MASSACHVSETTS INSTITVTE OF TECHNOLOGY
Department of Electrical Engineering and Computer Science
6.099—Introduction to EECS I
Fall Semester, 2006

Assignment for Weeks 11 and 12

Issued: Tuesday, November 14

Because of the holiday, we're rearranging things slightly. See below for a detailed description of what exercises are due on what days.

- Pre-lab problems **due Thursday, November 16**
  Read this whole document; **Questions 1–10**, written
- In-lab exercices for **Thursday, November 16**
  Questions 11–26; 29–34
- Post-lab problems **due Tuesday, November 21**
  Questions 27, 28
- In-lab exercises for **Tuesday, November 21**
  Questions 35–37
- Post-lab problems due **Tuesday, November 28**
  Questions 38–42 and written lab report containing answers to all questions from in-lab exercises (be sure to keep notes!) **11–26, 29–34, 35–37**

State estimation

In this lab, 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.

1. Grid World Simulator

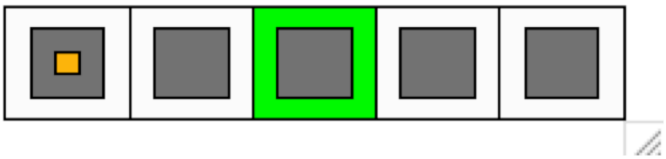
Pre-Lab Due Nov 16

Download the code for this lab, in `ps11.zip` from the calendar page. In it, you will find `DrawingWindowOld.py`, `StateEstimationNoisyNN.pyc`, `MissingCode.py` and `Lab11.py`. Visit `Lab11.py` in Emacs and evaluate the buffer.[1] You'll see a command at the end of that file, that says:

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

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

[1] Note that if you want to run this code on a Windows machine, it (annoyingly) doesn't work if you run it from Emacs; there's some kind of disagreement between Emacs and TKinter (the graphics system that Python uses) that will make Emacs hang if you try it. So, on Windows, just start Python in a shell, and explicitly do `from Lab11 import *`, and remember to do `reload(Lab11)` if you change it.



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.[2]
- 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 (using `w51pp.east()` and `w51pp.west()`), 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.

The sensor model, generally speaking, is supposed to specify a probability distribution over what the robot sees given what state it is in: $\Pr(O_t|S_t)$. In our case the random variable 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

[2]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.

possible observations. So, we really only need to specify $\Pr(observedColor|actualColor)$. In our program, this set of distributions is specified as a tuple of tuples, where each row corresponds to a different actual color, in the following order: white, black, read, green, blue. 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), and create a world to test it in.

