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

Work for Week 12, Issued Tuesday, April 29

Overview of this week's work

In software lab

- Work through the software lab.

Before the start of your design lab on May 1 or 2

- Read the class notes and review the lecture handout.
- Do the on-line tutor problems in section 12.1.
- Read the entire description of the design lab, so that you will be ready to work on it when you get to lab.

In design lab

- Do the nano-quiz.
- Work through the design lab.

Before the beginning of your next software lab on May 6 or 7

- Do the on-line tutor problems in section 12.2.
- Submit written solutions to questions 2—9, 13, 14, and 15. All written work must conform to the homework guidelines on the web page.

Note: Exploration is due on Thursday May 8!!! That is the last legal due date.

Do `athrun 6.01 update` to get a directory `lab12` that contains the relevant files. In order to do this entire lab, it will also be important to have the latest version of SoaR. Athena machines and lab laptops will update automatically; to update your laptop, please download a new version of SoaR from the 6.01 software page.

Software Lab: State estimation part 1

In the next two software 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, next week.

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

> 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 `se.py`. Open this file in Idle, but **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`), and connect to the `lab12` directory

```
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.
>>>
```

If you are using your own laptop, download the `lab12` *files from the calendar page on the web. If don't know how to do the operations above on your own computer, then you can start Python, and type*

```
>>> import os
>>> os.chdir('wherever\you\store\your\lab12\files')
```

And continue as follows.

In this Python, type

```
>>> import se
>>> from se import *
```

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

```
>>> reload(se)
```

And if you define a new name in `se.py`, you'll have to do

```
>>> from se import *
```

again as well.

You'll see a set of procedures at the end of the `se.py` file, which will make example worlds of different kinds. We'll start by working with the world defined by `make51p` (which stands for 5 by 1, perfect). In your Python (not Idle) shell, type

```
>>> w = make51p()
```

As a result, 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:

```
w.run()
```

and responding to the text prompts. Typing `done` returns control to Python, but the window will remain. **Do not kill the window until you are completely done with it.** You can type:

```
w.reset()
w.run()
```

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

The state of the system is described with two integers, which are the discrete `x` and `y` coordinates of the robot's actual location in the world. The robot doesn't have access to this information though; all it gets to do is make an observation of the color of the room it is in. When the world is created, and after every step, it prints out the "belief" distribution over the array of possible robot locations. Then, it prints out what observation the robot actually makes of its world (if the observation function is probabilistic then the robot might observe "red" even if it is standing in a square that is white). Finally, it prompts you for an action that the robot should try to take. The simulator executes that action (using the motion model in the world, which may be stochastic), updates the hidden state of the world (the robot's location), and generates a new observation.

Question 1: To get the idea of how this works, move the robot east few times. Write down the numerical values associated with the robot's belief that it is in each of the squares. Give a qualitative explanation for how the belief evolves. 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: $\mathsf{P}(\mathsf{O}_t = \mathtt{o}_t | \mathsf{S}_t = \mathtt{s}_t)$. In our case $\mathtt{o}_t$ ranges over *white*, *black*, *red*, *green*, and *blue*, and $\mathtt{s}_t$ 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:

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

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 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 and colors are the following dictionaries defined in `se.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 `w` so that it has only 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 for a world that is a torus (that is, the surface of a donut. There's no "edge" in this world, the surface wraps around to the other side) and where motion is deterministic. Make sure that your motion model works on two-dimensional worlds.

Test your motion model in a 5 x 5 world such as the one defined by `make55` in `se.py`. Select test cases that demonstrate the necessary features of the motion model. Show the input and output for calls to your motion model function. Explain.

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 landed in the torus model. Make sure that you do this so that it works on two-dimensional worlds.

Provide test cases as described in problem 7.

Question 9: Define your own noisy two-dimensional motion model, implement it and test it (as described in problem 7). Make sure you also provide an English description of the behavior.

