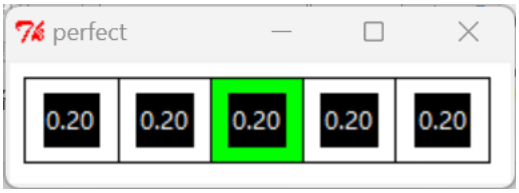
Step1

Check Yourself 1.

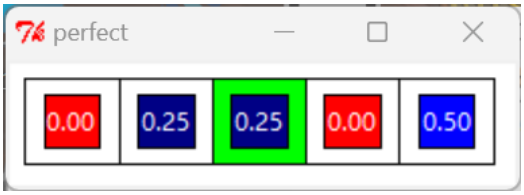
In the initial state, the robot does not know its position and does not observe the color of the room. In all rooms, the probability is equal and it is reflected as black.



The robots are randomly distributed in a room. After we give the first command, the first thing the robot does is make an observation about the color of the room. In this perfect model, where there is no noise of any kind, the robot gives a belief state after observing the color of the room it is in. It should be noted that this belief state is based on the previous belief state, and then the robot calculates the new belief state after executing the instruction.

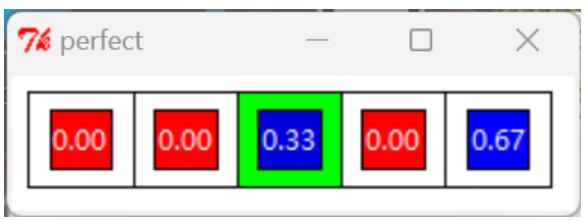
```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.250000, 1: 0.250000, 3: 0.250000, 4: 0.250000)
after trans 1 DDist(1: 0.250000, 2: 0.250000, 4: 0.500000)
```

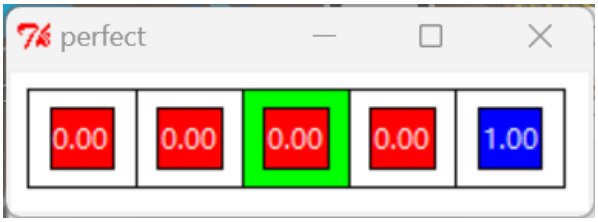
In the initial state, the robot observes that it is in a room with a true color of white, and gives its belief state, that is, all the white rooms are equally divided in probability, and then gives a new belief state based on the observed color after perfectly executing the instruction that we input to move one step to the right.



By parity of reasoning:

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(1: 0.333333, 4: 0.666667)
after trans 1 DDist(2: 0.333333, 4: 0.666667)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(4: 1.000000)
after trans 1 DDist(4: 1.000000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): quit
Taking action 0 before quitting
after obs white DDist(4: 1.000000)
after trans 0 DDist(4: 1.000000)
[('white', 1), ('white', 1), ('white', 1), ('white', 0)]
```





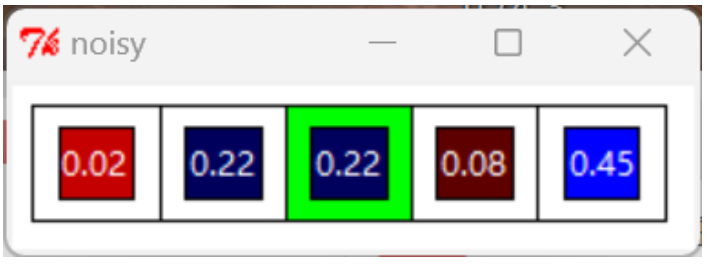
The quit command indicates that we expect the car to complete the last color observation to get the belief state, exit without action, and output each color observed by the robot itself and the motion instructions we input by a list.

Step2

Check Yourself 2.

In a system with noise, the meaning of data printed by our python shell is exactly the same as that in the ideal model, except that there is noise in our observation model and motion model, and each corresponding state obtained is a probabilistic statistical model

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.248804, 1: 0.248804, 2: 0.004785, 3: 0.248804, 4: 0.248804)
after trans 1 DDist(0: 0.024880, 1: 0.223923, 2: 0.224402, 3: 0.078469, 4: 0.448325)
```



In the initial state, we let the robot move one unit to the right, and we could not get the probability that the percentage is at the leftmost end is

0, which proves that our trans model is not perfect

```
>>> n = makeNoisy()
>>> n.run(20)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.248804, 1: 0.248804, 2: 0.004785, 3: 0.248804, 4: 0.248804)
after trans 1 DDist(0: 0.024880, 1: 0.223923, 2: 0.224402, 3: 0.078469, 4: 0.448325)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.031901, 1: 0.287113, 2: 0.005533, 3: 0.100612, 4: 0.574840)
after trans 1 DDist(0: 0.003190, 1: 0.054232, 2: 0.233434, 3: 0.100683, 4: 0.608460)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs green DDist(0: 0.000247, 1: 0.004202, 2: 0.940600, 3: 0.007802, 4: 0.047149)
after trans 1 DDist(0: 0.000025, 1: 0.000618, 2: 0.097447, 3: 0.758395, 4: 0.143515)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(0: 0.000027, 1: 0.000683, 2: 0.002072, 3: 0.838536, 4: 0.158681)
after trans 1 DDist(0: 0.000003, 1: 0.000090, 2: 0.000757, 3: 0.101448, 4: 0.897703)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs red DDist(0: 0.000003, 1: 0.000090, 2: 0.000757, 3: 0.101448, 4: 0.897703)
after trans 1 DDist(0: 0.000000, 1: 0.000011, 2: 0.000148, 3: 0.100529, 4: 0.899311)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): quit
Taking action 0 before quitting
after obs white DDist(0: 0.000000, 1: 0.000011, 2: 0.000003, 3: 0.100544, 4: 0.899442)
after trans 0 DDist(0: 0.000001, 1: 0.000009, 2: 0.010058, 3: 0.170380, 4: 0.819552)
[('white', 1), ('white', 1), ('green', 1), ('white', 1), ('red', 1), ('white', 0)]
```

Red, which does not exist in the actual true color, appears in the room color observation, which confirms that there are errors in our observation model

Step3

Wk.11.1.1

Part 1: White == Green

Define an observation noise model (that is, a conditional distribution over the observed color given the actual color of a room) in which white and green are indistinguishable, in the sense that the robot is just as likely to see white as to see green when it is in a white square or a green square, but where all other colors are observed perfectly. This function should return a `dist.DDist` over observed colors, given the actual color passed in as an argument.

```
#无法区分白色和绿色并且看到绿色和白色的概率相等。
def whiteEqGreenObsDist(actualColor):

    #在绿色或者白色的方形里，返回绿色和白色的概率分别是0.5。

    if actualColor == 'white' or actualColor=='greem':
        return dist.DDist({white: 0.5, green: 0.5})
    else:                              #在其他颜色的房间里返回房间的实际颜色，即实际颜色的概率为1。
        return dist.DDist({actualColor: 1})
```

Part 2: White <-> Green

Give an observation noise distribution in which white always looks green, green always looks white, and all other colors are observed perfectly.

```
#将白色看成绿色。将绿色看成白色。
def whiteVsGreenObsDist(actualColor):
    if actualColor == 'white':        #白色房间内看到绿色，返回绿色的概率为1。
        return dist.DDist({green:1})
    if actualColor == 'green':        #绿色房间看到白色，返回白色的概率为1。
        return dist.DDist({white:1})
    else:                             #其他房间看到真实颜色，真实颜色的概率为1。
        return dist.DDist({actualColor:1})
```

