Design Lab 13 *6.01 – Fall 2011*

I Walk the Line

Goals: In this lab you implement a system for estimating the location of a robot as it moves down a hallway starting from an uncertain location. You will:

- Understand a preprocessor, which transforms sensor data into inputs for a state estimator
- Construct a state estimator and stochastic state machine with realistic observation and transition models for localization with the 6.01 robots
- Build a complete sytem demonstrating robot position localization in a one-dimensional world, within the `soar` simulator

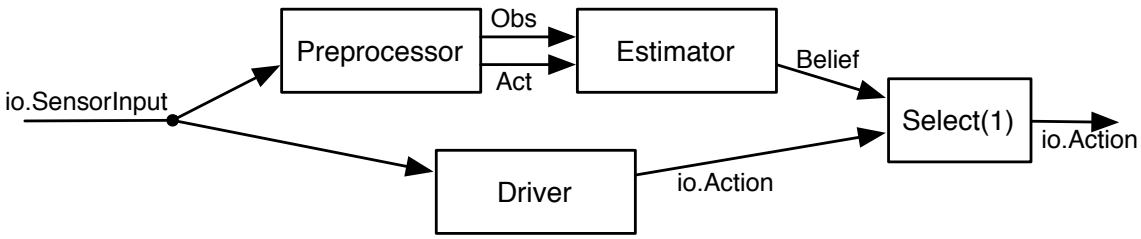
Resources: You can do the lab on any computer with soar. This lab should be done with a partner.
Do `athrun 6.01 getFiles`. The relevant files (in `~/Desktop/6.01/designLab13`) are
- `lineLocalizeSkeleton.py`: file to write your code in
- `lineLocalizeBrain.py`: brain file to run, to test robot localization

If you haven't done **Wk.12.2.3** and read **Wk.11.1.7** and **homework 4**, do that now.

1 Overview

In this lab, we will implement, and ultimately test in soar, a robot localizer, as outlined in tutor problems **Wk.11.1.7** and **Wk.12.2.3**. This lab will focus on localization in a one-dimensional world, but the basic ideas will allow you to implement a localizer for a two-dimensional world in a later lab.

Here is the architecture of the system we will construct.



The `Driver`, `PreProcessor` and `Select` state machine classes are already implemented. The `Driver` machine generates instances of `io.Action` that make the robot move forward; the `Select(1)` machine takes tuples (or lists) of values as input and always returns the second element of the tuple as output. The robot knows the ideal readings for each of the possible discrete locations it might be in, but doesn't know where it is initially; the goal of the state estimation process

is to determine the robot's location. The effect of this behavior is that the robot always drives forward, but the state estimation process is running in parallel, and as a side effect, the current belief state estimate of where the robot is in the world will be displayed in a window.

2 Preprocessor

The preprocessor module takes sensor data from the sonar and the odometer of the robot, and transforms them into input information about the observation made and action performed, for the state estimator. Tutor problem **Wk.12.2.3** describes the preprocessor module in detail. You will probably need to refer back to some of the definitions in that problem, so plan on keeping the Tutor page open.

In the file `lineLocalizeSkeleton.py` you will find the state machine class `PreProcess` that implements the preprocessor module. The code is reproduced below:

```
class PreProcess(sm.SM):
    def __init__(self, numObservations, stateWidth):
        self.startState = (None, None)
        self.numObservations = numObservations
        self.stateWidth = stateWidth
    def getNextValues(self, state, inp):
        (lastUpdatePose, lastUpdateSonar) = state
        currentPose = inp.odometry
        currentSonar = idealReadings.discreteSonar(inp.sonars[0],
                                                   self.numObservations)
        if lastUpdatePose == None:
            return ((currentPose, currentSonar), None)
        else:
            action = discreteAction(lastUpdatePose, currentPose,
                                    self.stateWidth)
            print (lastUpdateSonar, action)
            return ((currentPose, currentSonar), (lastUpdateSonar, action))
# Only works when headed to the right
def discreteAction(oldPose, newPose, stateWidth):
    return int(round(oldPose.distance(newPose) / stateWidth))
```

Step 1. Be sure that you understand the implementation of the preprocessor machine.

Check Yourself 1. What is the internal state of the machine?

What is the starting state?

