Design Lab 11 *6.01 – Fall 2011*

Robots in Hallways

Goals: Design Lab 11 lays the groundwork for estimating the location of a robot as it moves down a hallway starting from an uncertain location. You will:

- Explore observation models and transition models
- Simulate bayesian state estimation for robot moving in a hallway
- Prepare for building a real-world system for robot position localization

Resources: This lab should be done **individually**.
Do `athrun 6.01 getFiles`. The relevant file (in `~/Desktop/6.01/designLab11`) is
- `designLab11Work.py`: code for simulating state estimation on colored hallways

1 Introduction

When we have a system with internal state that we cannot observe directly, then we can consider the problem of *state estimation*, which is trying to understand something about the internal hidden state of the system based on observations we can make that are related to its state. Examples of such systems include:

- A copy machine, where the hidden state is condition of its internal machinery, the actions we can take are to make copies, and the observations are the quality of the copies.
- A robot moving through a hallway, where the hidden state is the location of the robot, the actions we can take are to move to the east and west, and the observations are colors (which may not always accurately reflect the true underlying colors of the walls).
- A person playing a video game, where the hidden state is which monster the person is trying to kill, there are no explicit actions, and the observations are the moves the person is making.

State estimation is the process of taking in a sequence of inputs we have given to the system (we sometimes call them actions) and observations we have made of the system, and computing a probability distribution over the hidden states of the system. In the next couple of labs, we'll use basic **state estimation** to build a system that estimates the robot's pose, based on noisy sonar and odometry readings.

To do probabilistic state estimation we need a model with three components:

- An initial distribution: a probability distribution over the states of the system that tells us the probability that any state is the initial one.
- An observation model: a conditional probability distribution that tells us the probability of seeing each of the possible observations, given the state.
- A transition model: a conditional probability distribution that tells us the probability of being in each state at time $t + 1$, given a state at time t and an action at time t

Some of the software and design labs contain the command athrun 6.01 getFiles. Please disregard this instruction; the same files are available on the 6.01 OCW Scholar site as a .zip file, labeled Code for [Design or Software Lab number].

We'll start by building up some familiarity with observation and transition models, then we'll build your intuition for the application of these ideas in a simple simulated world, then we'll move on to modeling the real robots in more detail, in preparation for design lab 13.

2 Hallway World

We are going to work with an abstract simulation of a robot moving up and down a one-dimensional hallway made up of a fixed number of colored rooms.

- The robot starts out knowing how many rooms there are, but not knowing what room it is in.
- The robot knows a probability distribution over what observations (colors) it will make in each room.
- The robot can attempt to move in each direction in the hallway; it will not always move completely reliably. If it is up against the left end of the hallway and tries to move to the left, it will stay in the leftmost room; similarly for moving right when it is in the rightmost room.

The goal of the robot is to determine what room it is in. We will approach this problem using **state estimation**.

Step 1. Start idle with `-n`, and load the file `designLab11Work.py` and run it. (We will not be using Soar. This is a stand-alone piece of software). Now, type

```
p = makePerfect()
p.run(10)
```

At this point, `p` is an instance of a little application that both simulates the robot moving in the world and shows the estimated belief state.

Each square corresponds to a room in the world. Each room has a "true" color, which determines the distribution over the observations that the robot will make in that room. The room's true color is shown in the outer rim of each square.

The robot's *belief state* is a probability distribution over which room it is in. In the window, the current belief state is displayed in the colors of the inner squares in each block, with brighter red values closer to zero and brighter blue values closer to one. The color black is assigned to the probability associated with the uniform distribution (in this case 0.2). The probability assigned to each room is also printed in each square.

In fact, `p` is a combination of a ***stochastic state machine*** that simulates the behavior of the robot-world system and a *state machine* that does **state estimation**, based on the robot's actions and observations, to compute a new belief state on each time step.

The way this complex machine works is the following:

- When `p` is initialized, at the beginning of each call to `p.run`:
  1. The state estimator *initializes the belief state* to the starting belief state which, in this case, is the uniform distribution over the rooms.
  2. The simulator selects an *initial starting location* for the robot at random from the starting distribution. Note that *the state estimator does not know this true location*; it is just used inside the simulation. It is also not displayed in any way in the window or the printed output.
- On every step of `p`:

1. The simulator *generates an observation*, drawing from the distribution of observations that is associated with the room that the robot is currently really located in. This observation is a color name, like 'white' or 'green'.
2. The simulator *prompts the user for an action* that the robot should take. The action must be an integer between -4 and 4, inclusive.
3. The simulated *robot moves* a number of rooms that depends on the specified action; but if the robot's motion model is noisy (we will discuss what that means in detail, later), then it won't necessarily be the exact number of rooms commanded. In addition, it will not move past either end of the hallway. An action of 0 will cause the robot to try to stay in its current location.
4. The state estimator does an *observation update* of its belief state, based on its old belief state and the observation. The update depends on the observation model, which specifies the probability distribution over observations for each state. This belief state is printed out.
5. The state estimator does a *transition update* of its belief state, based on the belief state that resulted from the observation update and the specified action. The update depends on the transition model, which specifies the probability distribution over next states, given the previous state and action.
6. The *squares are redrawn* with colors and numbers that reflect the probabilities in the new belief state.