Part 3: Noisy

Define an observation noise distribution in which there is a probability of 0.8 of observing the actual color of a room, and there is a probability of 0.2 of seeing one of the remaining colors (equally likely which other color you see). All possible colors are available in the list called `possibleColors`.

```
#0.8的概率观察到真实颜色，0.2的概率观察到其他颜色且每种颜色被观察到的概率相等。
def noisyObs(actualColor):
    d={}
    for Colors in possibleColors:    #0.2的概率观察到possibleColors中的剩余颜色
        d[Colors] = (1.0-0.8)/(len(possibleColors)-1.0)    #先对possibleColors中的颜色都赋以概率，概率的值为
                                                           #0.2/possibleColors中的非真实颜色数，即总颜色数-1
    d[actualColor] = 0.8             #再重新覆盖，定义看到真实颜色的概率为0.8。
    return dist.DDist(d)
```

Part 4: Observation Models

The observation noise distributions that you have defined so far encode the error model for observation in a way which is independent of location. But, ultimately, we need to be able to obtain a probability distribution over the possible observations (colors) at a particular location along the hallway. The function `makeObservationModel` allows us to construct such a model, given a list of colors, such as `standardHallway`, and an observation noise distribution, such as you've written above.

Enter the expression for creating the full observation model for the `standardHallway` (assume it's already defined), with the observation noise distribution `noisyObs` (assume it's already defined).

```
noisyObsModel = makeObservationModel(standardHallway, noisyObs)
```

Check Yourself 3.

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?

	green	white
Pr(obs | R0)	1	0

	green	white
Pr(obs | R1)	0	1

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

	green	white
Pr(obs | R0)	0.5	0.5

	green	white
Pr(obs | R1)	0.5	0.5

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

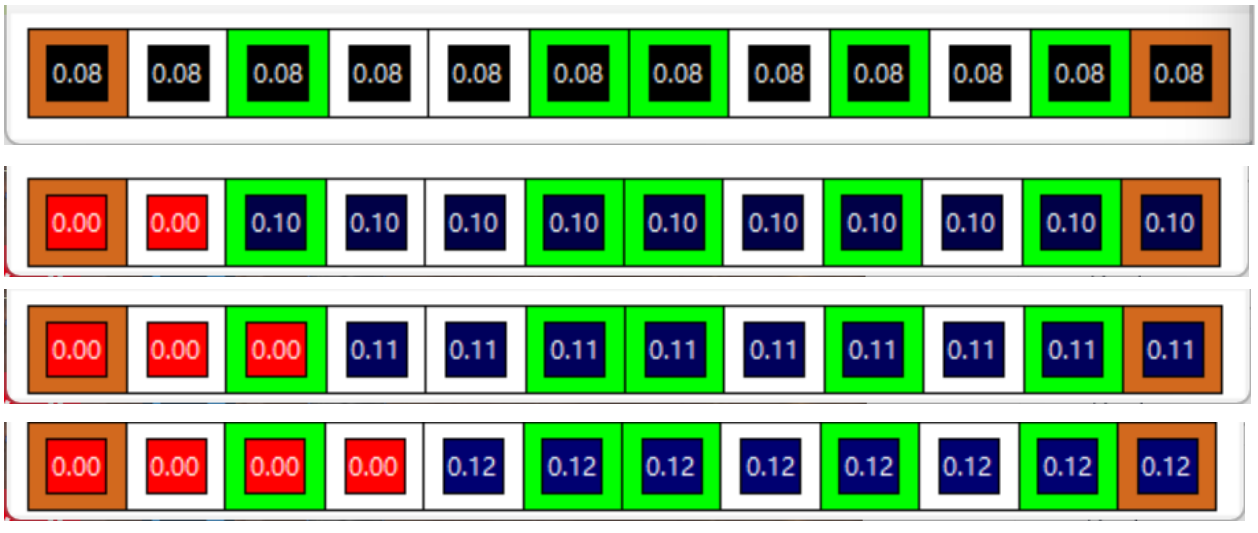
	green	white
Pr(obs | R0)	0	1

	green	white
Pr(obs | R1)	1	0

Step4

Our state trans model is perfect at this point

1. In using whiteEqGreenObsDist, we actually showed that the robot could not correctly distinguish the meaning of green and white. It can be imagined that for the robot, "green" and "white" are actually a separate color that does not belong to the. In the shell, it is obvious that all of our green rooms and white rooms are equally likely without moving to the defined 'chocolate' color end (where the robot can be located directly)




```
>>> w = makeSim(testHallway, actions,
whiteEqGreenObsDist,
standardDynamics, perfectTransNoiseModel)
>>> w.run(50)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(1: 0.100000, 2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000)
after trans 1 DDist(2: 0.100000, 3: 0.100000, 4: 0.100000, 5: 0.100000, 6: 0.100000, 7: 0.100000, 8: 0.100000, 9: 0.100000, 10: 0.100000, 11: 0.100000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs green DDist(2: 0.111111, 3: 0.111111, 4: 0.111111, 5: 0.111111, 6: 0.111111, 7: 0.111111, 8: 0.111111, 9: 0.111111, 10: 0.111111)
after trans 1 DDist(3: 0.111111, 4: 0.111111, 5: 0.111111, 6: 0.111111, 7: 0.111111, 8: 0.111111, 9: 0.111111, 10: 0.111111, 11: 0.111111)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(3: 0.125000, 4: 0.125000, 5: 0.125000, 6: 0.125000, 7: 0.125000, 8: 0.125000, 9: 0.125000, 10: 0.125000)
after trans 1 DDist(4: 0.125000, 5: 0.125000, 6: 0.125000, 7: 0.125000, 8: 0.125000, 9: 0.125000, 10: 0.125000, 11: 0.125000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(4: 0.142857, 5: 0.142857, 6: 0.142857, 7: 0.142857, 8: 0.142857, 9: 0.142857, 10: 0.142857)
after trans 1 DDist(5: 0.142857, 6: 0.142857, 7: 0.142857, 8: 0.142857, 9: 0.142857, 10: 0.142857, 11: 0.142857)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs chocolate DDist(11: 1.000000)
after trans 1 DDist(11: 1.000000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): quit
Taking action 0 before quitting
after obs chocolate DDist(11: 1.000000)
after trans 0 DDist(11: 1.000000)
[('white', 1), ('green', 1), ('white', 1), ('white', 1), ('chocolate', 1), ('chocolate', 0)]
>>>
```

2. In using whiteVsGreenObsDist, we show that the robot will always perceive the true color "green" as white, and the corresponding true color "white" as green.

```
>>> w = makeSim(testHallway, actions,
whiteVsGreenObsDist,
standardDynamics, perfectTransNoiseModel)
>>> w.run(50)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(8: 0.200000, 10: 0.200000, 2: 0.200000, 5: 0.200000, 6: 0.200000)
after trans 1 DDist(11: 0.200000, 9: 0.200000, 3: 0.200000, 6: 0.200000, 7: 0.200000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ):
```

After the initialization of the analysis robot, it is observed that the room color is white. Then according to our logic, the initial state of the robot is actually a room with a true color of "green", corresponding to the probability positions of rooms 2, 5, 6, 8 and 10 are exactly the same as those printed