Step 2. Test the preprocessor on the example from tutor problem **Wk.12.2.3** as follows:

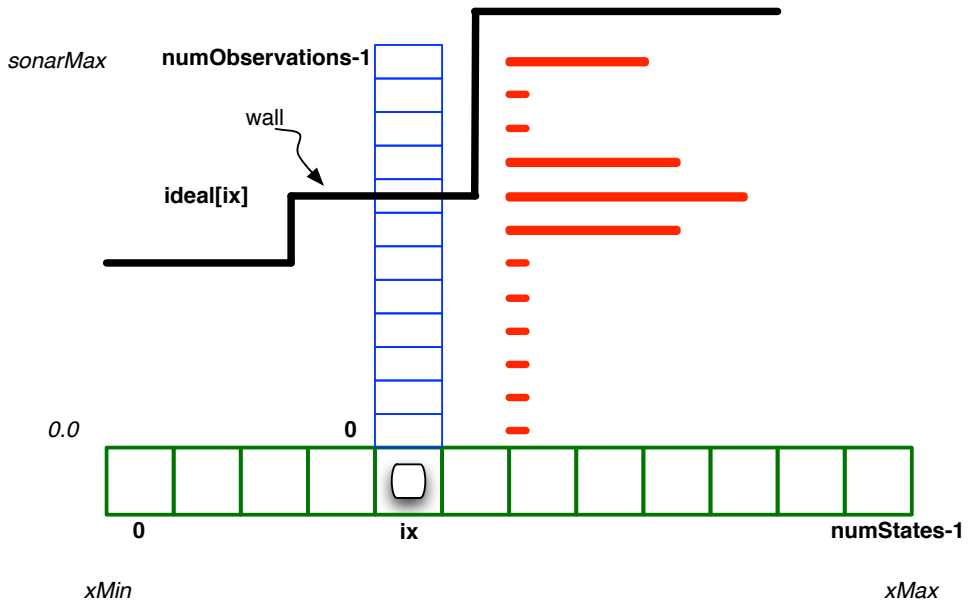
- Run the `lineLocalizeSkeleton.py` file in Idle.
- Make an instance of the preprocessor machine, called `pp1`, using parameters that match the tutor problem: 10 discrete observation values, 10 discrete location values, `xMin = 0.0` and `xMax = 10.0` (this means that the state width is 1.0 in this example).
- Do `pp1.transduce(preProcessTestData)`.
- Make sure the outputs match the ones from the tutor problem.

Note that the `PreProcess` machine print its output on each step.

3 State Estimator

The estimator module in our architecture will be an instance of **`seGraphics.StateEstimator`**, which we have already written; it's just like the state estimator you wrote in **Wk.12.2.2**, but it displays the current belief state and observation probabilities in a pair of windows. Whenever we make an instance of a state estimator, we have to pass in an instance of **`ssm.StochasticSM`**, which describes what we know about the system whose hidden state we are trying to estimate. Our job, in this section of the lab, is to create the appropriate `ssm.StochasticSM`, with an initial **belief distribution, an observation model, and a transition model**, for the robot localization problem. The state that we are trying to estimate is the *discretized* x coordinate of the robot's location, which can be in the range `0` to `numStates - 1`.

You may find it helpful to refer to the following figure which provides a schematic illustration of a robot with discretized position, observing discretized sonar readings from a staggered wall next to it.



Bold labels are discrete state and observation indices.
Italic labels are real-valued robot locations or sonar readings.
Dark line is actual location of a wall.
Green boxes are possible discrete robot locations.
Blue boxes are possible discrete sonar readings.
Red bars indicate the distribution observationDistribution(ix).

The file `lineLocalizeSkeleton.py` contains the following skeleton of a procedure that should construct and return the appropriate `ssm.StochasticSM` model. The parameters are:

- `ideal`: a list of **discretized** ideal sonar readings, of length `numStates`
- `xMin, xMax`: the minimum and maximum x coordinates the robot can travel between
- `numStates`: the number of discrete states into which the x range is divided
- `numObservations`: the number of discrete observations

```
def makeRobotNavModel(ideal, xMin, xMax, numStates, numObservations):
    startDistribution = None
    def observationModel(ix):
        pass
    def transitionModel(a):
        pass
    return ssm.StochasticSM(startDistribution, transitionModel, observationModel)
```

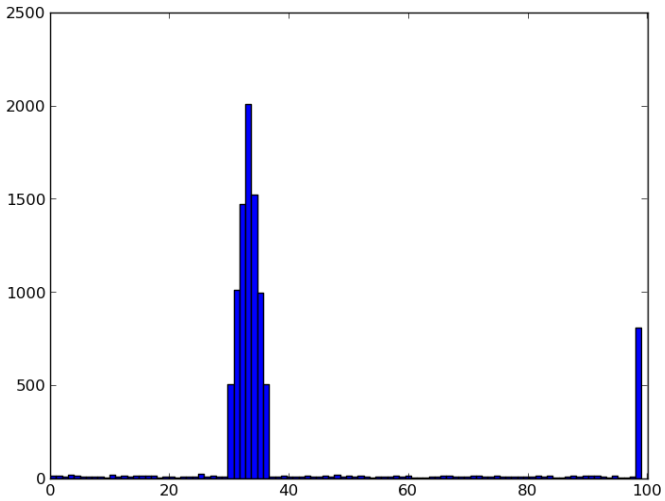
3.1 Initial distribution

Step 3. Define `startDistribution`, which should be uniform over all possible discrete robot locations. You can create a uniform distribution over a range of integers with `dist.squareDist(lo, hi)`.