Final Project : Part 1: Seek the Light

For the final project, we want to be able to build a brain that can lead the robot to a light somewhere in a maze. We will assume that we know the map for the maze but are starting from an unknown location; we will also assume that the robot knows several candidate locations for the lights. Developing the software to localize the robot relative to the map and traverse the maze will be the subject of next week's lab. This week, we will focus on the light finding.

In Lab 10, you built a software controller for the robot head running from Python and doing input and output through the NIDAQ box. This week we will integrate that into SOAR. We will be using the terminating sequential machine framework that we have seen several times (Labs 4 and 11 and the midterm). This will enable us to combine the light-finding behaviors more readily with the planning and execution behaviors that we explored in last week's lab.

By the end of this lab, we would like your robot to be able to exhibit the following behavior:

- It drives around the world in a square;
- When it reaches one of the corners of the square, it rotates 360 degrees;
- During the time that it is rotating, if the light in either of its photosensors is above some threshold, it begins following the light, with both its head and its body; and
- When it comes very close to the light, it stops.

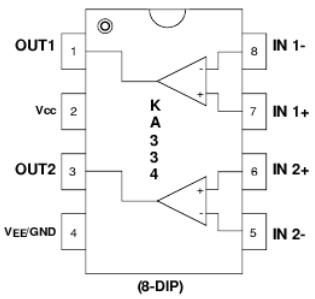
A head on your shoulders

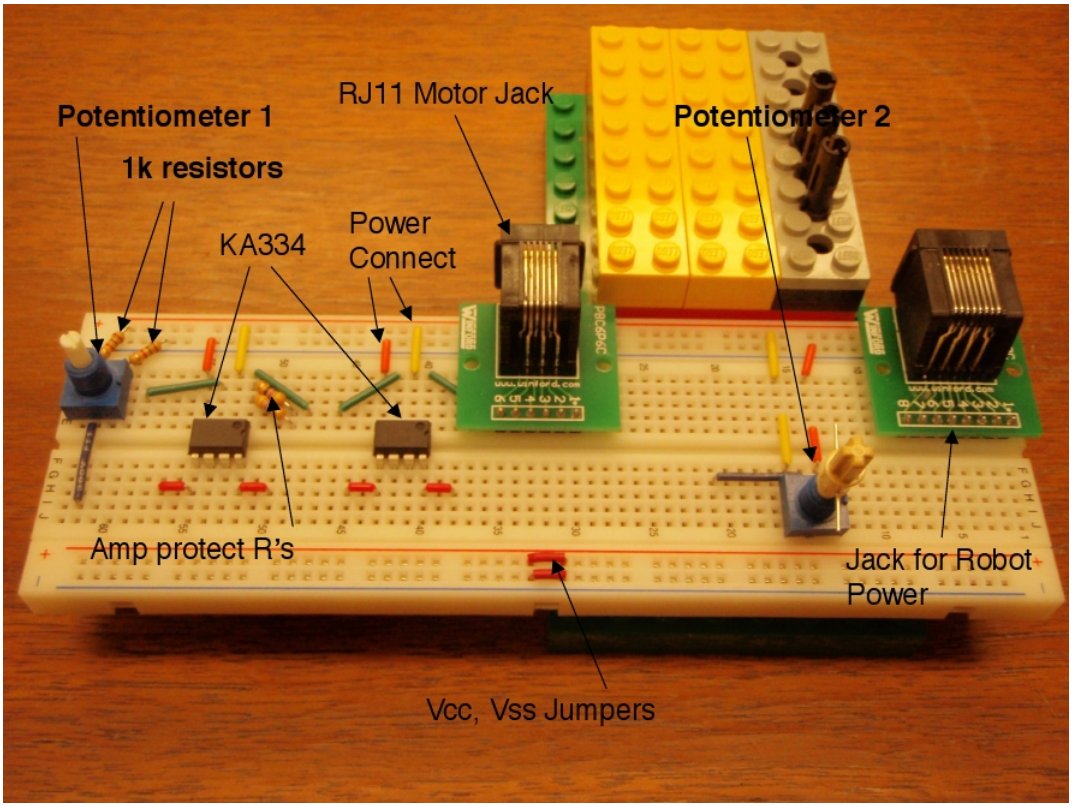
The circuits needed for this lab are a basic version of we built in lab 10. Because we are changing partners and many heads were left with a random assortment of useful and non-useful circuitry, we're starting them all fresh.