We believe that this does not affect the output and position judgment in this relatively symmetrical structure

Step5

Wk.11.1.2

Part 1: Living on a donut

Write a ``ring'' dynamics model, in which the room 0 is connected to room `hallwayLength-1`.

```
def ringDynamic(loc, act, hallwayLength):
    if loc+act >=0 and loc+act <= hallwayLength-1 :   #如果loc+act没有超出0至hallwayLength-1，返回三者中间值
        return util.clip(loc + act , 0 , hallwayLength - 1)
    elif loc+act > hallwayLength-1:     #如果(loc+act)>hallwayLength-1,相当于向右运动并且至hallwayLength-1向右的下一步从跳跃到0
                                        #(loc+act)是假如hallwayLength是无限时的坐标，其与hallwayLength的余数即为有限hallwayLength实际坐标。
        return (loc+act) % hallwayLength

    else:       #如果(loc+act)<0，相当于向左运动并且至0向左的下一步从跳跃到hallwayLength-1
                #其方法与上一步类似，但因为是从右向左运动所以实际坐标应该用hallwayLength减去余数。
        return hallwayLength-abs(loc+act) % hallwayLength
```

Part 2: Slipping to the left

Write the ***left slip*** noise model, in which, with probability 0.1, the robot lands one square to the **left** of its nominal location and otherwise lands at its nominal location. It should not be allowed to transition off the end of the world, though: if it is nominally at the leftmost location, it should stay there with probability 1.

```
def leftSlipTrans(nominalLoc, hallwayLength):
    if nominalLoc >0 :      #当理论位置大于0时，由于向左滑动概率为0.1，所以在理论位置概率为0.9，在其左边位置的概率为0.1
        return dist.DDist({nominalLoc:0.9, nominalLoc-1:0.1}
    else:                   #当理论位置小于等于0时，由于无法向左移动，所以留在0处的概率为1。
        return dist.DDist({nominalLoc:1})
```

Part 3: Noisy Transitions

Write a transition noise model in which the robot lands one square to the left of where it should be with probability 0.1, one square to the right with probability 0.1, and in the nominal square with probability 0.8. It should not be allowed to transition off either end of the world; any probability associated with a square off the end of the hallway should be associated instead with the square at that end of the hallway.

```
def noisyTrans(nominalLoc, hallwayLength):
    if nominalLoc >0 and nominalLoc <hallwayLength-1:   #当理论位置在非中间时刻的位置时，返回在理论位置的概率为0.8，在理论位置左右的概率各为0.1。
        return dist.DDist({nominalLoc:0.8,nominalLoc-1:0.1,nominalLoc+1:0.1})
    elif nominalLoc == hallwayLength-1:     #当理论位置在最右边时，由于无法向右运动，所以在其左边位置的概率为0.1，在其理论位置的概率为0.9。
        return dist.DDist({nominalLoc:0.9,nominalLoc-1:0.1})
    else:       #当理论位置在最左边时，由于无法向左运动，所以在其右边位置的概率为0.1，在其理论位置的概率为0.9。
        return dist.DDist({nominalLoc:0.9,nominalLoc+1:0.1})
```

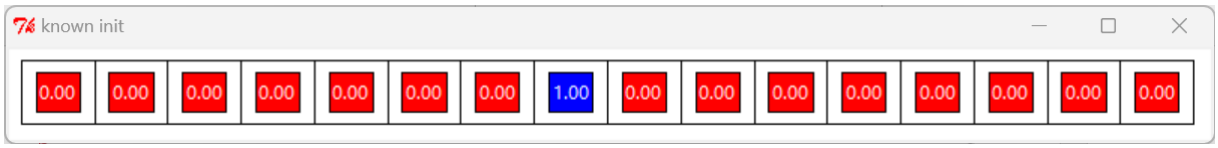
Part 4: Transition Models

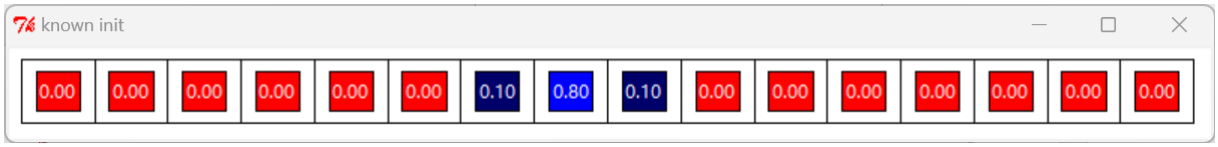
The function `makeTransitionModel`, described in the handout, puts together the dynamics and noise model to construct a full transition model for a hallway of a given length.

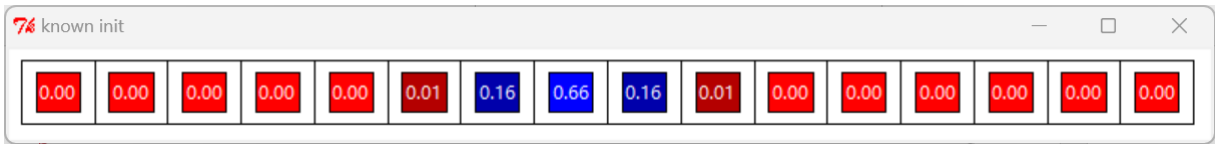
Enter the expression for creating the full transition model for the `standardDynamics` and the `noisyTrans` (assume it's already defined) in the `standardHallway` (assume it's already defined) which is 5 squares long.

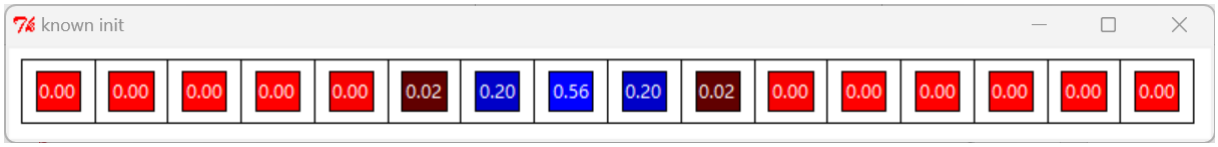
```
noisyTransModel = makeTransitionModel(standardDynamics, noisyTrans, standardHallway)
```

