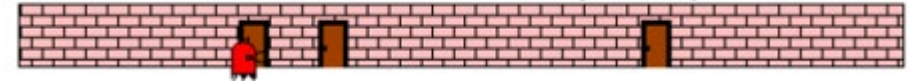


Particle filter for 1D localization

Lab -- ENGR 509

Problem Setup

Problem Setup



- Robot localization
 - Given the map, where am I (on the map)?

$$Bel(x_t) = \eta p(z_t | x_t) \int p(x_t | x_{t-1}, u_t) Bel(x_{t-1}) dx_{t-1}$$

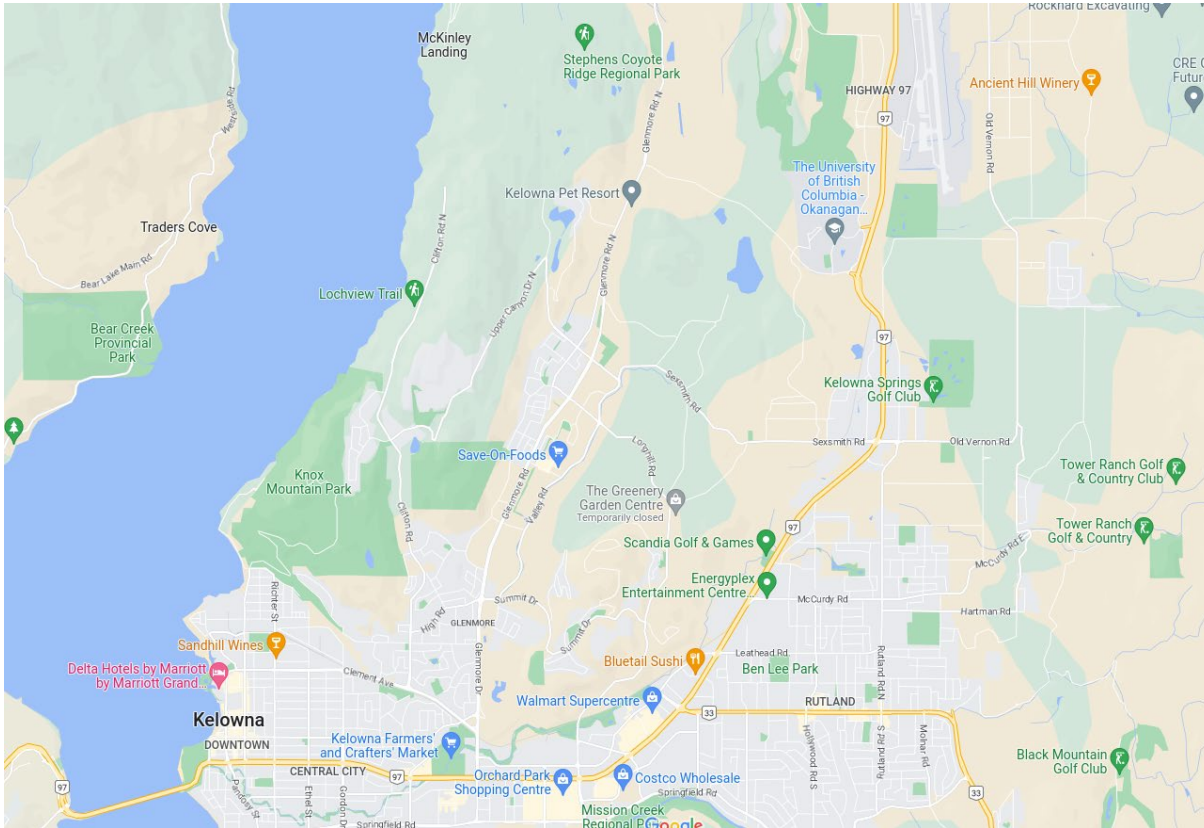
current
location
estimate

observation
model

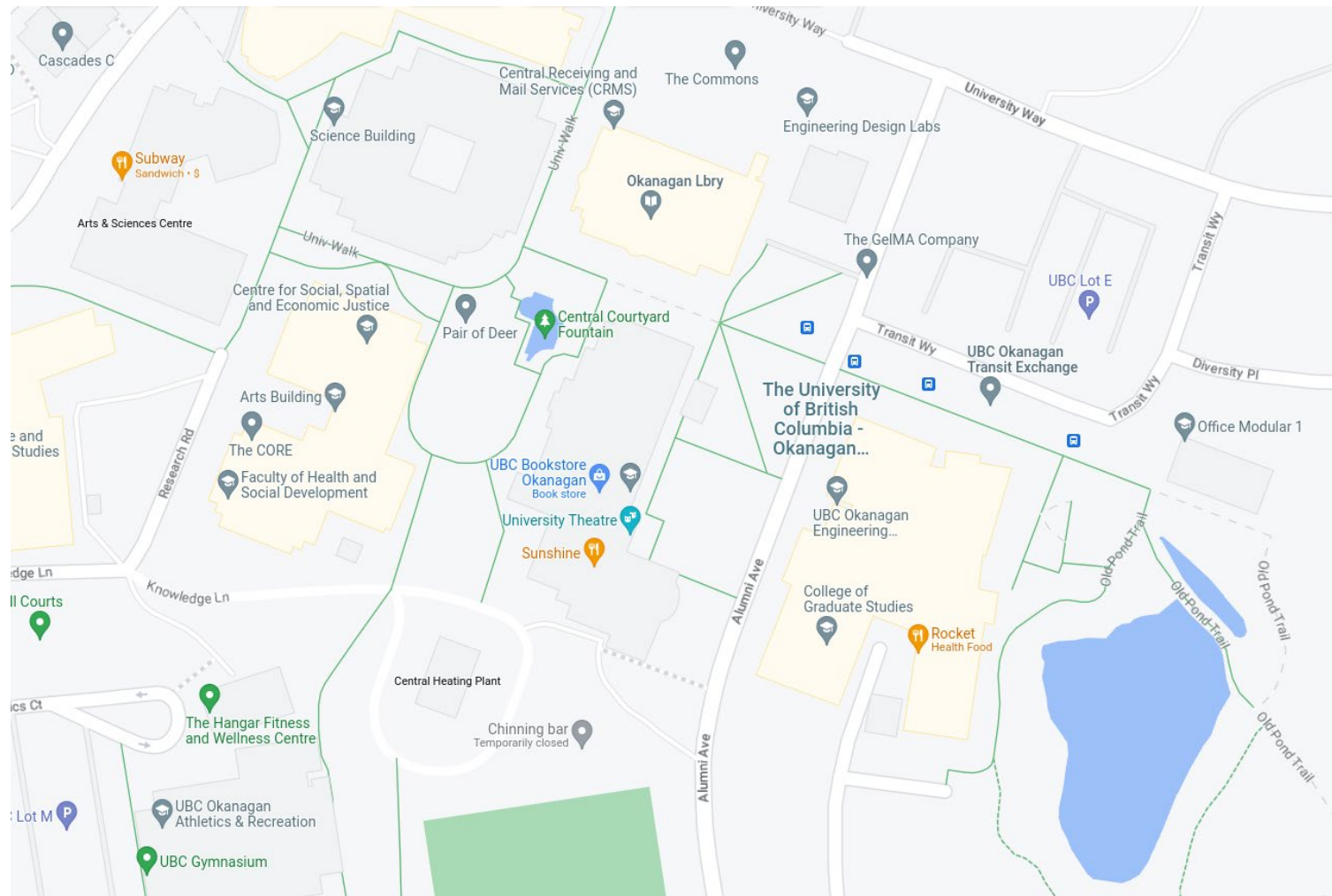
motion
model

previous
location
estimate

Examples



Map



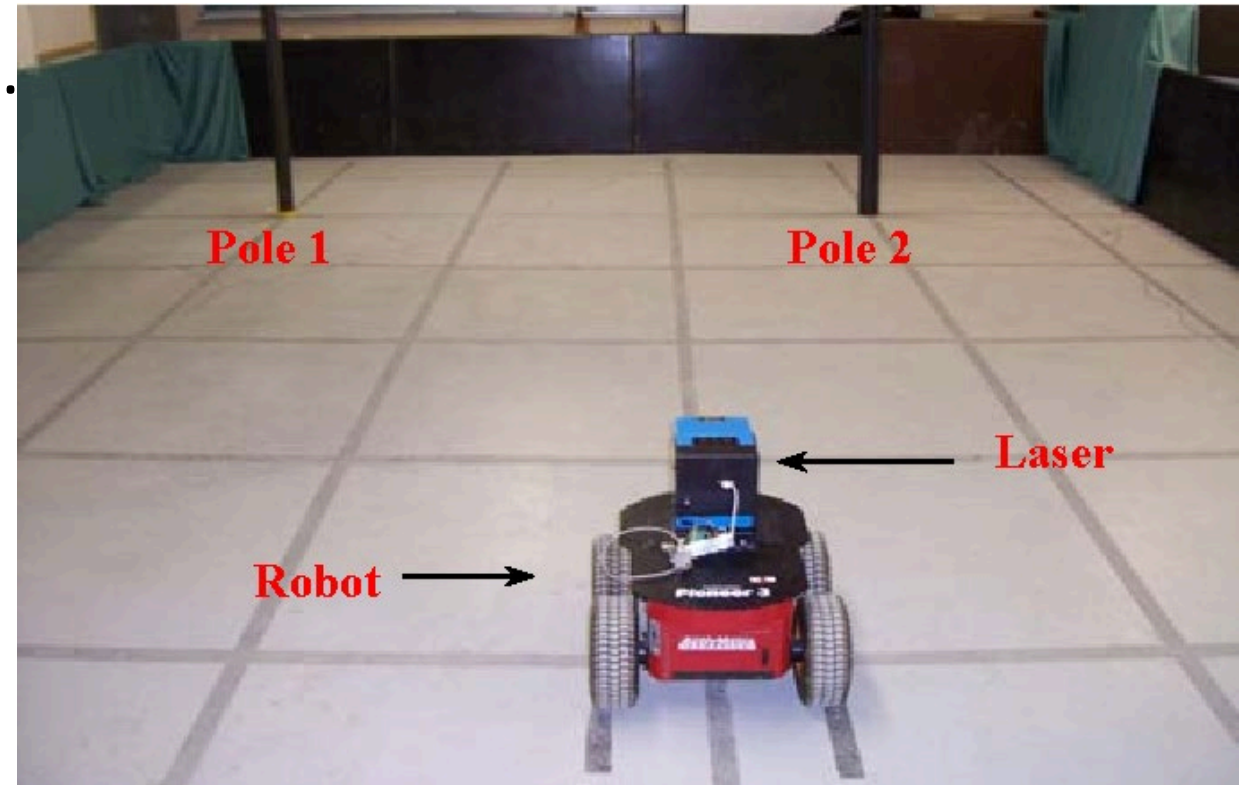
What are the information on the map that we need to do localization?