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_t. That is, $\Pr(S_{t+1}|S_t, A_t)$. 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 also 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 list 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):
    def cx(a): return clip(a, 0, xmax-1)
    def cy(a): return clip(a, 0, ymax-1)
    return (((cx(rx+ax),cy(ry+ay)), 1.0),)
```

Question 7. Explain what the procedure `perfectMotionModel` does.

Question 8. 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 9. 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. Test it out by moving the robot around.

Belief state update

The robot's *belief state* is a probability distribution over possible states of the world. The belief state at time t represents the probability that the robot is in each of the possible states at time t, given that it started with some initial uncertainty about where it was (the belief state at time 0, also sometimes called the *prior* distribution), and took the sequence of action that it actually took, and got the sequence of observations it got.

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.

When the robot takes a transition, we need to update the probability that the robot is in some state s_i. If b_{itold} was the old belief state (a whole vector of values) so that $b_{old}(s_i)$ was the belief that the robot was in state s_i, then after the robot takes an action a, its new belief that it is in state s_i is:

$$b_{trans}(s_i) = \sum_{s_j} \Pr(S_{t+1} = s_i | S_t = s_j, A_t = a) b_{old}(s_j) \ .$$

That is, for each state s_j that the robot might have been in before, we add up how likely it is that the robot was in that state, times the probability that it would have moved from s_j to s_i by trying to do action a.

Now, the robot makes an observation, o, and we need to readjust the belief state to take this into account. For each state, s_i, the sensor model tells us how likely it is that we would have

made that observation if we were in state s_i, $\Pr(O_{t+1} = o|S_{t+1} = s_i)$. We want to calculate $\Pr(S_{t+1} = s_i|O_{t+1} = o)$, which by Bayes' rule, is

$$\frac{\Pr(O_{t+1} = o|S_{t+1} = s_i)\Pr(S_{t+1} = s_i)}{\Pr(O_{t+1} = o)} .$$

This isn't too bad. $\Pr(O_{t+1} = o|S_{t+1} = s_i)$ is our sensor model, and $\Pr(S_{t+1} = s_i) = b_{trans}(s_i)$ is our belief state after the transition update. But what about the denominator? We can use the law of total probability, and then the definition of conditional probability, to get:

$$\begin{aligned}\Pr(O_{t+1} = o) &= \sum_{s_j} \Pr(O_{t+1} = o, S_{t+1} = s_j) \\ &= \sum_{s_j} \Pr(O_{t+1} = o|S_{t+1} = s_j)\Pr(S_{t+1} = s_j)\end{aligned}$$

This is cool! It means that we can calculate the new belief state from b_{trans} by:

- Multiplying the entry for each s_i by $\Pr(O_{t+1} = o|S_{t+1} = s_i)$
- Dividing each entry by the sum of all the entries

Question 10. Convince yourself that at the end of this, all of the entries in the belief state will sum to 1.

In Lab on Nov 16

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 11. First, what is the robot's prior belief $b(s_i)$ for each of the states?

Question 12. 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 $\Pr(S_{t+1} = s_i | S_t = s_j, \mathit{east})$ for all i, j. Now, use this to compute $b_{\mathit{trans}}(s_i)$ for all i.

Question 13. 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, compute $\Pr(O_{t+1} = \mathit{white} | S_{t+1} = s_j)$ for all states.
- Then compute $\Pr(O_{t+1} = \mathit{white} | S_{t+1} = s_j) \Pr(S_{t+1} = s_j)$ for each state. Remember that at this point the $\Pr(S_{t+1} = s_j)$ we are using is b_{trans}, which is the belief state that resulted from the transition, not the original prior.
- Now, compute $\Pr(O_{t+1} = \mathit{white})$.
- Finally, compute $\Pr(S_{t+1} = s_i | O_{t+1} = \mathit{white})$.

Question 14. Which action could the robot take at this point to make its location completely unambiguous?

Question 15. 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.

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.

Question 16. 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 17. Now, what's the distribution over what the robot predicts that it will see, given its current belief state? That is, what is $\Pr(O_1 | A_1 = \mathit{east})$?

Question 18. Compute the next belief state if the robot sees "white". That is, what is $\Pr(S_1 | O_1 = \mathit{white}, A_1 = \mathit{east})$?

Question 19. Compute the next belief state if the robot sees "green".

Question 20. Compute the next belief state if the robot sees "red".

Question 21. Go east again. (That's twice now). Explain why, in this particular run, in your simulation, you got the result you got.

Question 22. What (approximately) would happen if you did the `stay` action 10 times?

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

Question 23. What is the belief state after telling the robot to go east, but before it gets the observation?

Question 24. What is the next belief state if it observes "white"?

Question 25. Move the robot to the green square. What is the belief state now?

Question 26. 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.

PostLab due Nov 21

Writing code

Look in the file `MissingCode.py`. It contains all the definitions from `StateEstimationNoisyNN.pyc`, but with these definitions (of methods for class `World`) missing. Please provide them and run your code and verify that it computes the same thing as the compiled code we gave you. These are the computational instantiations of the transition and sensor parts of the belief-state update. (Remember to change the `import` statement at the top of your file to import `MissingCode` instead of `StateEstimationNoisyNN`.

```
def actionUpdate(self, (ax, ay)):
    # Make an empty belief state
    newbelief = zeros((self.xdim, self.ydim), Float)
    # Your code here, to compute a new belief state
    # Use self.motionModel