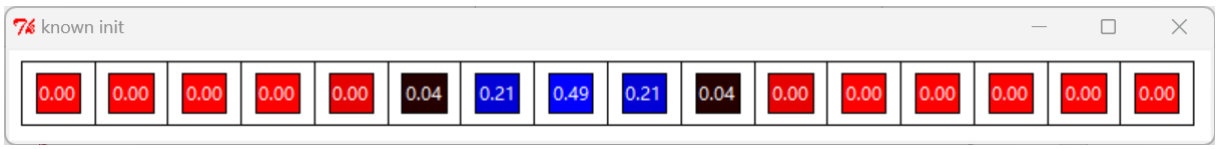
Step6







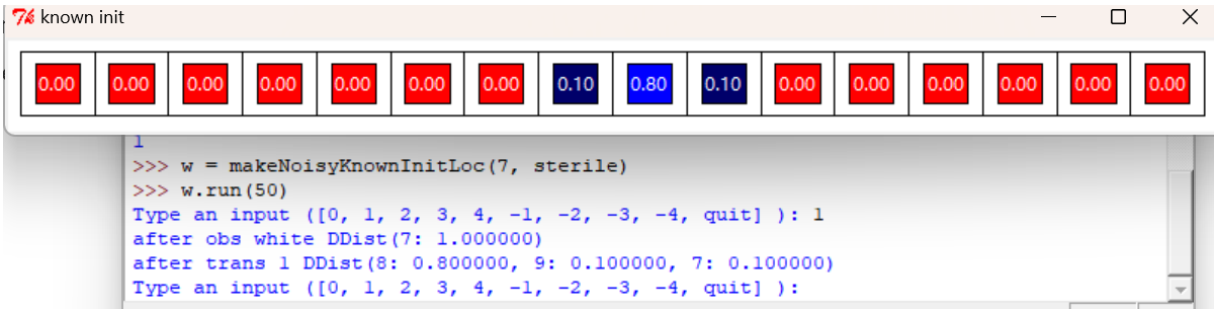




```
>>>
>>> w = makeNoisyKnownInitLoc(7, sterile)
>>> w.run(50)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(7: 1.000000)
after trans 0 DDist(8: 0.100000, 6: 0.100000, 7: 0.800000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(8: 0.100000, 6: 0.100000, 7: 0.800000)
after trans 0 DDist(8: 0.160000, 9: 0.010000, 5: 0.010000, 6: 0.160000, 7: 0.660000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(8: 0.160000, 9: 0.010000, 5: 0.010000, 6: 0.160000, 7: 0.66000
0)
after trans 0 DDist(4: 0.001000, 5: 0.024000, 6: 0.195000, 7: 0.560000, 8: 0.195000,
9: 0.024000, 10: 0.001000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 0
after obs white DDist(4: 0.001000, 5: 0.024000, 6: 0.195000, 7: 0.560000, 8: 0.19500
0, 9: 0.024000, 10: 0.001000)
after trans 0 DDist(3: 0.000100, 4: 0.003200, 5: 0.038800, 6: 0.214400, 7: 0.487000,
8: 0.214400, 9: 0.038800, 10: 0.003200, 11: 0.000100)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): quit
Taking action 0 before quitting
after obs white DDist(3: 0.000100, 4: 0.003200, 5: 0.038800, 6: 0.214400, 7: 0.48700
0, 8: 0.214400, 9: 0.038800, 10: 0.003200, 11: 0.000100)
after trans 0 DDist(2: 0.000010, 3: 0.000400, 4: 0.006450, 5: 0.052800, 6: 0.224100,
7: 0.432480, 8: 0.224100, 9: 0.052800, 10: 0.006450, 11: 0.000400, 12: 0.000010)
[('white', 0), ('white', 0), ('white', 0), ('white', 0), ('white', 0)]
>>>
```

```
[('white', 0), ('white', 0), ('white', 0), ('white', 0), ('white', 0), ('white', 0),
('white', 0), ('white', 0), ('white', 0), ('white', 0), ('white', 0), ('white', 0),
('darkGreen', 0), ('white', 0), ('white', 0), ('gold', 0)]
```

Since there is noise in our transformation model, even if the input "0" is given, our robot is still likely to drift left and right. After repeated accumulation, the robot will be distributed in space, rather than keeping precisely at the origin



A similar principle is followed when the robot moves from side to side

```
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs white DDist(8: 0.160000, 9: 0.010000, 5: 0.010000, 6: 0.160000, 7: 0.660000)
after trans 1 DDist(5: 0.001000, 6: 0.024000, 7: 0.195000, 8: 0.560000, 9: 0.195000, 10: 0.024000, 11: 0.001000)
Type an input ([0, 1, 2, 3, 4, -1, -2, -3, -4, quit] ): 1
after obs red DDist(5: 0.001000, 6: 0.024000, 7: 0.195000, 8: 0.560000, 9: 0.195000, 10: 0.024000, 11: 0.001000)
after trans 1 DDist(5: 0.000100, 6: 0.003200, 7: 0.038800, 8: 0.214400, 9: 0.487000, 10: 0.214400, 11: 0.038800, 12: 0.003200, 13: 0.000100)
```

It's worth noting that our belief states didn't change if the robot read colors that weren't in the actual room because of the noise

Checkoff 1.

For A sensor that always reads 'black' no matter what room it is in，This is completely unusable for us.

The whiteEqGreenObsDist is actually a slightly unreliable perfect sensor that combines two colors into one. The whiteVsGreenObsDist is a sensor that affects the operation of a system in an asymmetric structure. But both are available and even useful under certain conditions.

Step7

Wk.11.1.4

Part 1: Perfect sensor, perfect action

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states s?
   $B_0(s) = Pr(S_0 = s)$
$\frac{1}{3}$	$\frac{1}{3}$	$\frac{1}{3}$

2. First, the robot makes an observation. Let's assume it sees 'white', because it is in a white room and there's no sensor noise. So, $O_0 = white$. We want to know what is the new belief state after this observation?
   - First, figure out $Pr(O_0 = white|S_0 = s)$ for each state s
1	0	1
   - Then compute $Pr(O_0 = white|S_0 = s)Pr(S_0 = s)$ for each state s. Note that this is the same as $Pr(O_0 = white, S_0 = s)$
$\frac{1}{3}$	0	$\frac{1}{3}$
   - Now, compute $Pr(O_0 = white)$.
$\frac{2}{3}$
   - Compute the new belief state after the observation; using the definition of conditional probability and the previous two results:
     $B'_0(s) = Pr(S_0 = s|O_0 = white)$.
$\frac{1}{2}$	0	$\frac{1}{2}$

3. If we told the robot to go right one room with action $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
0	$\frac{1}{2}$	$\frac{1}{2}$

4. Now, assume the robot observes 'green' because it's in a green room and there's no noise, $O_1 = green$. What will the belief state be after this?
   $B'_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = green)$
0	1	0

5. If we told the robot to go right with action $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $B_2(s) = Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$
0	0	1

Part 2: Noisy sensor, Perfect action

1. What is the robot's prior belief for each of the states *s*?
$B_0(s) = Pr(S_0 = s)$
$\frac{1}{3}$ $\frac{1}{3}$ $\frac{1}{3}$

2. What is the distribution over what the robot sees? That is, what is $Pr(O_0 = o)$ for all possible colors *o*?
$P(O_0 = black) = \frac{1}{20}$ $P(O_0 = white) = \frac{11}{20}$ $P(O_0 = red) = \frac{1}{20}$
$P(O_0 = green) = \frac{3}{10}$ $P(O_0 = blue) = \frac{1}{20}$

3. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white$). We want to know the new belief state after the observation.
$B_0'(s) = Pr(S_0 = s|O_0 = white)$
$\frac{16}{33}$ $\frac{1}{33}$ $\frac{16}{33}$

4. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account? Recall that motion is perfect.
$B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
0 $\frac{16}{33}$ $\frac{17}{33}$

5. Now, what is the distribution over what the robot sees? That is, what is $Pr(O_1 = o)$ for all possible colors *o*?
$P(O_1 = black) = \frac{1}{20}$ $P(O_1 = white) = \frac{264}{455}$ $P(O_1 = red) = \frac{1}{20}$
$P(O_1 = green) = \frac{91}{220}$ $P(O_1 = blue) = \frac{1}{20}$

6. Now, assume the robot sees 'white' again, $O_1 = white$). What will the belief state be after this?
$B_1'(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = white)$
0 $\frac{1}{18}$ $\frac{17}{18}$

7. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account?
$Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$
0 0 1

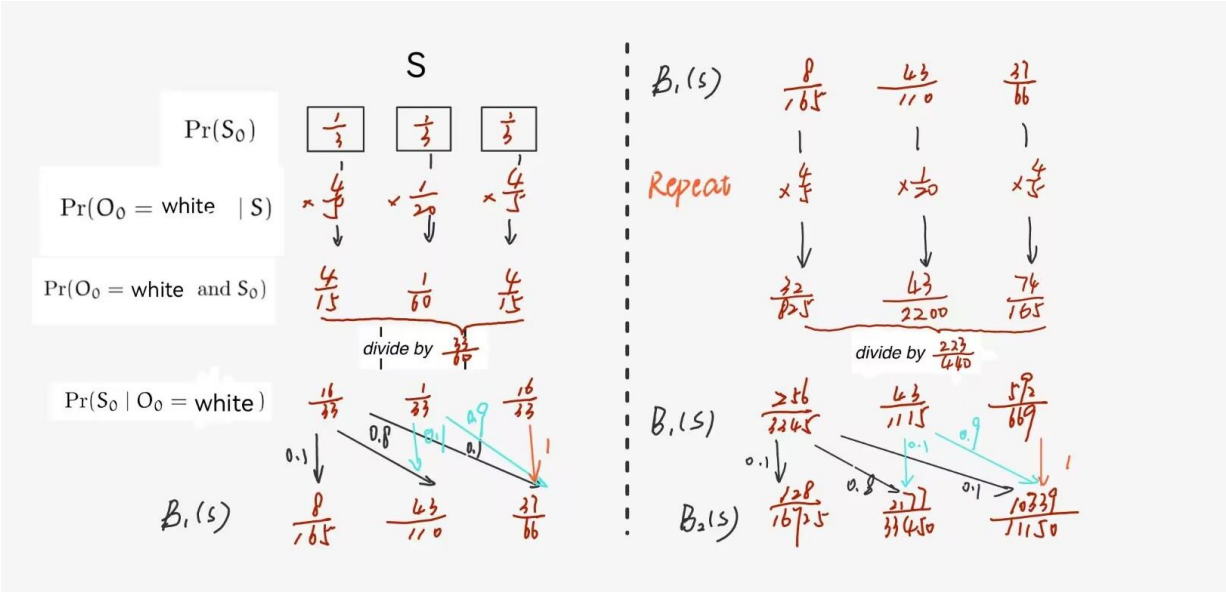
8. If instead of having seen 'white' and gone right (as above), the robot had seen 'green' and gone right, what would the belief state be? That is, what is
$Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = green, I_1 = 1)$
0 0 1

Step8

Wk.11.1.5

1. What is the robot's prior belief $B_0(s) = Pr(S_0 = s)$ for each of the states s?
   $B_0(s) = Pr(S_0 = s)$
   $\frac{1}{3}$ $\frac{1}{3}$ $\frac{1}{3}$
2. First, the robot makes an observation. Let's assume it sees 'white'. So, $O_0 = white$. We want to know the new belief state after the observation
   $B_0'(s) = Pr(S_0 = s|O_0 = white)$
   $\frac{16}{33}$ $\frac{1}{33}$ $\frac{16}{33}$
3. If we told the robot to go right $I_0 = 1$, what would the belief state be after taking the state transition into account?
   $B_1(s) = Pr(S_1 = s|O_0 = white, I_0 = 1)$
   $\frac{8}{165}$ $\frac{43}{110}$ $\frac{37}{66}$
4. Now, assume the robot sees 'white' again, $O_1 = white$. What will the belief state be after this?
   $B_1'(s) = Pr(S_1 = s|O_0 = white, I_0 = 1, O_1 = white)$
   $\frac{256}{3345}$ $\frac{43}{1115}$ $\frac{592}{669}$
5. If we told the robot to go right $I_1 = 1$, what would the belief state be after taking the state transition into account?
   $B_2(s) = Pr(S_2 = s|O_0 = white, I_0 = 1, O_1 = white, I_1 = 1)$
   $\frac{128}{16725}$ $\frac{2177}{33450}$ $\frac{10339}{11150}$

Thought of calculation：



Step9

Preparing to localize

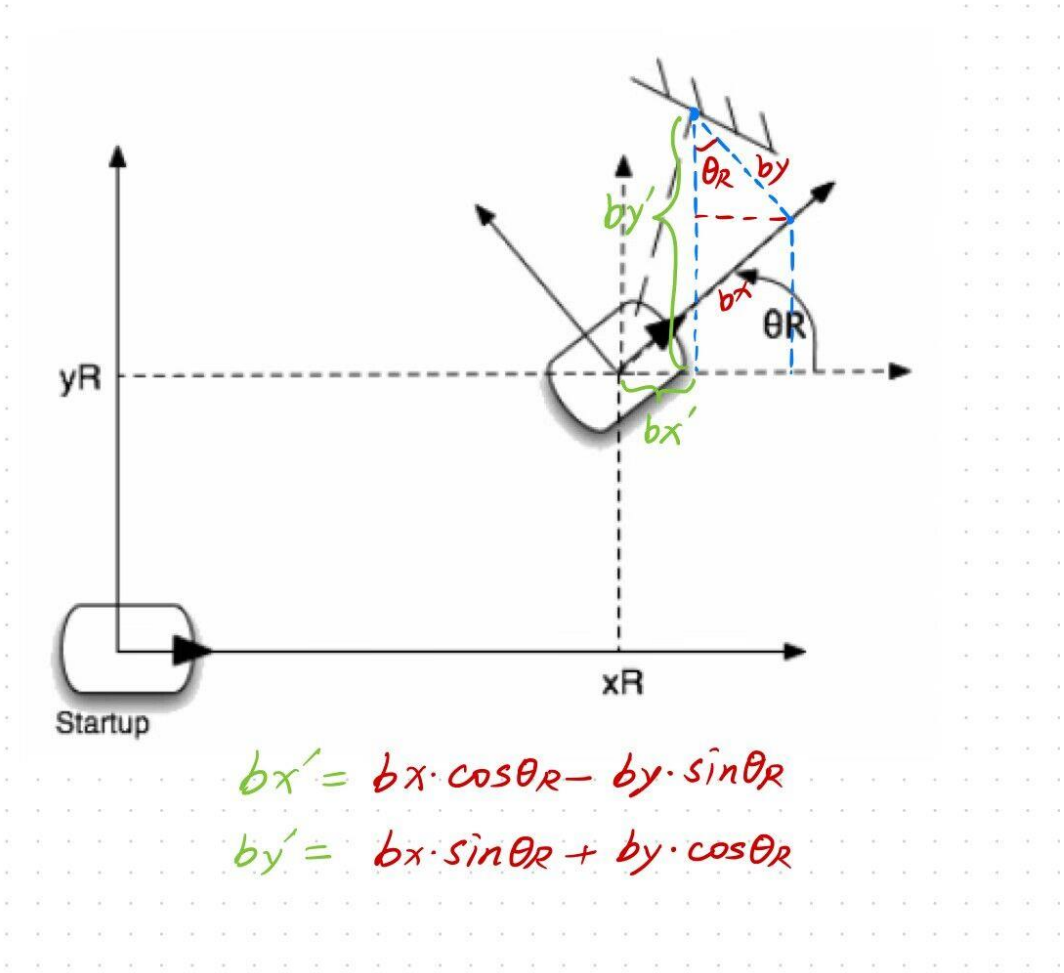
Wk.11.1.6

The target of this problem:Given a distance measurement from one of the sonars, return an instance of Point (look at the documentation of the util module), in the global odometry frame, the same coordinate frame that the robot's pose is measured, representing where the sonar beam bounced off an object.

