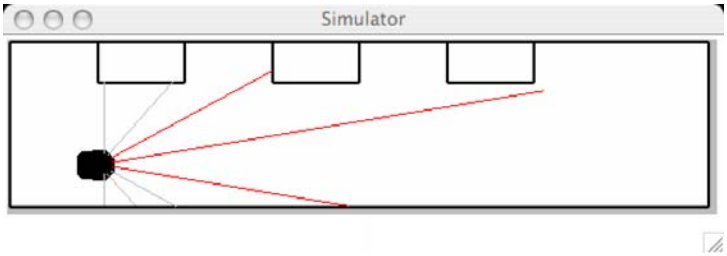
Problem Wk.11.1.7: Ideal Sonar Readings

We call the problem of figuring out where a robot is in the world, given a known map, but unknown starting location, the problem of **localization**. We can frame the problem as one of doing state estimation. If we specify a model of how our robot interacts with the world, in the form of the definition of a *ssm.StochasticSM*, and feed that into the state estimator, then the state estimator will continually update a probability distribution representing its beliefs about the location of the robot given the actions and observations taken so far.

In Design Lab 13 we are going to build a simple system in which the robot starts out facing to the right, and moves along a straight line to the right, using just the values of its left (number 0) sonar sensor to get information.



Part of the robot model is an error model for the sonar readings. Our error model for sonar readings will have a similar structure to the error model for observing colors in the hallway: we determine the sonar observation we would expect to see in each state, if our observations were perfect, and then use that as the basis for constructing a distribution over possible observations.

In this problem, we consider how to compute the ideal sonar readings. We are given:

- `wallSegs`: A list of line segments representing the walls in the simulated world. Each line segment is an instance of the `util.LineSeg` class (look at the software documentation for this class).
- `robotPoses`: A list of poses (instances of `util.Pose` class) of the robot, corresponding to the center of each of the discretized robot states.
- `sonarHit(distance, sonarPose, robotPose)`: a procedure that takes the length of a line segment emanating from a sonar sensor (`dist`), the pose of the sensor in the robot's frame (represented as a `util.Pose`), and the pose of the robot in the global frame (represented as a `util.Pose`), and returns a `util.Point` representing the position of the end of the sonar segment in the global frame. This function is already defined.
- `sonarPose0`: the pose of sonar sensor 0 with respect to the robot is given as an instance of `util.Pose`.
- `sonarMax`: the maximum meaningful value of a sonar reading; any distance longer than this should be represented as `sonarMax`.
- `numObservations`: the sonar readings (from 0 to `sonarMax`) will be discretized into integers between 0 and `numObservations-1`.

Implement two procedures:

1. `discreteSonar` discretizes sonar readings: it takes a sonar reading as input and generates an integer bin index as ouput. Sonar readings will range between 0 and `sonarMax`. This range should be discretized into a fixed number, `numObservations`, of bins. Note that the bin indices are from 0 to `numObservations-1`. Any sonar reading

greater than `sonarMax` should be placed into the last bin.

We strongly suggest that you draw a picture, with the interval from 0 to 1 divided into three bins. Consider what answers you would want to return for inputs of 0.32 and 0.34. Here are some useful things to know about Python:

- The `round` procedure takes a floating point number and returns the nearest whole number, as a float. So, `round(2.8)` returns `3.0` and `round(2.1)` returns `2.0`.
- The `int` procedure takes a floating point number and returns just its integer part, as an integer. So, `int(2.1)` returns `2`.

2. `idealReadings` takes a list of wall segments describing the world and a list of robot poses, corresponding to each of the states of the system, and returns **a list of discretized ideal sonar readings for sonar 0, one for each state**. For each state:
   - Determine the line segment starting at the point where sonar sensor 0 is currently located in the world, and going out in the direction of the sensor for length `sonarMax`.
   - Find the point of intersection between that line segment and any wall segment in the world that is closest to sonar sensor 0.
   - Compute the ideal sonar reading, which is the distance from the sensor to that closest point of intersection. You can use the `intersection` method of `util.LineSeg`, which returns the `util.Point` of intersection between two line segments, if there is one; it returns `False` if there isn't one. If there is no intersection, then the ideal sonar reading is `sonarMax`.
   - Compute a discretized value for the ideal sonar reading.

   Return the list of discretized ideal readings, one for each state.

Debug this in Idle and paste your answer here.

One test case is:

```
def wall((x1, y1), (x2, y2)):
   return util.LineSeg(util.Point(x1,y1), util.Point(x2,y2))
wallSegs = [wall((0, 2), (8, 2)),
            wall((1, 1.25),(1.5, 1.25)),
            wall((2, 1.75),(2.8, 1.75))]
robotPoses = [util.Pose(0.5, 0.5, 0), util.Pose(1.25, 0.5,0),
              util.Pose(1.75, 1.0, 0), util.Pose(2.5, 1.0, 0)]
```

```
sonarMax = 1.5
numObservations = 10
sonarPose0 = util.Pose(0.08, 0.134, 1.570796)

def discreteSonar(sonarReading):
    pass

def idealReadings(wallSegs, robotPoses):
    pass
```

6.01SC Introduction to Electrical Engineering and Computer Science

Spring 2011