Following is a schematic that will provide:

- Wiring to the NIDAQ inputs so that you can read the voltage drop across each of your photosensors, and the neck potentiometer.
- Wiring to the NIDAQ output to set the voltage on one terminal of your motor, with a buffer to provide enough current.
- The other terminal of the motor connected to a buffered voltage of approximately 5V.

You will need to interface your circuit to the NIDAQ box on your robot and you will need to get power from the robot's battery.





You will be plugging in the blue wire on top of your robot into the 8-pin "Jack for Robot Power" shown in the figure above. Note that the jack for connecting the motor has 6 pins; there's a different jack with 8 pins that you need here. If you don't have such a jack on your protoboard, please find one in the lab and insert it into your board. You will need to move the jack forward from where it is shown in the figure so you can connect wires to the pins.

You need to make the following connections to the pins of this jack:

Pin 1: AO0 on NIDAQ - connect to your motor via an op-amp buffer

Pin 2: AO1 on NIDAQ - leave disconnected

Pin 3: AI5 on NIDAQ - connect to the neck pot (Potentiometer 2)

Pin 4: AI6 on NIDAQ - connect to one photo-sensor divider

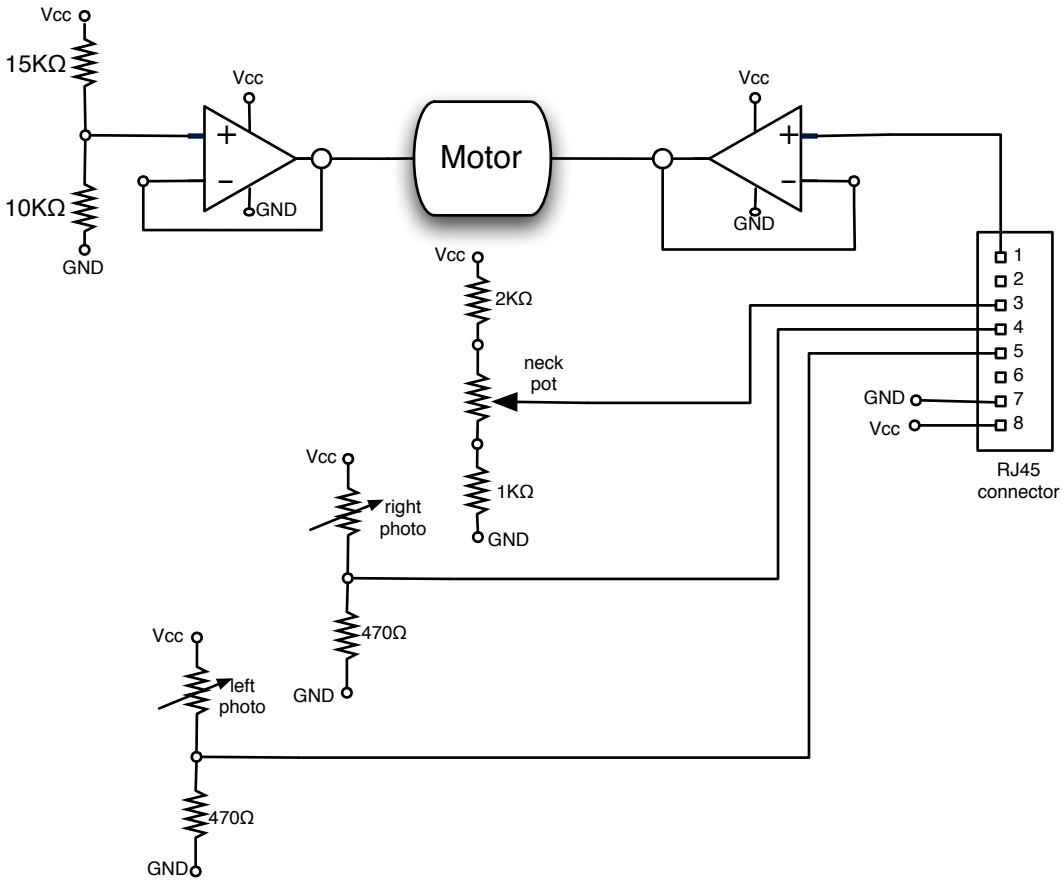
Pin 5: AI7 on NIDAQ - connect to one photo-sensor divider