- locations of landmarks

As long as we have enough landmarks, and relative location to the landmarks, we can localize yourself.

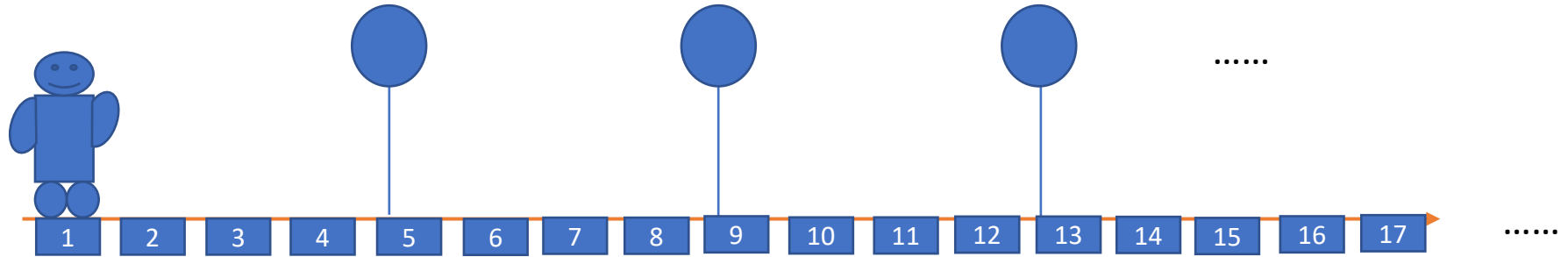
Our implementation

- landmark: poles
- A range sensor tells how far to a pole.
- Remember: robot knows the map.



Particle Filter on 1D localization

Particle Filter for 1D localization



Now, we change the rule:

- The robot movement is not perfect. Although the control command is moving forward by 1.0 unit, the robot can move 1.0 unit plus some errors.
- The robot measurements are not perfect. For example, the measurement is 2.5 units to a pole, meaning distance could be 2.5 units plus some errors.
- For simplicity, we model the errors follow zero-mean Normal distributions.

Particle Filter for 1D localization

$$Bel(x_t) = \eta p(z_t | x_t) \int p(x_t | x_{t-1}, u_t) Bel(x_{t-1}) dx_{t-1}$$

draw x_{t-1}^i from $Bel(x_{t-1})$

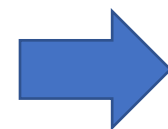
draw x_t^i from $p(x_t | x_{t-1}^i, u_t)$

Importance factor for x_t^i :

$$\begin{aligned} w_t^i &= \frac{\text{target distribution}}{\text{proposal distribution}} \\ &= \frac{\eta p(z_t | x_t) p(x_t | x_{t-1}^i, u_t) Bel(x_{t-1}^i)}{p(x_t | x_{t-1}^i, u_t) Bel(x_{t-1}^i)} \\ &\propto p(z_t | x_t) \end{aligned}$$

Particle Filter for 1D localization

- Sample the next generation for particles using the proposal distribution
- Compute the importance weights :
 $weight = target\ distribution / proposal\ distribution$
- Resampling: “Replace unlikely samples by more likely ones”



Algorithm Particle_filter($\mathcal{X}_{t-1}, u_t, z_t$):

$\bar{\mathcal{X}}_t = \mathcal{X}_t = \emptyset$

for $m = 1$ *to* M *do*

sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$

$w_t^{[m]} = p(z_t \mid x_t^{[m]})$

$\bar{\mathcal{X}}_t = \bar{\mathcal{X}}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$

endfor

for $m = 1$ *to* M *do*

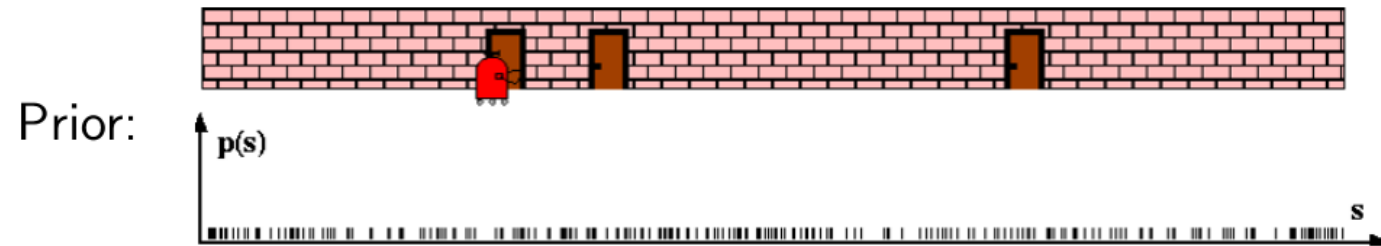
draw i *with probability* $\propto w_t^{[i]}$

