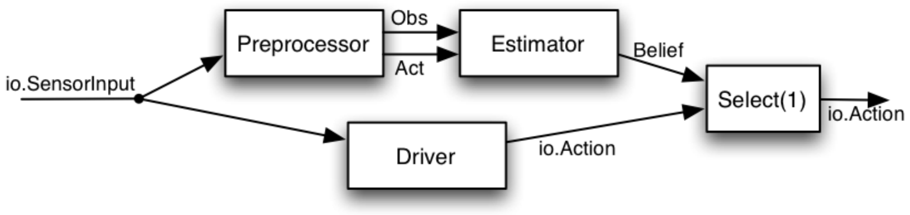
# Problem Wk.12.2.3: Localization

Earlier (in Wk.11.1.7) you considered how to compute the ideal sonar readings for a world. Please refer to that problem for a reminder about the set-up of a robot moving along a hallway and using a single side sonar to estimate its position in the hallway. Now we will look at the overall structure of the localization system. After you familiarize yourself with the structure, we'll ask you to hand simulate a simple case.

Here is the architecture of the system:



The key modules in the system are:

- **Preprocessor**: This state machine takes, as input, instances of `io.SensorInput` and generates as output pairs of `(observation, action)`, suitable for input to the state estimator.
- **Estimator**: This state machine is an instance of `seGraphics.StateEstimator`, which is, essentially, the state estimator you implemented previously.
- **Driver**: This state machine is an instance of the `move.MoveToFixedPose` class. It can be used to move the robot forward in a straight line.

Here is a description of the key parameters for the system (with the values we'll use in this tutor problem):

- `numObservations = 10`: a positive integer describing the number of discrete values we will divide the sonar range into. We will assume that sonar readings are in the range 0.0 to `sonarMax` which is the maximum reliable sonar readings.
- `numStates = 10`: a positive integer describing the number of discrete values we will divide the robot's possible *x* coordinate into. We will assume that robot's *x* position are in the range `xMin` to `xMax`.
- `xMin = 0.0`: the smallest possible *x* coordinate of the robot in the world (in meters).
- `xMax = 10.0`: the largest possible *x* coordinate of the robot in the world (in meters)
- `y = 0.5`: a fixed *y* coordinate of the robot in the world (in meters).
- `sonarMax = 1.5`: the maximum sonar reading (in meters).

We are interested in estimating the robot's *x* coordinate, based on its sequence of observations and actions. So, we will treat the state space in the same way as the observations, and divide the robot's possible range of *x* coordinates into `numStates` fixed intervals, and let the *state* of the robot be the index of the interval into which its true *x* coordinate falls. Let *w* be the 'width' of the intervals in *x* space that constitute states.

## Preprocessor

The preprocessor is a state machine with these inputs and outputs:

- **Input:** Instance of `io.SensorInput`.

- **Output:** Pair of `(obs, act)`, each of which is an integer index. `obs` is the index of a discrete sonar reading and `act` is the index of a discrete motion.

To make this problem fit into our state estimation framework, we will *discretize* the sensor readings by dividing the range of possible sonar values into `numObservations` fixed-length intervals, and encoding the value as the index of the interval into which it falls.

Because we would like our state estimator to run in parallel with, but not depend directly on, another machine that is controlling the robot's actions, we will infer an 'action' based on observing the robot's odometry. Recall that the odometry readings are relative to the arbitrary location where the robot started in the world (which we don't know). So, the odometry is only good for estimating *relative displacements*.

Let $x_t$ be the robot's observed $x$ coordinate at time $t$. We will say that the 'action' that the robot took at time $t-1$ is $x_t - x_{t-1}$. We need to discretize the action space, so we will actually report the action as a rounded number of intervals: it is $int(round((x_t - x_{t-1}) / w))$, where $w$ is the width of a state interval interval (that is, the range of possible $x$ coordinates divided by the number of discrete states).

Because our state estimation process incorporates an observation and then the subsequent transition, **it is crucial that the preprocessor output an observation from step *t-1* and the action that took place between steps *t-1* and *t*.**

When the state machine is first started there will not be any previous observation or odometry available, so we will simply generate an output of `None`. We will make a special modification in our state estimator, so that if the input to the machine is `None`, then no state update will be made at all.

