Software Lab 12 *6.01 – Fall 2011*

I think therefore where am I

Goals: In this lab we look at the general state estimation algorithm. You will:

- Implement state estimation for general stochastic state machines using the procedure we **showed graphically in the lecture** and the readings.
- Prepare for using this algorithm to localize robots in hallways.

Resources: This lab should be done **individually**.
Do `athrun 6.01 getFiles`. The relevant file (in `~/Desktop/6.01/swLab12`) is

- `swLab12Work.py`: code for building and testing the state estimation algorithm.

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. We have seen examples of such systems, including:

- 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).

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 this software lab and the next design lab, we'll use basic **state estimation** to build a system that estimates the robot's pose, based on noisy sonar and odometry readings.

We'll start by building up some familiarity with **stochastic state machines**, then we'll build the core state estimation algorithm, then we'll look at the application of this algoritm in robot localization, in preparation for Design Lab 13.

2 Stochastic state machines

A state estimator needs to know a *model* of the system whose state it is trying to estimate. This model contains three crucial components:

- initial state distribution
- state transition model
- observation model

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].

You had practice, last week, specifying state transition and observation models for the hallway domain.

We will collect these three components together into instances of a Python class, called **`ssm.StochasticSM`**. It is described in **section 7.5 of the readings**: what matters most to us are the three attributes `startDistribution`, `transitionDistribution`, and `observationDistribution`.

Here is an example `ssm.StochasticSM` definition for a hallway world in which there are five rooms with colors white, white, green, white, white. The internal states are the integers 0 through 4 representing which room the robot is really in, the inputs are integers -4 through 4 representing an attempt to move that many squares, and the observations are strings representing colors.

```
def observationModel(s):
    # All of the rooms in our world have white walls, except for room
    # 2, which has green walls;  observations are wrong with
    # probability 0.1
    if s == 2:
        return dist.DDist({'green' : 0.9, 'white' : 0.1})
    else:
        return dist.DDist({'white' : 0.9, 'green' : 0.1})

def transitionModel(action):
    def transGivenA(oldS):
        # Robot moves to the nominal new location (that is, the
        # old location plus the action) with probability 0.8;  some
        # chance that it moves one step too little, or one step too
        # much.  Be careful not to run off the end of the world.
        nominalNewS = oldS + action
       # This is from Wk.11.1.2, Part 3
        d = {}
        dist.incrDictEntry(d, util.clip(nominalNewS, 0, 5), 0.8)
        dist.incrDictEntry(d, util.clip(nominalNewS+1, 0, 5), 0.1)
        dist.incrDictEntry(d, util.clip(nominalNewS-1, 0, 5), 0.1)
        return dist.DDist(d)
    return transGivenA

simpleHallway = ssm.StochasticSM(dist.UniformDist(range(5)),
                                 transitionModel,
                                 observationModel)
```

Stochastic state machines have a `transduce` method, which takes a list of inputs and generates a list of outputs. Because they are stochastic (probabilistic), however, several different runs with the same input sequence may generate different output sequences. This is because the starting state, each state transition, and each output is, in general, chosen probabilistically based on the specified distributions.

So, for example, 10 trips down the hallway, each of which is supposed to be made up of four steps to the right, might generate output like this:

```
>>> for i in range(10):
        print simpleHallway.transduce([1, 1, 1, 1])
['white', 'green', 'white', 'green']
['green', 'white', 'white', 'white']
```

```
['white', 'green', 'white', 'white']
['white', 'green', 'white', 'white']
['white', 'green', 'white', 'white']
['white', 'white', 'white', 'white']
['white', 'green', 'green', 'white']
['white', 'green', 'white', 'white']
['white', 'white', 'white', 'white']
['white', 'white', 'white', 'white']
```

Step 1.

> **Wk.12.2.1** Do this tutor problem to get practice with the types of the components of a stochastic state machine.

3 State Estimation

Now, we're going to take the cool-but-potentially-confusing step of making the state estimator. A state estimator is, itself, an instance of the **state machine class `sm.SM`**: it is a regular deterministic state machine. Its job is to take inputs that are pairs, `(o, i)`, where `o` is the observation that we made of some system at some time `t` and `i` is the input that was given to the system at time `t`. The internal state of the state estimator is a *belief state*, or probability distribution over the possible hidden states of the system whose internal state we are trying to estimate. The output of the state estimator will be the same as its internal state.

To make an instance of a state estimator, we pass in a model of the system whose state we are trying to estimate, in the form of a stochastic state machine. So, for example, if we wanted to try to estimate the robot's position in the simple hallway described above, we could make a state estimator:

```
hallwayLocalizer = se.StateEstimator(simpleHallway)
```

Now, `hallwayLocalizer` is a regular old state machine, and we can use familiar methods, like `transduce` on it. Here, we feed in a sequence of observation-action pairs, asking what we believe about the robot's location after it observes white, moves to the right one square, observes white again, moves to the right one square, observes green, and then stays where it is:

```
>>> hallwayLocalizer.transduce([('white', 1), ('white', 1), ('green', 0)])
[DDist(0: 0.024324, 1: 0.218919, 2: 0.221622, 3: 0.070270, 4: 0.464865),
 DDist(0: 0.003029, 1: 0.051496, 2: 0.224196, 3: 0.060546, 4: 0.660733),
 DDist(0: 0.002819, 1: 0.087084, 2: 0.581842, 3: 0.113220, 4: 0.215035)]
```

The result of `transduce` is a list of three belief states.