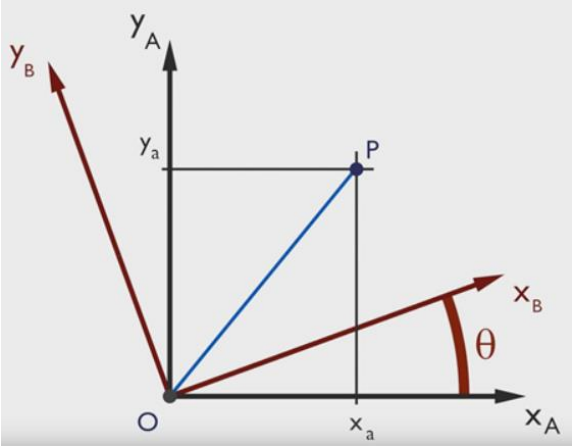
1. Understand the coordinate change（The coordinate change formula has been given in WK.11.1.6）

```
ax = xB + cos(thetaB)*bx - sin(thetaB)*by
ay = yB + sin(thetaB)*bx + cos(thetaB)*by
```

(1)Based on the geometry:



(2) Based on the rotation matrix



In two-dimensional space, rotation can be defined by a single angle. As a convention, a positive angle represents a clockwise rotation. The matrix that rotates the column vector of Cartesian coordinates counterclockwise with respect to the origin is:

$$M(\theta) = \begin{bmatrix} \cos\theta & -sin\theta \\ sin\theta & \cos\theta \end{bmatrix} = \cos\theta \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} + \sin\theta \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} = \exp\left(\theta \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix}\right)$$

2. Write the function 'sonarHit':

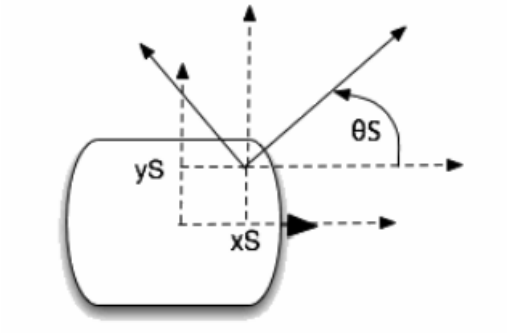
```
def sonarHit(distance,sonarPose,robotPose):
    relative_x=sonarPose.x+distance*math.cos(sonarPose.theta)
    relative_y=sonarPose.y+distance*math.sin(sonarPose.theta)
    target_point=util.Point(relative_x,relative_y)
    return robotPose.transformPoint(target_point)
```

Explanation:

(1)

Relative_x **->** bx Relative_y **->** by.

sonarPose.x **->**xS sonarPose.y**->**yS



$$bx = xS + d * cos\theta,\ by = yS + d * sin\theta$$

(2)

transformPoint(self, point)

Applies the pose.transform to point and returns new point.

Parameters:

- `point` - an instance of `util.Point`

transformPoint(self,Point) is a method of class 'Pose'.

robotPose.transformPoint(p) returns an instance of the Point class, the value of Point.x and Point.y is equal to applying the unit rotation matrix to the x,y coordinates of p (the rotation angle is pose.theta), and translating the rotated coordinates (robotPose.x, robotPose.y) units in a two-dimensional coordinate system. Through this method, the coordinate value of the object detected by the sonar under the global odometry frame can be obtained.

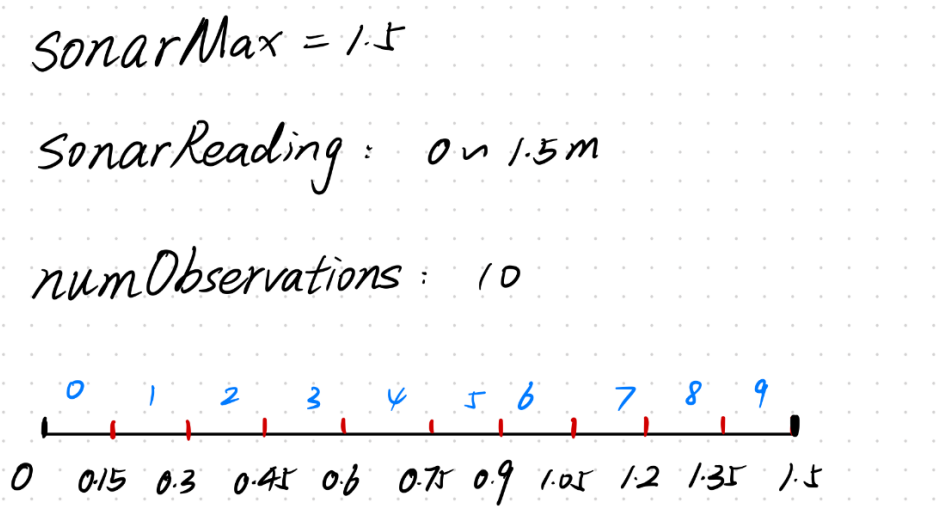
Step10

Wk.11.1.7

Ideal Sonar Readings:

1. discreteSonar:

```
def discreteSonar(sonarReading):
    binWidth=sonarMax/numObservations
    if sonarReading>=sonarMax:
        return numObservations-1
    else:
        return int(sonarReading/binWidth)
```



Range:0~1.5 should be discretized into a fixed number, numObservations, of bins. And the bin indices are from 0 to numObservations-1(0~9). When the sonarReading is within 0~1.5m, it will fall into any bin with an indice from 0 to 9. "int(sonarReadings/binWidth)" can represent the index value corresponding to a sonar reading. For example, when sonarReadings=1.4, int(sonarReadings/binWidth)=9. So, sonarReadings=1.4 corresponds to the discreted ideal sonarReadings "9"

2.idealReadings:

```
def idealReadings(wallSegs,robotPoses):
    bin_length=sonarMax/numObservations
    sonarline=[]
    idealreading=[]
    for i in range(len(robotPoses)):
        d=[]
        p1=sonarHit(0,sonarPose0,robotPoses[i])
        p2=sonarHit(sonarMax,sonarPose0,robotPoses[i])
        sonarline.append(util.LineSeg(p1,p2))
        for k in range(len(wallSegs)):
            inter_point=sonarline[i].intersection(wallSegs[k])
            print inter_point
            if inter_point:
                d.append(p1.distance(inter_point))
            else:
                d.append(sonarMax)
        idealreading.append(discreteSonar(min(d)))
    return idealreading
```

Explanation for my code:

bin_length defines the width of the bin.

sonarLine=[], idealreadings=[] create two empty lists for later use.