add $x_t^{[i]}$ *to* \mathcal{X}_t

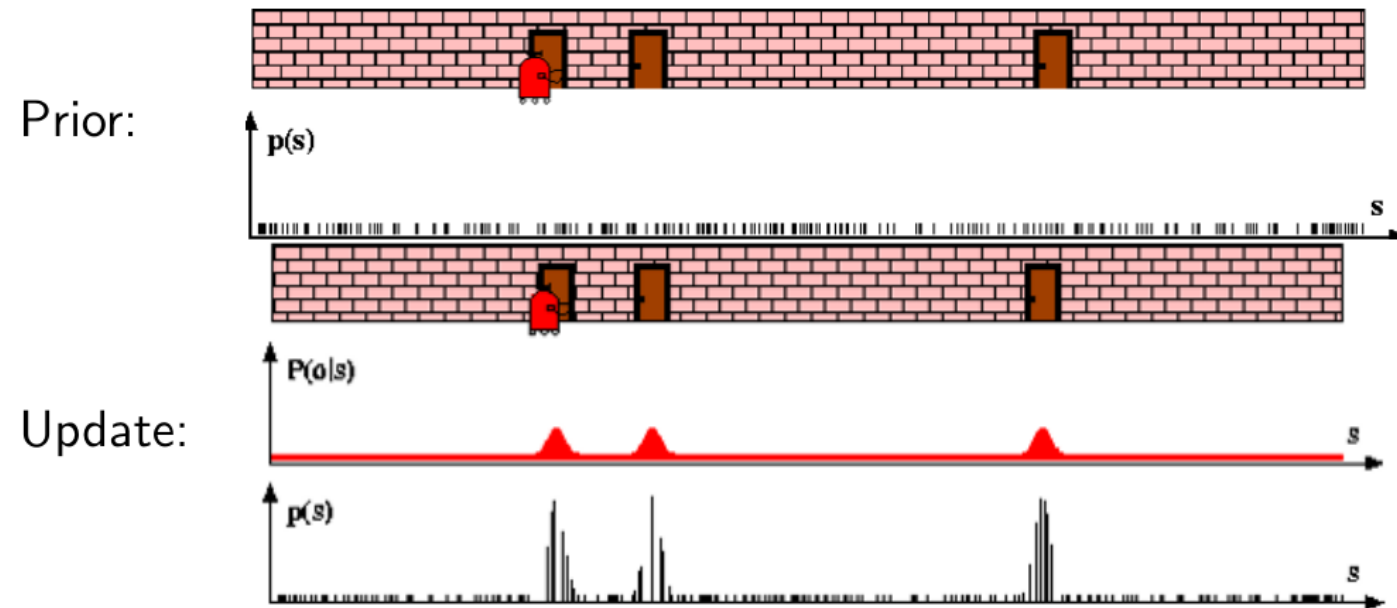
endfor

return \mathcal{X}_t

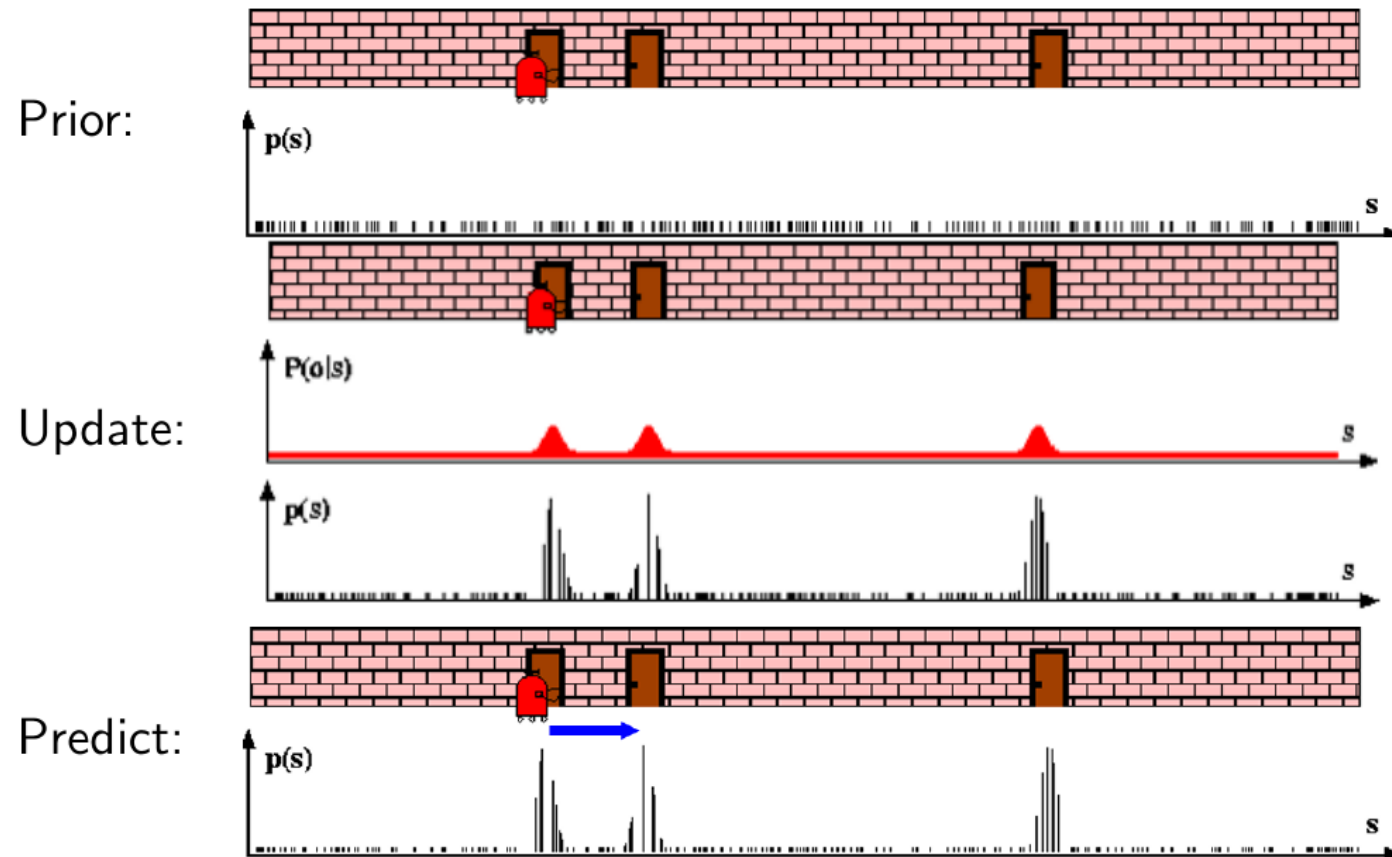
Particle filter for 1D localization



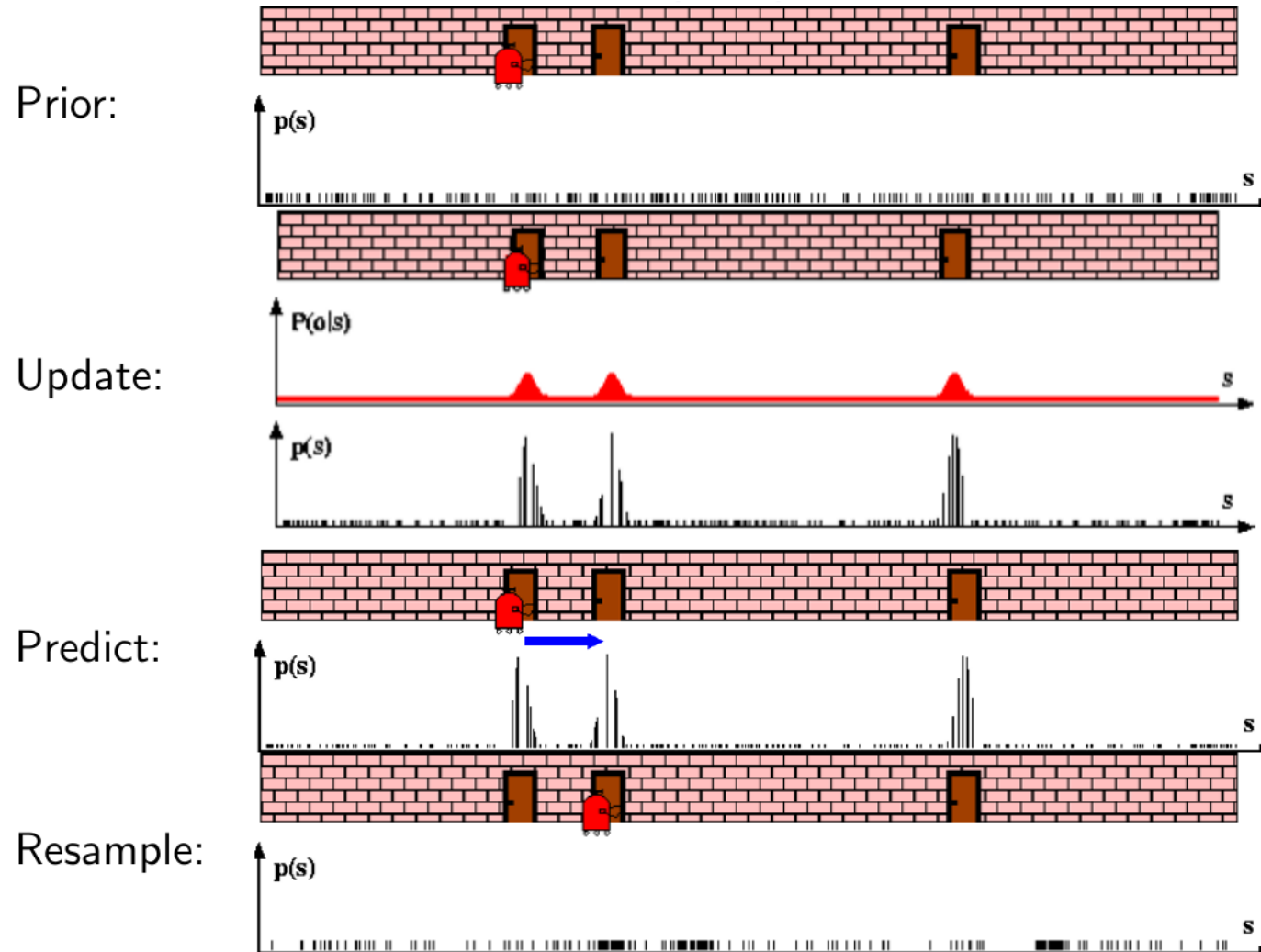
Particle filter for 1D localization



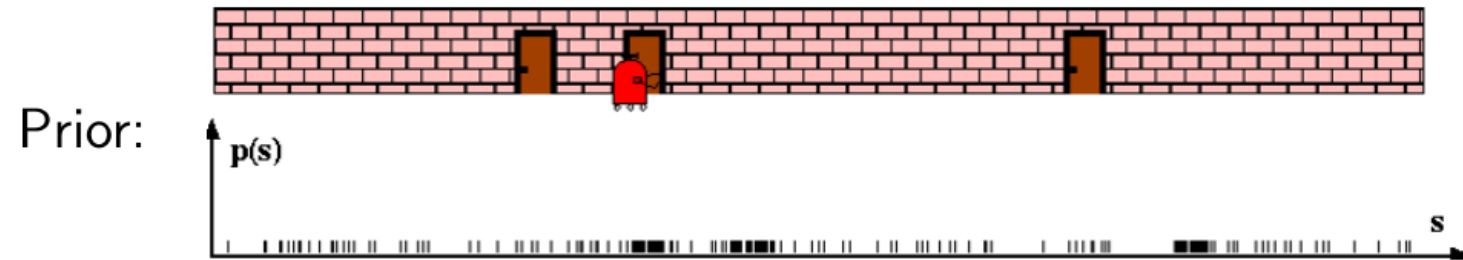
Particle filter for 1D localization



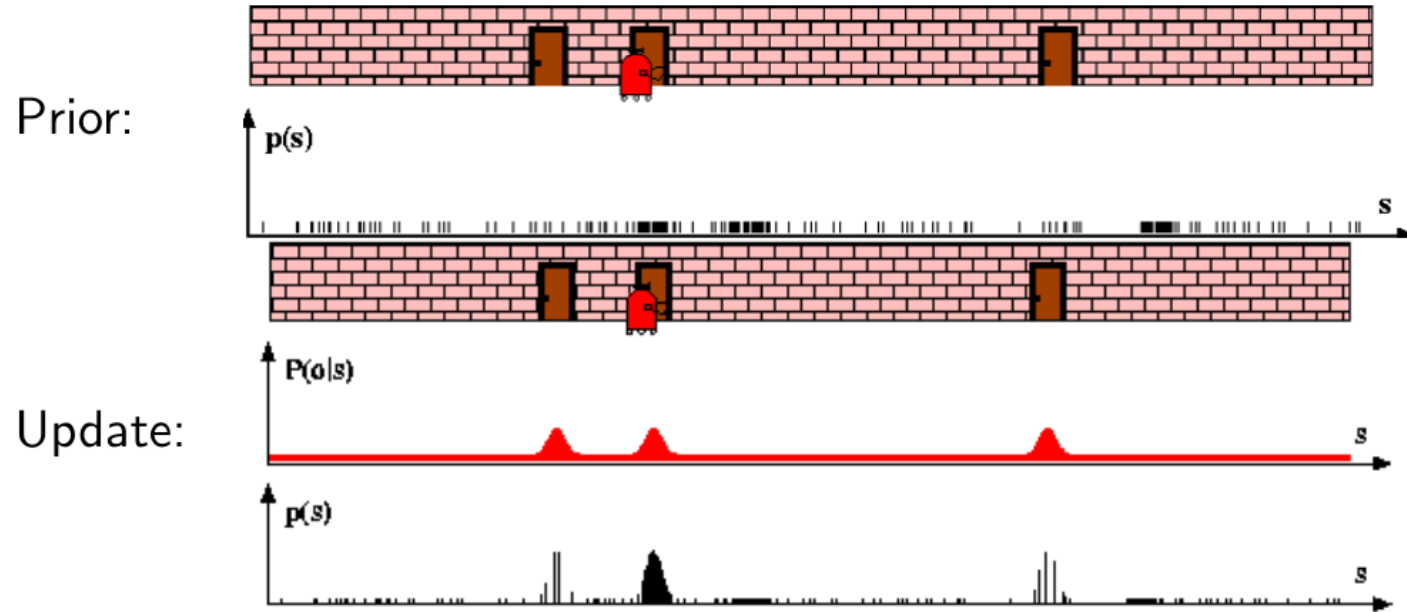
Particle filter for 1D localization



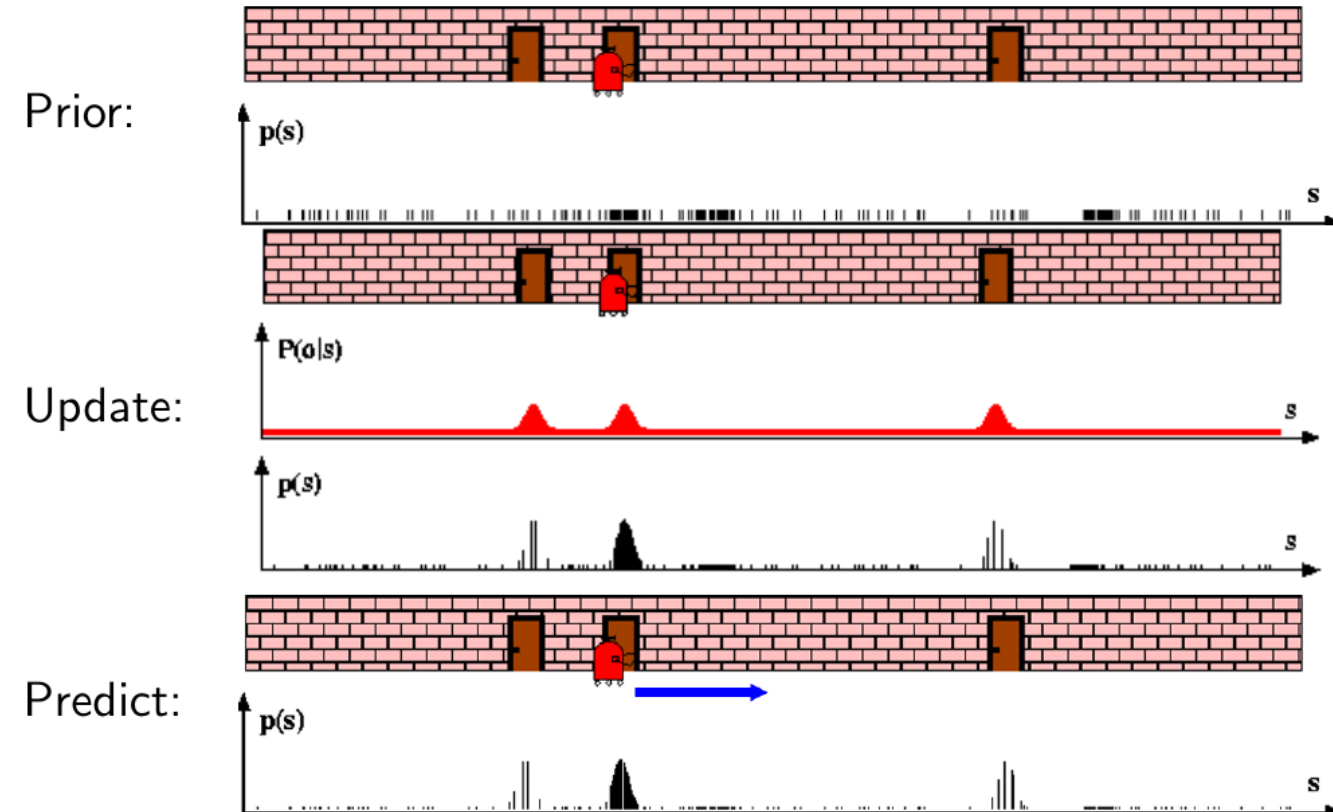
Particle filter for 1D localization



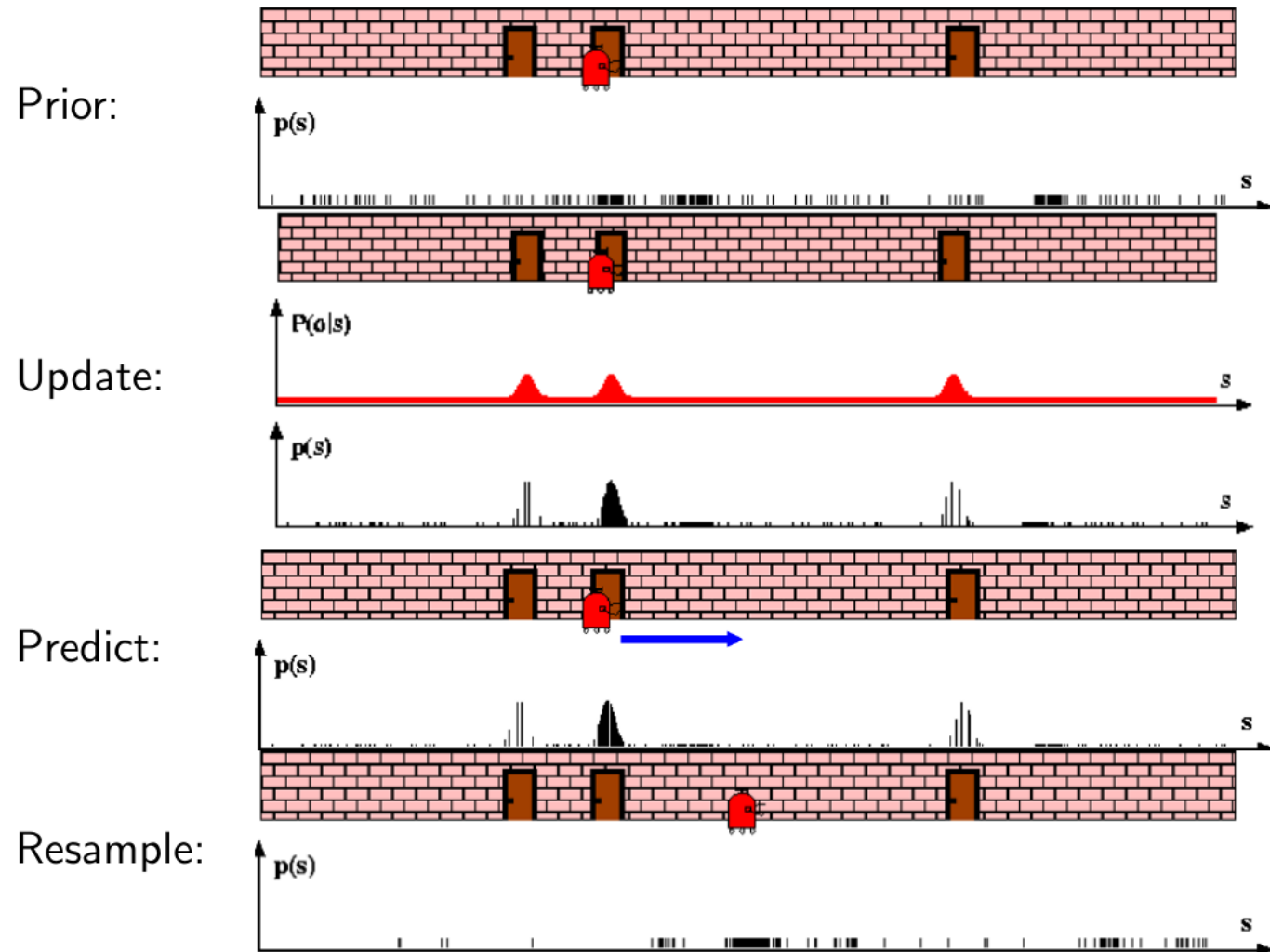
Particle filter for 1D localization



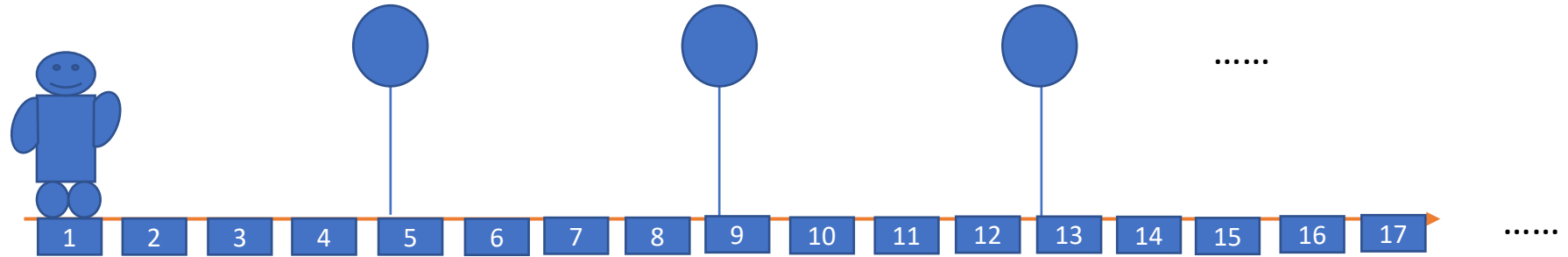
Particle filter for 1D localization