    self.belief = newbelief


def observationUpdate(self, actualObs):
    # Your code here, to update the belief state
    # Use self.sensorModel
```

Question 27. Provide your code with some comments, and a demonstration that it works.

Generic state estimator

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

- a tuple of states (these could potentially be integers or strings)
- a tuple of actions

- a tuple of observations
- a transition model, which is a function that consumes a state (s_j), and action (a), and another state (s_i), and returns $\Pr(S_{t+1} = s_i | S_t = s_j, A_t = a)$
- an observation model, which is a function that consumes a state and an observation, and returns $\Pr(O_{t+1} = o | S_{t+1} = s)$.

Your class should have the following methods:

- to set the prior belief state (it might be nice to have special ones for the uniform belief state, or for one in which you start out with certainty in some state).
- to update the belief state based on an action
- to update the belief state based on an observation
- to print out the belief state

Question 28. 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.

2. Robot Pose Estimation

In Lab on Nov 16

Now it's time to apply the ideas of state estimation to the real robot. We can test them first in Pyro, and then on the actual robot. The idea is to put the robot in a space with obstacles, and give the robot a "map" of the space, so it knows where the obstacles are. The problem is, that the robot doesn't know where it is, in that space. The goal is for it, in spite of its noisy odometry and sensor readings, to localize itself (that is, to figure out where it is located on the map). This problem is conceptually the same as in our little grid world, but there are lots of complications.

State space First, we have to think about what the underlying state space of the robot is. Our robots can occupy what is essentially a continuous range of (x, y) positions in the world. Furthermore, in order to predict the result of an action or the expected observation, we need to know the robot's orientation. So, the robot's state can be described as the product of the continuous variables x, y, and $theta$. It is very difficult to represent a completely general probability distribution over such a space. We'll take a relatively easy way out by discretizing the space: we'll divide each of those state variables into k uniform-sized segments, resulting in a state space that consists of k^3 possible points, uniformly spaced on a three-dimensional grid, embedded in the underlying continuous space.

Action space Similarly, the action space of the robot is very hard to characterize simply. We send the robot velocity commands, but it's hard to predict what the results will be very precisely, because we don't know much about the robot's acceleration, how often our commands are executed, etc. However, if we sample the odometry reading at time t and again at some future time $t + k$, we have a quantity that we can plausibly treat as the commanded action. It is a $(\delta x, \delta y, \delta theta)$ that describes how the robot thinks it changed pose. Of course, we know from lab 1 that the robot's odometry is pretty noisy. But we can model that just as we modeled noise in the state transitions for the grid world. We'll treat the odometry delta as the "intended" action, and model the actual action as being a noisy version of the intended one.

Observation space Finally, we have to model the robot's sensory abilities. We'll just model the sonar sensors, which can give us a lot of information about where the robot is. But we have to remember that there is noise in the sensor readings and also that many different (x, y, θ) poses of the robot would yield the same sensor readings. So, for us, an observation will be a vector of 8 sonar readings, $\langle o_0 \dots o_7 \rangle$. The sensor model is supposed to give us a probability distribution on the sonar readings, given the robot's pose: $\Pr(\langle o_0 \dots o_7 \rangle | x, y, \theta)$. This seems very difficult, because the sonar readings are continuous values and because there are eight of them.

Let's tackle the second problem first. Given the robot's pose, we can easily compute a *nominal* set of sonar readings; let's call them $\langle g_0 \dots g_7 \rangle$ (where g stands for good). These are the readings we would get if the sonars were perfect, and returned the distance along the ray centered at each sensor to the nearest obstacle. Just as we did in the simple grid world, we're going to make the simplifying assumption that the noisy sonar readings only depend on the nominal readings; that any two poses of the robot with the same nominal readings will generate the same distribution over actual readings. So, now our problem is to specify $\Pr(\langle o_0 \dots o_7 \rangle | \langle g_0 \dots g_7 \rangle)$.

It's time for yet another reasonable assumption (you'll find that the art of making mathematical models of the real world is figuring out what kinds of assumptions are both mathematically helpful and realistically plausible). We'll assume that the amount of noise in one sensor reading is independent of the noise in the other sensor readings, given the nominal readings. This would be a good assumption if the noise is a property of the individual sonar sensors for example; it would be a bad assumption if it were a more global property of the environment, like having an ultrasonic rodent-repeller in the room. If we can make this independence assumption, we get a huge simplification in our problem:

$$\begin{aligned} \Pr(\langle o_1 \dots o_7 \rangle | x, y, \theta) &= \Pr(\langle o_1 \dots o_7 \rangle | \langle g_1 \dots g_7 \rangle) \\ &= \prod_{i=0}^{7} \Pr(o_i | g_i) \end{aligned}$$

For those of you have run across the notion of independence or conditional independence of random variables, it will make sense that we can convert the joint distribution into a product.

And assuming that all of the sensors have basically the same noise behavior, we only need to specify a single model: $\Pr(o|g)$. This is a probability distribution over a single actual sonar reading o, given its nominal reading g. We could do this by discretizing the values of o, and providing a probability for each of those values. But we'd have to discretize the g values, too, and the table starts to get big. Instead, we'll keep things continuous and use a Gaussian distribution, which is a way to say which elements of a continuous interval are more likely than which other ones.

Just to review, the Gaussian (also called normal) distribution has two parameters: the mean μ, which describes the most likely value, and the variance σ^2, which describes how spread out the distribution is. If we fix σ^2 to some value (we could determine this by collecting sonar readings and doing some statistical analysis) and set μ to be the nominal value g, then this distribution provides a plausible description of the likelihood of various o values.

So, we'll assume that the conditional distribution on an observation random-variable O, which has nominal value g, is $N(g, \sigma^2)$; that is, that it is normally distributed with mean g and variance σ^2. In our model, which you'll use below, we set σ to 0.5, which makes the variance 0.25. (As you must know by now, this is a pretty terrible model of the error in the sonars: we should really be modeling the fact that the sonars often bounce off of walls when the angle is too steep, but we're being lazy for now).

In Simulation

Look in the directory "SoaR/brains/ps12code/". It should have the following files:

- `Gridmodule.py`
- `GridWander.py`
- `LPKWanderModule.py`
- `utilities.py`
- `shannon20.dat`
- `empty20.dat`
- `tutorial10.dat`
- `tutorial20.dat`
- `tutorial30.dat`

Now, start up SoaR in simulation, using the `Tutorial` world, and use `GridWander.py` as the brain. When it starts up, you'll see two new windows.

The first window, labeled `Belief State`, shows the outline of the obstacles in the world, and a grid of colored squares. Squares that are colored black represent locations that cannot be occupied by the robot. (Note that the obstacles seem too "fat": this is because they are grown by the radius of the robot, which lets us think of the robot just as being a point at the centroid of the real robot.[3]) For the purposes of this window, for each (x, y) location, we find the θ^* value that is most likely, and then draw a color that's related to the probability that the robot is at pose (x, y, θ^*). The colors go in order from more to less likely: red, green, blue, purple. At the absolutely most likely pose, the robot is drawn, with a nose, in orange. You may also notice, as the belief state evolves, a small black line; it is drawn starting from the most likely square on the previous step, and shows the (x, y) component of the odometry reading that was used in the update.

The second window, labeled `Pr(o|s)`, shows, each time a sonar observation o is received,

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

for each square x, y. That is, it draws a color, as above, that shows how likely the current observation was in each square, using the most likely possible orientation. It also draws an orange robot with a "nose" at the location for which this percept was the most likely.

The brain `GridWander.py` just uses our standard avoid and wander program, but keeps the robot's belief state about its pose updated as it does so. In case you're wondering, the file `tutorial20.dat` contains the ideal sensor readings at every x, y, θ pose on a 20 x 20 x 20 grid, assuming that the walls of the *Tutorial 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.

Run the simulation. 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. Currently the belief state is updated every 5 steps of the underlying avoid and wander behavior. Try changing that (the variable `updateInterval` in `GridWander.py` and seeing what happens.

[3] This is a somewhat crude version of the the right thing, which is to think about exactly what poses are legal (in some cases, the robot's centroid could be at an x, y location without bumping into things if it is at one orientation, but not another.)

Question 29. Explain the relationship between the two windows, and why they often start out similar and diverge over time.

Question 30. What happens when the robot is kidnapped?

Quality of Localization

In the simulator, you can "cheat" and find the true pose of the robot. Inside a `brain`, this method will get you the actual pose of the robot (the angle business is due to a mismatch between the coordinate systems of Pyro and of our maps):

(**Note:** this code snippet was corrected for Tuesday's lab.)