## Estimator

The `Estimator` is an instance of a state machine that acts as a state estimator; its inputs and outputs are:

- **Input:** Either `None` or a pair of `(obs, act)`, each of which is an integer index. `obs` is the index of a discretized sonar reading at time *t* and `act` is the index of a discretized motion started at time *t*.
- **Output:** Belief state, represented as a `dist.DDist` over possible discrete *x* locations of the robot in the world. If the input is `None`, then the output belief state is the same as the belief state from the previous time step (or the initial state, if this is the first time step).

To make the estimator state machine, it's necessary to construct a `ssm.StochasticSM` that contains the initial distribution, and transition and observation models for the localization problem. We can let the starting distribution be uniform.

The starting state of the state estimator is just the initial belief state of the SSM. And to get the next values, we do a step of Bayes evidence with the observation and an application of the law of total probability with the observation.

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

    def getNextValues(self, state, inp):
        if inp == None:  return (state, state)
        (o, i) = inp
        sGo = dist.bayesEvidence(state, self.model.observationDistribution, o)
        dSPrime = dist.totalProbability(sGo,
                                        self.model.transitionDistribution(i))
        return (dSPrime, dSPrime)
```

### Observation model

Remember that the observation model is a conditional distribution, that is, a procedure that takes a state as input and returns a distribution over possible observations. The state will be one of our discrete state indices.

In this problem, assume a *perfect* observation model, that is, in each state, the sonar returns the *discretized* ideal reading for the state (you wrote code to compute these readings in the previous problem).

### Transition model

Similarly, the transition model is a procedure that takes an action as input and returns a procedure; that procedure takes a starting state as input, and returns a distribution over resulting states.

Assume a *perfect* transition model, that is, given an action $\Delta$ (an integer), and a state $s$ (also an integer), the resulting state is $s + \Delta$ but clipped to stay within the legal range of $0 \le s < numStates$.

**REMINDER**: Because our state estimation process incorporates an observation and then the subsequent transition, **it is crucial that the preprocessor output an observation from step *t-1* and the action that took place between steps *t-1* and *t*.**

In this problem, we will assume that the ideal (discretized) sonar readings for each state are:

```
ideal = ( 5, 1, 1, 5, 1, 1, 1, 5, 1, 5 )
```

Fill in the output values for the `Preprocessor` and `Estimator` state machines for the first three time steps below. For non-zero probabilities, please enter 3 digits after the decimal (enter decimal values or fractions).

**If the output of the Preprocessor is `None`, enter `None` in both boxes here; but when you implement this state machine in design lab 13, your machine should output a single None.**

1. `Preprocessor (at time 0):`
    - Input: an instance of `io.SensorInput`:
        - `sonars = (0.8, 1.0, ...)`
        - `odometry = Pose(1.0, 0.5, 0.0)`
    - Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
        - obs =
        - act =
2. `Preprocessor (at time 1):`
    - Input: an instance of `io.SensorInput`:
        - `sonars = (0.25, 1.2, ...)`
        - `odometry = Pose(2.4, 0.5, 0.0)`

- Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
  - `obs =`
  - `act =`

3. `Preprocessor (at time 2):`

   - Input: an instance of `io.SensorInput`:
     - `sonars = (0.16, 0.2, ...)`
     - `odometry = Pose(7.3, 0.5, 0.0)`
   - Output: a tuple `(obs, act)`; if the output is `None`, enter `None` in both boxes.
     - `obs =`
     - `act =`

4. `Estimator (at time 0):`

   - Input: `(obs, act)` the output tuple from Preprocessor
   - Output: probability distribution over robot states (x indices)

5. `Estimator (at time 1):`

   - Input: `(obs, act)` the output tuple from Preprocessor
   - Output: probability distribution over robot states (x indices)

6. `Estimator (at time 2):`

   - Input: `(obs, act)` the output tuple from Preprocessor
   - Output: probability distribution over robot states (x indices)