- If you enter `quit` as an action, the whole machine terminates. It will stop after 10 steps unless you call `run` with a larger numeric argument.

  If you want to run the machine again, it's best to create a new instance; it's okay if you call `run` again on the old instance, but the initial belief display will be incorrect until it does one update.

> *Check Yourself 1.* Move the robot around in the perfect simulator. Be sure you understand what the colors representing the belief state mean and that the numbers being printed out in the Python shell make sense. Feel free to ask a staff member for clarification.

Step 2. The world you just created has perfect motion and perfect sensing. You can create and run one with noisy motion and sensing as follows:

```
n = makeNoisy()
n.run(20)
```

> *Check Yourself 2.* Move the robot around in the noisy simulator. Be sure you understand what the colors mean, and have a basic idea of what might be going on. Feel free to ask a staff member for clarification.

2.1 The observation model

Section 7.6 of the readings may help with understanding the next sections.

The observation model is a conditional probability distribution specifying what color the robot sees given what room it is in: $P(O_t = o_t | S_t = s_t)$. In our case o_t ranges over

```
('black', 'white', 'red', 'green', 'blue', 'purple', 'orange', 'darkGreen',
'gold', 'chocolate', 'PapayaWhip', 'MidnightBlue', 'HotPink', 'chartreuse')
```

and s_t ranges over all the possible rooms the robot could be in.

If there are m possible observations and n possible locations, then it will, in general, require $m \cdot n$ numbers to specify the observation model. In this problem, we make an assumption that makes the model much more compact: the robot's observation only depends on the *actual color* of the room it's in. That is, all rooms that are actually white have the same distribution over possible observations and all rooms that are actually green have the same distribution over possible observations (which will generally be different from the observation distribution for rooms that are actually white). Given this assumption, we only need to specify $P(\mathit{observedColor} \mid \mathit{actualColor})$, and then we can find the probability of observing each color in any room, as long as we know the actual color of that room:

$$P(O_t = \mathit{observedColor} \mid S_t = s_t) = P(O_t = \mathit{observedColor} \mid \mathit{ActualColor} = \mathit{actualColor}(s_t))$$

The **conditional probability distribution**

$$P(O_t = \mathit{observedColor} \mid \mathit{ActualColor})$$

which specifies a distribution on the observed color given the actual color of the robot's room, is called the *observation noise distribution*. We can specify an observation noise distribution in Python as a procedure that takes an actual color as input and returns a distribution on observed colors. Here is a very simple example that always observes the true color.

```
def perfectObsNoiseModel(actualColor):
    return dist.DDist({actualColor: 1.0})
```

Now, given an observation noise distribution `obsNoise`, such as `perfectObsNoiseModel`, we can construct the entire observation model (a conditional probability distribution on observed colors given robot location) as shown below:

```
def makeObservationModel(hallwayColors, obsNoise):
    return lambda loc: obsNoise(hallwayColors[loc])
```

Here, `hallwayColors` is a list specifying the true color of each location in the hallway, `loc` is an integer representing the location of the robot, and `obsNoise` is a conditional distribution of observed color given actual color. This procedure returns a conditional probability distribution, which is a procedure that takes a location as input and returns a distribution over observed colors.

The example world we have been using is specified with

```
standardHallway = ['white', 'white', 'green', 'white', 'white']
```

Given these procedures, we can specify the observation model for perfect observations with:

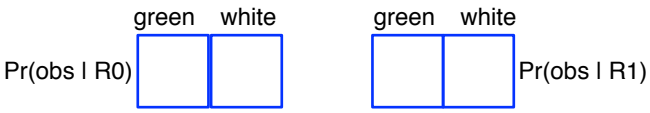
```
perfectObsModel = makeObservationModel(standardHallway, perfectObsNoiseModel)
```

Step 3.

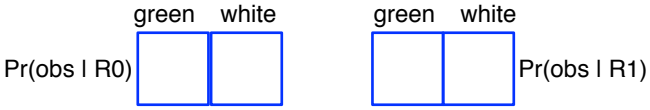
Wk.11.1.1 Do this tutor problem on defining observation models.

Check Yourself 3. Just to be sure you understand the observation models, consider a world with two rooms: room R_0 is actually green and room R_1 is actually white.

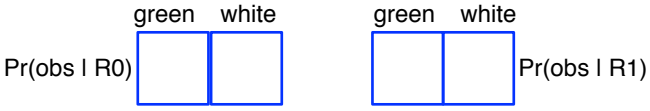
- With a perfect sensor, what is the probability distribution over observations for each room?



- With `whiteEqGreenObsDist`, what is the probability distribution over observations for each room?



- With `whiteVsGreenObsDist`, what is the probability distribution over observations for each room?