```
def truePose():
    simulator = cheat()
    bestPose = (simulator.absx, simulator.absy, simulator.absth)
    return bestPose
```

You can convert a pose (just a tuple of `(x, y, th)`) to a set of indices into the grid by doing

```
(ix,iy,ith) = robot.gridModule.grid.poseToIndices(pose)
```

and then you can find the logarithm of the probability associated with that grid square by doing

```
robot.gridModule.grid.grid[ix,iy,ith]
```

To convert this number to a probability, use:

```
import math
math.exp( robot.gridModule.grid.grid[ ix, iy, ith ] )
```

You can also convert a set of grid indices into a pose using `indicesToPose`.

The robot's opinion of the grid square with the highest probability (i.e., the tuple (ix, iy, ith) with the highest probability of containing the robot) can be found with:

```
robot.gridModule.grid.bestPose
```

Question 31. Think of a good way to measure how well the localization is working, and augment `GridWander.py` to compute that measure and print it out. Be sure your measure will be comparable across grids of different resolutions.

Question 32. Try using grids of different sizes. You can do this by changing `tutorialWorld20` in `GridWander.py` to `tutorialWorld10` or `tutorialWorld30`. How does that affect the localization performance?

Question 33. We've prepared two other environments for you. Try using `shannonsWorld20` instead of `tutorial20`. You'll have to choose `ShannonsWorld` in SoaR when you specify what simulator to use. Try using `emptyWorld20` in your program and `EmptyWorld` in the simulator. Which of these worlds is easier for the robot to localize in? Why? Could you fix it with better sensors?

Question 34. If the robot were stationary and kept getting sensor readings and doing state updates, what would happen to the belief state? Under what circumstances would this be a good or a bad thing?

On the Robot

In Lab on Nov 21

Using a robot and the available playpens (we'll have one corresponding to Shannon's World, which can be converted to Empty world by removing the box in the middle), gather some logs of sensor data as the robot moves around in the playpen[4] . Take notes while the robot is moving, and draw a rough picture of the trajectory it followed. Now, replay the logs and watch the state estimation windows. Have the code print out the most likely state at each state estimation step. Do they correspond to the actual positions of the robot, according to your notes?

If you find the computational overhead of doing the state estimation is too annoying while gathering your robot logs, you could actually comment out the state-estimation code while running on the real robot; since this is a passive part of the program and doesn't affect the robot's actions, you'll get the same data if you just run the basic avoid and wander behavior. Of course, this is only true because we have set a switch in SoaR that makes the robot move a fixed amount of time and then stop on each step, rather than moving continuously until the next move command is received.

Sensor noise We picked the standard deviation used in the sensor model pretty arbitrarily to be 0.5. Inside the file `GridModule.py` you will find a statement

```
sigma = 0.5
```

How should we set this value? There are actually two sources of variability that are being accounted for, here. The first is the usual one of how much variability there is in a sonar reading given an

[4]To gather a log of the sensor data that the robot gathers as it's moving around, add `writeLog("robot.log")` to your brain. This will keep track of all the sensor and odometry readings that your robot got. Now, you can run your program again, but replace that line with `readLog("robot.log")`; the robot won't move, and instead it will "hallucinate" the sensor readings that it had on the previous run whenever you call `sonarDistances()` or `pose()`. This allows you to sit quietly at your desk and print out values, or stare at the graphics windows to really understand what was going on, rather than trying to figure it out as you chase your robot around the lab.

actual distance to the wall. In the simulator, there is a little bit of noise on the sensor readings, so we aren't getting much variability from that source; but of course there is more of this kind of noise on the real robot sensors. The other one is due to the discretization: we are treating all the poses in one cube of the pose space as if they were the same, and there may be significant variability of the sensor readings from the different poses in that cube.

The right strategy for setting the value would be to sample a bunch of sensor readings, and then do some statistical analysis to estimate the mean and variance. Instead of doing that, you might try changing the variance in the model and seeing what effect that has on the quality of the estimates you get.

Question 35. Why is it that we had to make the robot move only a fixed amount of time per step, rather than moving until it gets the next move command?

Question 36. Does changing the sensor noise model make the estimation work better? How did you change it?

Question 37. Think back to lab 2. Is $sigma = 0.5$ reasonable for sonar measurements on bubble wrap? What would we need to do to our model if there were no bubble wrap?

General state estimation problems

Diagnosing Pneumonia

Post Lab due Nov 28

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.[5]

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).

[5]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.

The transitions between disease states can be modeled with a probabilistic state transition model, and the symptoms can be modeled as occuring probabilistically given the underlying disease state of the patient.

Question 38. 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 39. How many possible observations are there in this domain?

Question 40. 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 41. 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 42. 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.

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
- Robot localization can be done robustly, even with noisy sensors and effectors
- A variety of modeling tricks, including discretization and independence assumptions, are necessary to make the system practical
- Making formal models of real problems is difficult and important
- Programming practice