Particle filter for 1D localization



Step by step implementing a particle filter



Step 1: Generate particles based on our prior

- uniformly prior distributed (each spot has one particle)
- each particle moves following with robot moving, no uncertainties in their movements.
- at this step, we assume each particle holds their beliefs only to be true (belief=1) of false (belief=0)

Step by step implementing a particle filter

Step 2: Add uncertainties in particles' movements

- movement errors follow zero-mean Gaussian distribution with a predefined standard deviation
- create one particle, move 10 times, print and observe the measurements
- uncomment `# quit()` to see how distribution converges with more samples.

Step by step implementing a particle filter

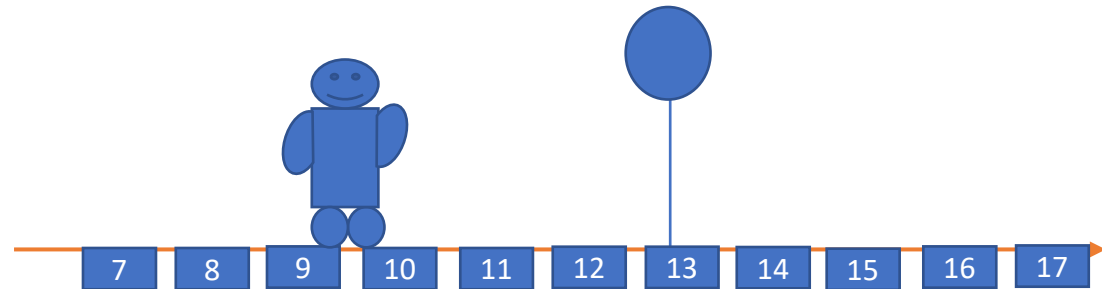
Step 3: More realistic sensor model:

Previous measurement

```
def detect_pole(self, poles):  
    if self.pos + 1 in poles:  
        self.pole_detected = True  
    else:  
        self.pole_detected = False
```

More realistic measurement considering sensor specifications:

- Maximum measurement range: 3 units
- If object within the detection range, report the distance to the closest object
- If no object detected, output -1000



Step by step implementing a particle filter

Your practice:

[Complete the step-3.py](#), fill in the measure function so that

“Measurement should be XXX”

matches

“You Measured: XXX”

Step by step implementing a particle filter

- Step 4: Update weights for each particle while taking into account measurement uncertainty in the sensor model.
 - For simplicity, we assume the sensor model: errors follow Gaussian distribution with a predefined standard deviation.
 - **Qs**: if robot got measurement 3.0 units to a pole, particle A is at the location distancing 2.0 units to the pole, particle B at the location distancing 3.0 units. How would you assign the weights?
 - Complete the step-4.py: update weights by setting the particle.weight value based on the given Gaussian probability density function.

Step by step implementing a particle filter

- Step 5: Resample

- Complete the `resample_particles` function
- Change the particle number and weights, do experiment and Observe
 - e.g., change the number of particles to be 100, and weights to be either 1 and 0.05 (i.e., large number of particles, but most of them have really small likeliness)

Step by step implementing a particle filter

- Step 6: Put it together

- You can reuse all the steps we implemented before
- Put them together to implement a complete particle filter for 1D localization

Any problems found?
How do you fix it?