Pin 6: AI4 on NIDAQ - leave disconnected

Pin 7: GND from robot - connect to ground rail on your board

Pin 8: +12V from robot - connect to the power rail on your board. This will supply all your power. **Note that the voltage from the robot is often closer to +13V. This shouldn't present too much of a problem, but you should measure it, so you know whether your dividers are working; and you may need to correct, in software, for the fact that your virtual ground is not exactly 5V.**

Please build this circuit on your board.



Question 10: What will be the minimum and maximum voltages into pin 3 if V_{cc} is 13?

Affix your head to the front of your robot by gently pushing the base of the head onto the gray Lego plate on top of the robot. **Make sure that your robot is turned off.** Plug in the blue wire with a telephone connector on top of your robot into the RJ11 jack.

Warning: When you power on your robot, your circuit will have power. If your NIDAQ box's output is 0, then your head motor will have a voltage drop of 5V across it, so it will smash itself against one of the stops and continue pushing as hard as it can. Since the resistance of the motor is 5 Ohms and you are commanding 5 Volts, your poor op-amp is pumping 1 Amp into the motor, therefore the op-amp will get very hot, melt the board and eventually burn out. Sometimes the failure can be dramatic (a pop, a cloud of smoke and the horrible smell of burning plastic).

So, PLEASE disconnect the cable from your circuit to the motor until you have your controller running and have determined that you have a sensible voltage (near 5V) commanded from your NIDAQ. You can see the commanded voltages in the NIDAQ server window, by printing them from your software, and by measuring them on the board with a multimeter.

Question 11: Verify that your board is set up properly:

a. Open a Terminal window, connect to the `lab12` folder and start the NIDAQ server:

```
cd ~/Desktop/lab12/designlab/
./NIDAQserver --output
```

b. Start Idle and load the file `test12.py`. Do `readTest()` while you shine a light on and off the photo-sensors. Measure and record the range of voltages on the photo-sensors when (a) there is no bright light near, (b) there is a bright light a few feet away and (c) the light is about a foot away. The robot needs to be powered up for your circuit to work.

Make sure you can read the neck potentiometer, and that the readings are sensible.

c. **Make sure that your motor cable is disconnected.** Use a multimeter to measure that you have 5V on one of the terminals of the motor jack. Run `writeTest()` and use your multimeter to verify that the output of the op-amp driving the other terminal of the motor has the voltages that are being written.

Checkpoint 1: 45 minutes

- Find a staff member and show them your head mounted on the robot, with the wiring to the jack from your board and with the wire to the motor disconnected. Show that you can read the photo-sensors and neck potentiometer, and that you can command a voltage to the motor terminals.

A brain for your head

We now want to write a SoaR brain to control the head. We want to do this in SoaR because we will want to move the robot in response to what the head is sensing. Since we will want to combine this functionality with the planning behaviors we did last week, we are going to structure the controller as another TSM (terminating state machine).

Look in the file `lightBrain.py` for a skeleton for you to fill in with the controller. Note that we have extended the input/output interface for behaviors to include the inputs and outputs available on the NIDAQ. The file `io.py` implements the new interface.