Step 4. Now consider a bigger test world, called `testHallway`. The rooms on each end have true color 'chocolate'. The others are either green or white.

Try out your `whiteEqGreenObsDist` and `whiteVsGreenObsDist` and see what happens to the belief state. Paste your definitions of those procedures from the tutor into your `designLab11Work.py` file. Use the perfect motion models, as shown below. **Be sure that you use something like `whiteEqGreenObsDist` as the third argument below.** (The variables `actions`, `standardDynamics`, and `perfectTransNoiseModel` are already defined for you in `designLab11Work.py`).

```
w = makeSim(testHallway, actions,
            <your observedColor given actualColor model>,
            standardDynamics, perfectTransNoiseModel)
w.run(50)
```

2.2 The state-transition model

The state-transition model is a conditional probability distribution over the state at time $t+1$, given the state at time t and the selected action a. That is, $P(S_{t+1} = s_{t+1} | S_t = s_t, A_t = a_t)$. 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: if n is the number of states of the world and m is the number of actions, then it would have size mn^2.

Often, the transition model can be described more sparsely or systematically. In this particular world, the robot can try to move some number of rooms to the right or left, 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 states are zero (there's no chance of the robot teleporting to the other end of the hallway, for example). This will allow us to describe the transition model more compactly.

We will start on defining the transition model by first defining a *dynamics* procedure, such as `standardDynamics` below. The dynamics is a procedure which returns the nominal new location resulting from taking action `act` in location `loc`, in a hallway with `hallwayLength` locations. This will be useful in other problems. The possible actions that the robot can take are: $-4, \ldots, -1, 0, 1, \ldots, 4$. So, we can just add the robot's action to its current location to get its nominal new location, except we have to be sure it doesn't move off of the edges of the world, so we use `util.clip` to keep the value from going below 0 or above `hallwayLength - 1`.

```
def standardDynamics(loc, act, hallwayLength):
    return util.clip(loc + act, 0, hallwayLength-1)
```

Next, we define a *noise model*, which is independent of location. It returns a distribution over the possible resulting locations given the nominal location that results from an action under the given dynamics. The simplest noise model assumes that transitions are perfect, so that the resulting location will be the nominal location.

```
def perfectTransNoiseModel(nominalLoc, hallwayLength):
    return dist.DDist({nominalLoc : 1.0})
```

Ultimately, we need to make a full transition model, which is a conditional probability distribution of the form $Pr(S_{t+1} \mid S_t, A_t)$. Because it is conditioned on two variables, we will represent it using nested procedures (representing conditional distributions). So, we'll think of it as something like $Pr(S_{t+1} \mid S_t \mid A_t)$, or, as a procedure that takes an a_t and returns a procedure that takes an s_t and returns a distribution over S_{t+1}.

Here is the basic form of the transition model that takes two procedures, a dynamics procedure and a noise model, and an integer indicating the length of the hallway.

```
def makeTransitionModel(dynamics, noiseDist, hallwayLength):
    return lambda act: lambda loc: noiseDist(dynamics(loc, act, hallwayLength),
                                             hallwayLength)
```

A perfect transition model for our standard hallway under the standard dynamics is constructed as follows:

```
perfectTransModel = makeTransitionModel(standardDynamics, perfectTransNoiseModel, 5)
```

Step 5.

> **Wk.11.1.2** Do this tutor problem on defining transition models.

Step 6. Now consider a test world that has only white rooms, and has noisy transitions and observations. We will initialize the state estimator with an initial belief state that assigns probability 1 to location 7 (and, of course, probability 0 to all other locations). The variable `sterile` specifies a hallway made up of 16 white rooms. You can create this world and state estimator with:

```
w = makeNoisyKnownInitLoc(7, sterile)
w.run(50)
```

Experiment with selecting action `0` several times in a row. What happens? What happens when you drive the robot around?

> *Checkoff 1.* How do `whiteEqGreenObsDist` and `whiteVsGreenObsDist` observation models (from Check Yourself 3) compare to:
> - A perfect sensor model
> - A sensor that always reads 'black' no matter what room it is in
>
> Demonstrate the world with noisy dynamics from Step 6 to a staff member and explain why it does what it does.

3 State estimation in the hallway world

Be sure you have read sections 7.5–7.7 of the readings very carefully.

We will build intuition of state estimation by doing some numerical examples of state estimation by hand. These are types of problems that we will expect you to be able to do in quizzes and exams.

Step 7.

> **Wk.11.1.4** Do this problem on state estimation in the hallway world.

Step 8.

> **Wk.11.1.5** Do this problem on state estimation in the hallway world.

4 Preparing to localize

In design lab 13, we will build a system that will allow a robot to 'localize' itself: that is, estimate its position in the world, given a map of the obstacles in the world and the ability to make local

sonar readings. These problems build up important concepts and components of the localization system.

Step 9.

Wk.11.1.6 Understanding sonar geometry.

Step 10.

Wk.11.1.7 Computing the ideal sonar readings for a pose of the robot.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011