3.2 Observation model

The observation model is a **conditional probability distribution**, represented as a procedure that takes a state (discrete robot location) as input and returns a distribution over possible observations (discrete sonar readings). Our job is to create an observation model that characterizes the distribution of sonar readings that are likely to occur when the robot is in a particular location.

This figure shows a histogram of 10,000 sonar readings generated in a situation in which there were 100 possible discrete sonar values over the range 0 to 1.5 m and where the ideal sonar reading was 0.5 m. The x axis is the discrete sonar reading and the y axis is the number of readings (out of 10,000) that fell into that interval.



It has the following features:

- There is always a non-trivial likelihood of getting an observation at the maximum range (due to reflections, etc). The maximum value is `sonarDist.sonarMax`.
- It is most likely to get an observation at the ideal distance, but there might be small relative errors in the observation (that is, we might see an object at 0.88 meters when it's really at 0.9 meters).
- There is some small chance of making any observation (due to someone walking by, etc.).

Pay particular attention to the 'width' of the noise distribution. It is important to write your mixture models so they are sensitive to the discretization granularity of the sonar readings: with the same amount of noise in the real world, the width in terms of the number of bins will be different for different granularities.

You should use `dist.MixtureDist`, `dist.triangleDist`, `dist.squareDist`, and `dist.DeltaDist` to construct a distribution that describes well the data shown in histogram.

Check Yourself 2. Sketch out your plan for the observation model. Be sure you understand the type of the model and the mixture distributions you want to create. Ask a staff member if you're unsure on any of these points.

Step 4. Implement the observation model and test it to be sure it's reasonable. It doesn't need to match the histogram in the figure exactly.

For debugging, you can create a model (which is a `ssm.StochasticSM`) like this:

```
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
```

Then you can get the observation conditional probability distribution like this:

```
model.observationDistribution
```

Here, `testIdealReadings` is the same set of ideal readings from tutor problem **Wk.12.2.3**.

Debug your distributions by plotting them, being sure that you have started Idle with `-n`. If `d` is a distribution you've created, you can plot it with `distPlot.plot(d)`.

If `observationModel` is your observation model, using the readings in `testIdealReadings`, write down the 4 highest-probability entries in `observationModel(7)` (this is an instance of `DDist`). What does the 7 stand for here?

Step 5. Now, make a model for the case with 100 observation bins, instead of 10.

```
model100 = makeRobotNavModel(testIdealReadings100, 0.0, 10.0, 10, 100)
```

Plot the observation distribution for robot location 7 in `model` and `model100`. Be sure they are consistent and correct.

Checkoff 1. **Wk.13.2.1**: Show your observation distribution plots to a staff member and explain what they mean.

3.3 Transition model

The transition model is a conditional probability distribution, represented as a procedure that takes an action as input and returns a procedure; that procedure takes a starting state (discrete robot location) as input, and returns a distribution over resulting states (discrete robot locations). You can compute the next location that would result if there were no error in odometry, and then return a distribution that takes into account the fact that there might be errors in the robot's reported motion.

For now, the only error in the transitions is due to discretization of the reported actions. Think about what discrete locations the robot could possibly have moved to, given a reported action of having moved k discrete locations. Use a triangle distribution to model the discretization error.

Check Yourself 3. Sketch out your plan for the transition model. Be sure you understand the type of the models and the distributions you want to create. Ask a staff member if you're unsure on any of these points.

Step 6. Implement the transition model and test it to be sure it's reasonable. Create a `ssm.StochasticSM`, and then get the transition model (which is a procedure that returns a conditional probability distribution) like this:

```
model = makeRobotNavModel(testIdealReadings, 0.0, 10.0, 10, 10)
model.transitionDistribution
```

If `transitionModel` is your transition model, write down `transitionModel(2)(5)` (this is an instance of `DDist`). What do the 2 and 5 stand for here? Be sure the result makes sense to you.

3.4 Combined preprocessing and estimation

Step 7. Now we'll put the two modules we just made together and be sure they work correctly. Use `sm.Cascade` to combine an instance of your `PreProcess` class and an instance of the `seGraphics.StateEstimator` class, which is given your `ssm.StochasticSM` model, using 10 discrete observation values, 10 discrete location values, xMin = 0.0, and xMax = 10.0. Call this machine `ppEst`.