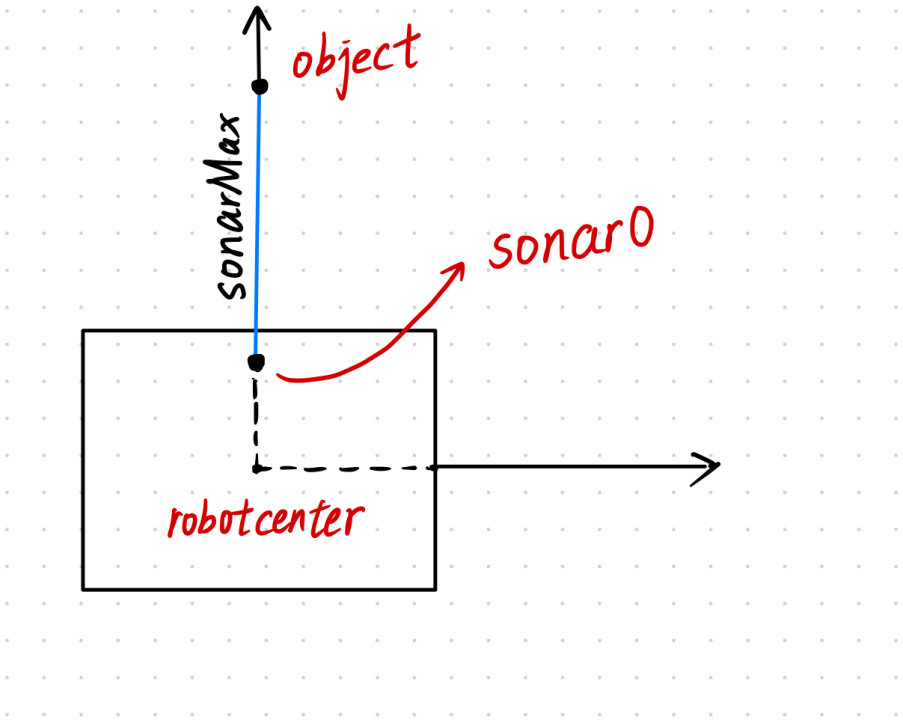
The purpose of the outer loop in the code is to read a gesture of robotPoses with each loop.

(*robotPoses* is a list of elements in the list that are instances of the Pose class that represent the robot's posture at this moment, e.g. robotPoses[0] represents the robot's first pose.)

P1=sonarHit(0,sonarPose0,robotPoses[i]) utilizes the the function *sonarHit* completed in WK.11.1.6, which returns the coordinate point P1 of the center of the sonar0 (relative to the center of the robot) in the global coordinate system

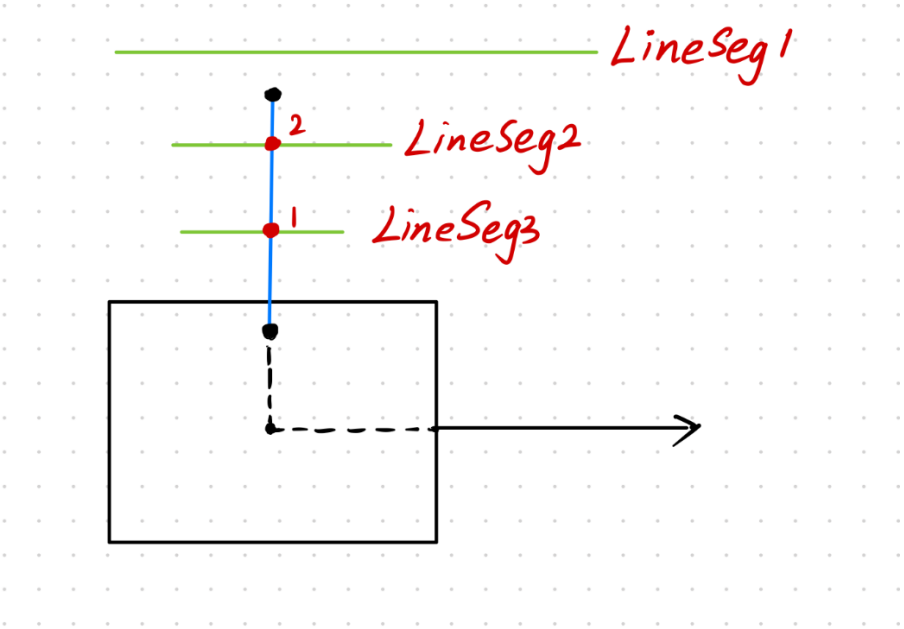
P2=sonarHit(sonarMax,sonarPose0,robotPose[i]) returns the position of the object in the global coordinate system at the detection distance of sonarMax (1.5m here) from sonar0

As shown in the following figure:



With the two points P1 and P2, we can generates a probe segment and the length of it is sonaMax. (the blue segment as shown in the figure). Each loop of the outer loop adds a probe line segment of length sonarMax to the list *sonarLine*. The inner loop reads an instance of util.LineSeg in the list *WallSeg* each time. The role of the inner loop is to find the nearest object detected by sonar0. The intersection method of class LineSeg can determine whether sonar0's probe segment intersects with a given segment in *WallSeg.*

As shown in the following figure:



Suppose the *WallSeg* has only 3 line segments, namely LineSeg1, LineSeg2, and LineSeg3. As can be seen from the figure, the detection line segment (blue line segment) has an intersection point with LineSeg2 and LineSeg3, and no intersection with LineSeg1, through the distance method of the Point class, the distance between the point where the center of sonar0 is located and the intersection point (that is, the detection distance) can be calculated, if there is no intersection, we return the detection distance as sonarMax. We use list d, add the three detection distances to the list after the end of the inner loop, judge the minimum value in list d in each outer loop, take min(d) as an argument to the function named *discreteSonar,* and add the

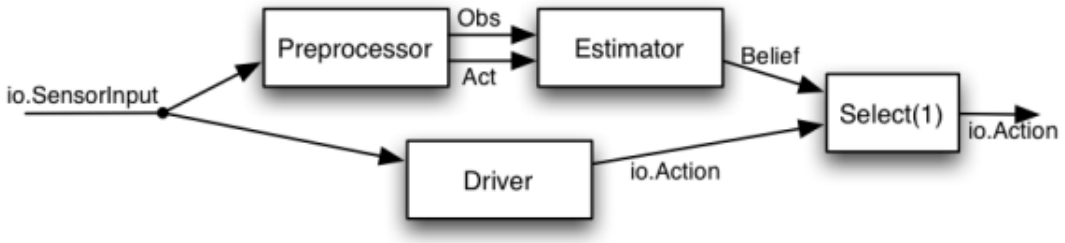
return value to the *idealreading.* Each time the outer loop, the discreted value added to the *idealreading* corresponds to the discreted ideal reading of sonar0 in each pose of the robot.

Step11

Wk.12.2.3

Localization：

Here is the architecture of the system:



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 )
```

Preprocessor output an observation from step t-1 and the action that took place between steps t-1 and t. The input of the Estimator is the output of the Preprocessor, The Estimator is an instance of a state machine that acts as a state estimator.

1. At time 0

Preprocessor：

```
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 =` None
  - `act =` None

Estimator：

`Estimator (at time 0):`

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

0.1	0.1	0.1	0.1	0.1	0.1
0.1	0.1	0.1	0.1		

Explanation：

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. As for the Estimator,we can let the starting distribution be uniform.