So, the input to terminating state machines serving as robot behaviors is an instance of the class `SensorInput` which has three variables, `sonars`, `odometry` (this is the robot pose) and `nidaqInputs` which are the neck pot voltage and the two eye voltages. The output of these state machines should be an instance of the class `Action` which specifies the robot's forward velocity, rotational velocity and the voltage for the NIDAQ AO0 (the head motor). Note that these arguments default to reasonable values if not specified, which allows us to write behaviors for controlling the wheels that are independent of whether or not we are also controlling the head (and vice versa).

```
class SensorInput:
    def __init__(self):
        self.sonars = sonarDistances()
        self.odometry = pose()
        if useNidaq:
            self.nidaqInputs = readNIDAQInputs()
```

```
referenceVoltage = 5.0
class Action:
    def __init__(self, fvel = 0.0, rvel = 0.0,
                 voltage = referenceVoltage):
        self.fvel = fvel
        self.rvel = rvel
        self.voltage = voltage

def doAction(a):
    if useNidaq:
        writeNIDAQ(a.voltage)
    motorOutput(a.fvel, a.rvel)
```

The TSM classes `RotateTSM` and `ForwardTSM` in `lightBrain.py` illustrate the use of this interface.

```
    def start(self, input):
        (x, y, theta) = input.odometry
    ...
    def currentOutput(self):
        return Action(rvel = self.rotationalGain * ...)
```

Question 12: Fill in the `trackLightTSM` class so that it has the required TSM methods (`__init__, start, step, currentOutput, done`) and it implements a light tracker similar to the one you did in Lab 10. (Note that in this slightly revised TSM class, the `start` method plays the same role as the `reset` method from early in the semester.) It should take the input from the photo-sensors and compute a voltage command for the robot. Pick a gain so that the head is quick but does not oscillate. Assume for now that this behavior never terminates. Debug this behavior running within SoaR. **Use your multimeter to make sure that the behavior produces sensible voltage commands before connecting the motor cable.**

Question 13: Extend your class definition so that it moves the robot to follow the light. The robot should turn in the direction that the head is pointing and should move towards the light. Have the robot stop moving forward (but do not terminate) when the light gets to the brightness that means that the light is about a foot away. You can debug this in the simulator without actually running the robot; if you have the robot powered up, so that your head circuit has power, the simulated robot should respond as desired. (Alternatively, you can rock the robot back so that its wheels are off the ground, and check visually to see if it's turning in the right direction, before running it around.) After your behavior works in the simulator, get it to work on the real robot. Hand in your definition of the completed `trackLightTSM` class and an explanation of how it works. Describe your testing.

Checkpoint 2: 90 minutes

- Demonstrate your robot following a light. Its head should track the light quickly and its body should follow.

A scanner brightly

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.
- `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.

Question 14: 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. Hand in your behavior definition and an explanation of how it works.

Question 15: 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. Hand in your behavior definition and an explanation of how it works. Describe your testing.

Checkpoint 3: End of lab

- 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.

Exploration: Due Thursday May 8

This is worth 5 out of 10 exploration points

State estimator

Implement a completely generic state estimator as a Python class. The initialization function 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 observations, $o_1, \ldots, o_t$, and return

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

Exploration 1: 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.

Exploration 2: Extend 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, and print out a test page. Make up a couple of interesting action and observation sequences and show the belief state that results afterwards. Argue whether it's reasonable.

Diagnosing Pneumonia

This is worth 5 out of 10 exploration points

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]

[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.

Let the possible states of the patient be: **free** of bacteria, **colonized** by bacteria, or **pneumonia** from bacteria. (Assume these states are mutually exclusive and exhaustive: every patient is in exactly one of them). 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).

Given these three observation variables, there are 8 possible observations (because each variable could be either true or false). You can, if it seems justifiable to you, make the assumption that each of these observations is independent, given the state of the patient. That is, that the probability of abnormal temperature given pneumonia is independent of the probability of abnormal blood gas given pneumonia. If the observations are conditionally independent given the state, then $\Pr(o_1, o_2, o_3|s) = \Pr(o_1|s)\Pr(o_2|s)\Pr(o_3|s)$, where o_1, o_2, and o_3 are the observations.

Exploration 3: 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.)

Exploration 4: How many possible observations are there in this domain?

Exploration 5: 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.)

Exploration 6: Use your state estimator from the previous part of the exploration to do 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.

Exploration 7: 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.