> *Check Yourself 1.* Be sure the results of transduce shown above make sense to you. If they don't, ask a staff member for clarification.

As we observed in **section 7.7.1 of the course readings**, the state estimation procedure can be seen as taking two steps:

1. Doing a step of Bayesian evidence updating, in which we update the current belief state based on the actual observation received, using the observation distribution stored in the `ssm.StochasticSM` that is the model of the underlying system.
2. Then, using that resulting intermediate belief, doing an update using the law of total probability to apply the transition distribution appropriate for the input that was given to the underlying system.

Given **the library of methods we have developed for operating on distributions**, this is very straightforward to implement in Python. The starting state of the state estimator is just the initial state distribution of the SSM. To get the next values (that is, the next belief state of the state estimator), we do a step of **Bayes evidence** with the observation and an application of the law of total probability with a transition distribution that depends on the input.

```
class StateEstimator(sm.SM):
    def __init__(self, model):
        self.model = model
        self.startState = model.startDistribution

   def getNextValues(self, state, inp):
        (o, i) = inp
        sGo = dist.bayesEvidence(state, self.model.observationDistribution, o)
        dSPrime = dist.totalProbability(sGo,
                                        self.model.transitionDistribution(i))
        return (dSPrime, dSPrime)
```

This code works fine, and is conceptually very simple, but each step is fairly inefficent. Our implementation of the `bayesEvidence` method (from **Wk.10.1.7**) actually creates a whole joint distribution between the state and observation, and then conditions on the observation, selecting a single row of the joint. We can make this more efficient by more closely mirroring the procedure we **showed graphically in the lecture** and the readings: multiplying the probability of each state s in the belief state by $\Pr(o \mid s)$, and then normalizing it.

In a similar way, **our implementation of `totalProbability`** (also from **Wk.10.1.7**) is also quite inefficient. It constructs the entire joint distribution between `s` (the state at time `t`) and `sPrime` (the state at time $t+1$), and then immediately marginalizes out `s`. Again, we can more closely emulate the procedure **shown graphically in the lecture** and the readings: making a new empty distribution over `sPrime`, then iterating over the states `s`, and adding, to each state `sPrime` the probability of being in state `s` times the probability of making a transition to state `sPrime`, given the input.

Your job is to implement the `StateEstimator.getNextValues` method. We would encourage you to write two internal helper procedures, one for Bayes evidence, and one for total probability, that have the same results as our old implementations, but which are more efficient.

Rules:

- Never access the dictionary of a distribution directly (use the `prob` method instead).
- Do not modify any of the distributions in the model. If you want a copy of a distribution, you can use the **`dictCopy` method of `DDist`** to copy the dictionary, and then use it to make a new `DDist`.
- Inside the tutor problem and the work file, the module `dist` has been imported; if you need to use functions or classes from there, you can access them with expressions like `dist.x`.

- The tutor will test the number of times you access the input probability distributions, and if your code is too inefficient (it makes too many accesses to the dictionaries) it will fail the check even if it generates the correct next belief.

  When you check your code in the tutor, the results of the test cases show the number of calls to the `prob` method of `dist.DDist`, followed by the actual result of state estimation. You don't have to have precisely the same number of calls as our solution, but it can't be too many more.

The tutor will test your code on the copy machine example from the readings:

```
transitionTable =
   {'good': dist.DDist({'good' : 0.7, 'bad' : 0.3}),
    'bad' : dist.DDist({'good' : 0.1, 'bad' : 0.9})}
observationTable =
   {'good': dist.DDist({'perfect' : 0.8, 'smudged' : 0.1, 'black' : 0.1}),
    'bad': dist.DDist({'perfect' : 0.1, 'smudged' : 0.7, 'black' : 0.2})}
copyMachine =
 ssm.StochasticSM(dist.DDist({'good' : 0.9, 'bad' : 0.1}),
                  lambda i: lambda s: transitionTable[s],
                  lambda s: observationTable[s])
obs = [('perfect', 'step'), ('smudged', 'step'), ('perfect', 'step')]
```

You can use these values for testing, but don't build any of them into your code. Your code should work for any stochastic state machine (the `model` argument). This example is provided only so you can understand the test cases. The variable `obs` is set to a sequence of observation, action pairs. In this copy-machine example, the action is ignored, but it has to be present in order to make all of the types match up correctly.

Step 2.

Wk.12.2.2 Do this tutor problem on how to implement the state estimator efficiently.

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. This next problem provides an overview of the components of the localization system.

Step 3.

Wk.12.2.3 Think about the localization process.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011