2. At time 1

Preprocessor：

`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` = 5
  - `act` = 1

Estimator:

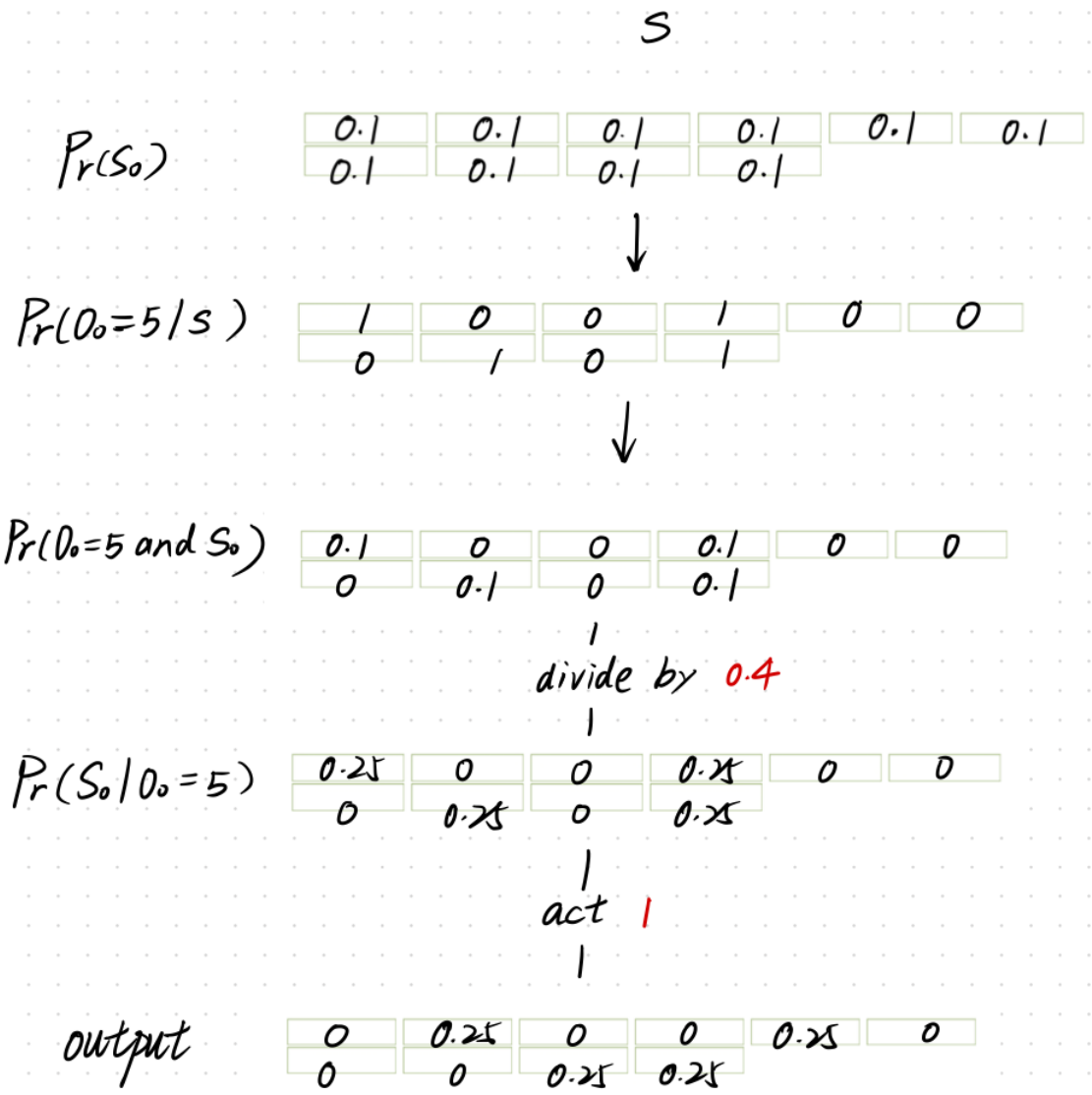
`. Estimator (at time 1):`

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

0	0.25	0	0	0.25	0
0	0	0.25	0.25		

Explanation:

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



3. At time 2

Preprocessor:

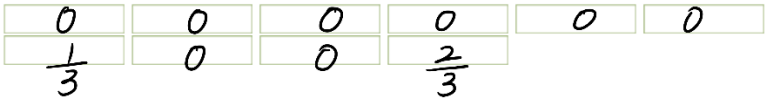
```
Preprocessor (at time 2):
```

- 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` = 1
  - `act` = 5

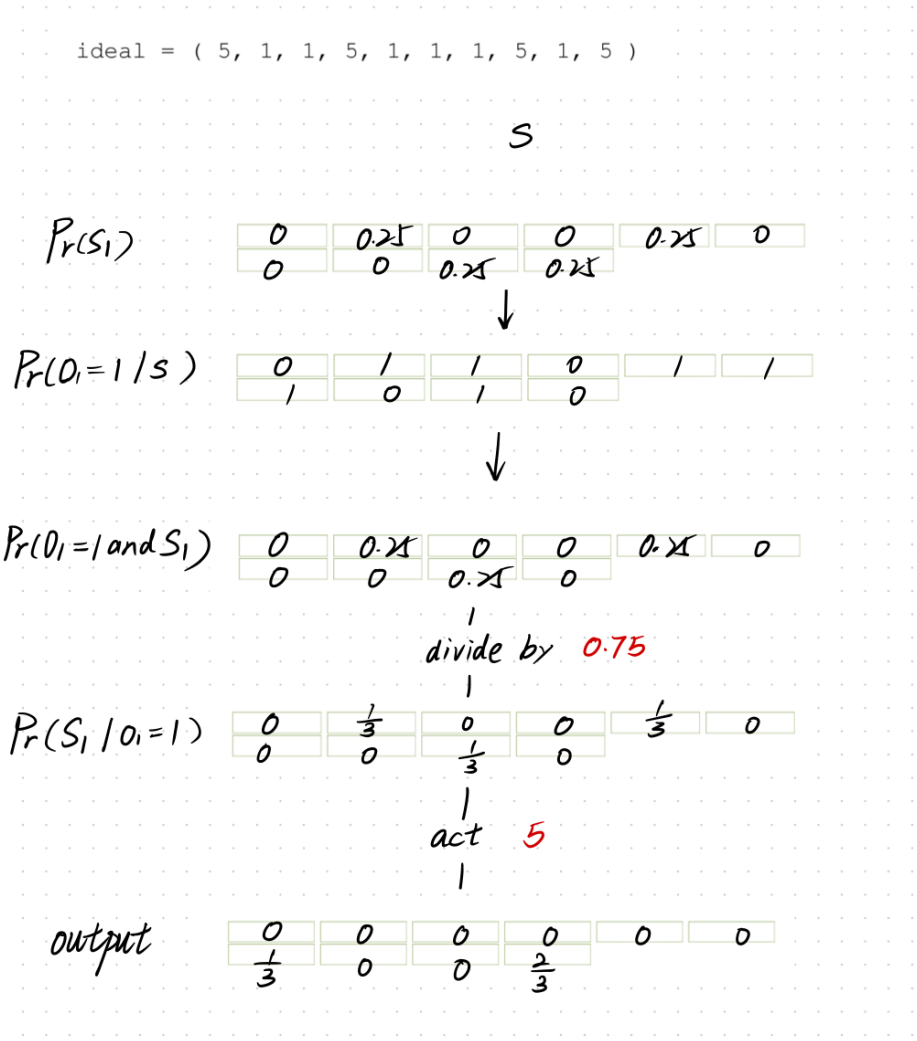
Estimator:

```
Estimator (at time 2):
```

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



Explanation:



Summary