallelWithFun(AvoidWanderTB(sonarDistances,
                                                   discreteMotor),
                                     makeBeliefUpdateEveryN(10))
    robot.driver.start()
```

Whew. That was all the initialization. Now, every time we're asked to do a step, we ask the driver to take a step.

```
def step():
    robot.driver.step()
```

Run the simulation

Now, start up SoaR in simulation, select the `she.py` world and run the `WanderEstBrain.py` 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 gray represent locations that cannot be occupied by the robot. 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 (bight red is most unlikely, bright green is most likely). At the absolutely most likely pose (x, y, θ), the robot is drawn, with a nose, in gold.

The second window, labeled `P(O|S)`, shows, each time a sonar observation o is received,

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

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)**; 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.

Repeat using the `emptySquare.py` world in the `lab13` directory. You will need to comment out the lines related to the `she.py` world and uncomment the lines corresponding to the `emptySquare` world in `WanderEstBrain.py`.

Question 26. Explain the relationship between the two windows, and why they often start out similar and diverge over time.

Question 27. 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 28. What happens when the robot is kidnapped?

Question 29. Assume you don't know where in the world you start, are there worlds that have a fundamental ambiguity that can never be resolved, even with perfect sensing and action?

Checkpoint: Thursday 12:15 PM

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

Using the planner

Edit `XYThEstBrain.py` so that the variable `dataDirectory` is defined to be whatever path you unpacked the sonar data files into.

Switch to using `XYThEstBrain.py`. Instead of running the avoid and wander behavior, it runs a planning system similar to that from lab 11. However, the state space it uses is not (x, y), as we saw in lab 11, but (x, y, θ), with θ quantized into four directions.

But how does it know its current pose? Of course, it doesn't really; but it takes the most likely pose out of the state estimator, assumes it's the correct pose, makes a plan, executes the first step, and then re-estimates the pose using our state estimator.

Question 30. Note the state space being used by this planner, what are the actions that it uses? Explain.

Question 31. Run it (in simulation) three or four times in `boxWorld` world (in the lab13 folder). You can change the target pose by editing `XYThEstBrain.py`. What happens? Explain any "strange" behavior that you see.

Question 32. Observe the behavior on the real robot (a couple of demonstrations will be running in the lab). Does it seem to behave differently there?

Question 33. If you wanted to confuse the robot, how would you change the `boxWorld`? Explain.

Checkpoint: Thursday 12:45 PM

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

Homework

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.[1]

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).

Question 34. 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 35. How many possible observations are there in this domain?

Question 36. 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 37. Use your state estimator that you wrote in lab 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 38. 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.

[1]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.

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