Check Yourself 4. Do `ppEst.transduce(preProcessTestData)`. Compare the result to the belief states in **Wk.12.2.3**. Remember that you are now assuming noisy observations and noisy actions. Are your results consistent with the ones you found in the tutor?

Checkoff 2. **Wk.13.2.2**: Show your answers to the questions above to a staff member. Explain what they mean.

4 Putting it All Together

Now, we'll put all the machines together to make a behavior that can control the robot. The file `lineLocalizeBrain.py` contains all the scaffolding necessary. It makes one call that you need to think about:

```
robot.behavior = \
    lineLocalize.makeLineLocalizer(numObservations, numStates, ideal, xMin, xMax, y)
```

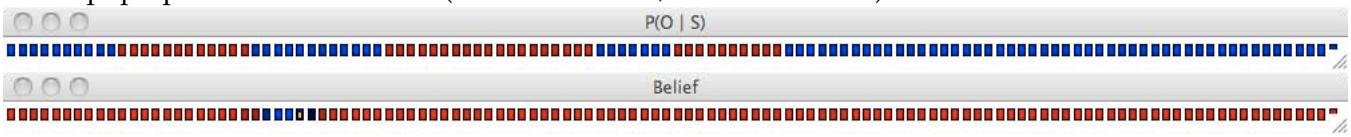
Step 8. In your `lineLocalizeSkeleton.py` file, implement the procedure `makeLineLocalizer` with the arguments shown above; it should construct a complete robot behavior, as outlined in the architecture diagram, whose inputs are `io.SensorInput` instances and whose outputs are `io.Action` instances. Read about the `sm.Select` state machine in the software documentation.

You will need instances of the preprocessor and estimator machines like those you made in the previous section, together with the driver state machine. The driver is a state machine whose input is an instance of `io.SensorInput` and whose output is an instance of `io.Action`. You can create it with

```
move.MoveToFixedPose(util.Pose(xMax, robotY, 0.0), maxVel = 0.5)
```

assuming that the robot starts at some location with `y` coordinate `robotY`, and will move to the right until its `x` coordinate is `xMax`.

Step 9. Start soar and run your behavior using `lineLocalizeBrain.py` in the world `worlds/oneDdiff.py`. It will pop up windows like these (to see the colors, look at it online):



The first window shows, for each state (possible discrete location of the robot), how likely the current observation is in that state. In this example, the robot's current observation is one that is likely to be observed when it is in any of the locations that is colored blue, and unlikely to be observed in the locations colored red. The second window shows the current belief state, using colors to indicate probabilities. Black is the uniform probability, brighter blue is more likely, brighter red

is less likely. The actual location of the robot is shown with a small gold square in the belief state window.

Use the **step** button in soar to move the robot step by step and look at and understand the displays. It is necessary to move the robot two steps before the displays become interesting.

Step 10. Now run your behavior in the world `oneDreal.py` (you will need to edit the line in `lineLocalizeBrain.py` that selects the world file, as well as select a new simulated world in soar). What is the essential difference between this world and `oneDdiff.py`?

Step 11. Now run your behavior in the world `oneDslope.py`, **without changing the world file selected in the brain**. This will mean that the robot **thinks** it is in the world `oneDreal.py`, and has observation models that are appropriate for that world, but it is, instead, in an entirely different world. What happens when you run it? What do the displays mean?

Checkoff 3. **Wk.13.2.3**: Demonstrate your running localization system to a staff member. Explain the meanings of the colors in the display windows and argue that what your system is doing is reasonable. Explain why the behavior differs between `oneDreal` and `oneDdiff`. Explain what happens when there is a mismatch between the world and the model.

5 If you're interested in doing more...

Here are some possible extensions to this lab.

5.1 Real robot

Try your localizer on the real robot. You'll need to:

- Set `maxVel` to 0.1
- Take out the `discreteStepLength` call from the brain.
- Change `cheatPose` to `False`
- Change the `y` value of the target pose for the driver to 0.0
- Use boxes covered with bubble wrap to set up a world that corresponds to `oneDreal.py`.
- (Possibly) adjust the amount of noise in your model of the sonar and the motion error.

5.2 Handle Teleportation

Add this code to your brain file:

```
teleportProb = 0.0
import random
class RandomPose:
    def draw(self):
        x = random.random()*(xMax - xMin) + xMin
        return (x, y, 0.0)
io.enableTeleportation(teleportProb, RandomPose())
```

If you set `teleportProb` to a value greater than 0, it will, with that probability, on each motion step, 'teleport' the robot to an `x` coordinate chosen uniformly at random from the robot's `x` range. (maintaining the same heading and `y` coordinate). This is a good way to test your localization.

If necessary, modify your transition distribution so that it can cope with a world in which the robot might teleport. Think about what parameter in your model should match `teleportProb`.

Turn up the teleportation probability and see if your robot can cope.

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011