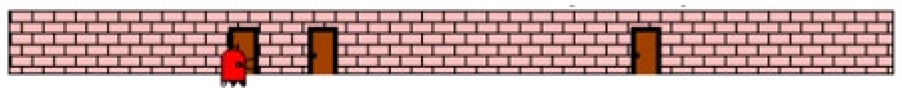
# lab-Particle filter

ENGR 509

## Problem Setup

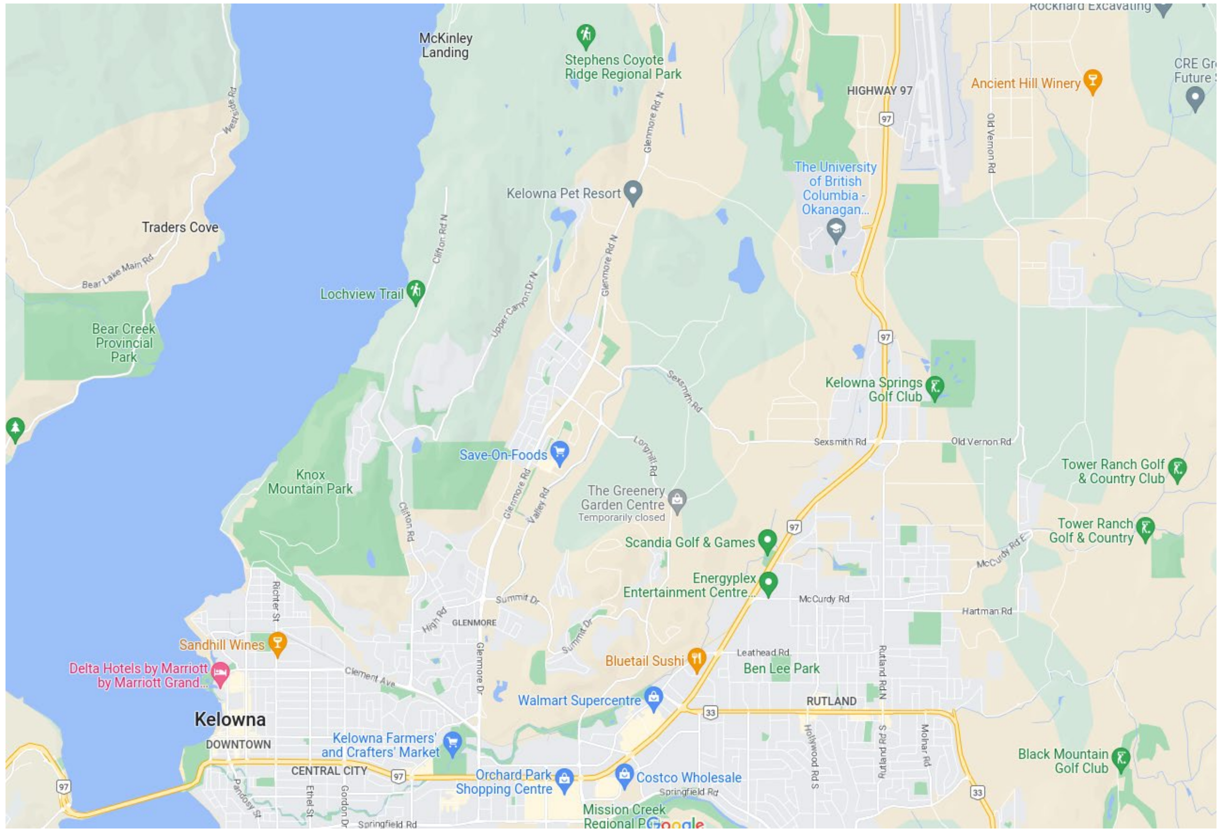
# Problem Setup



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

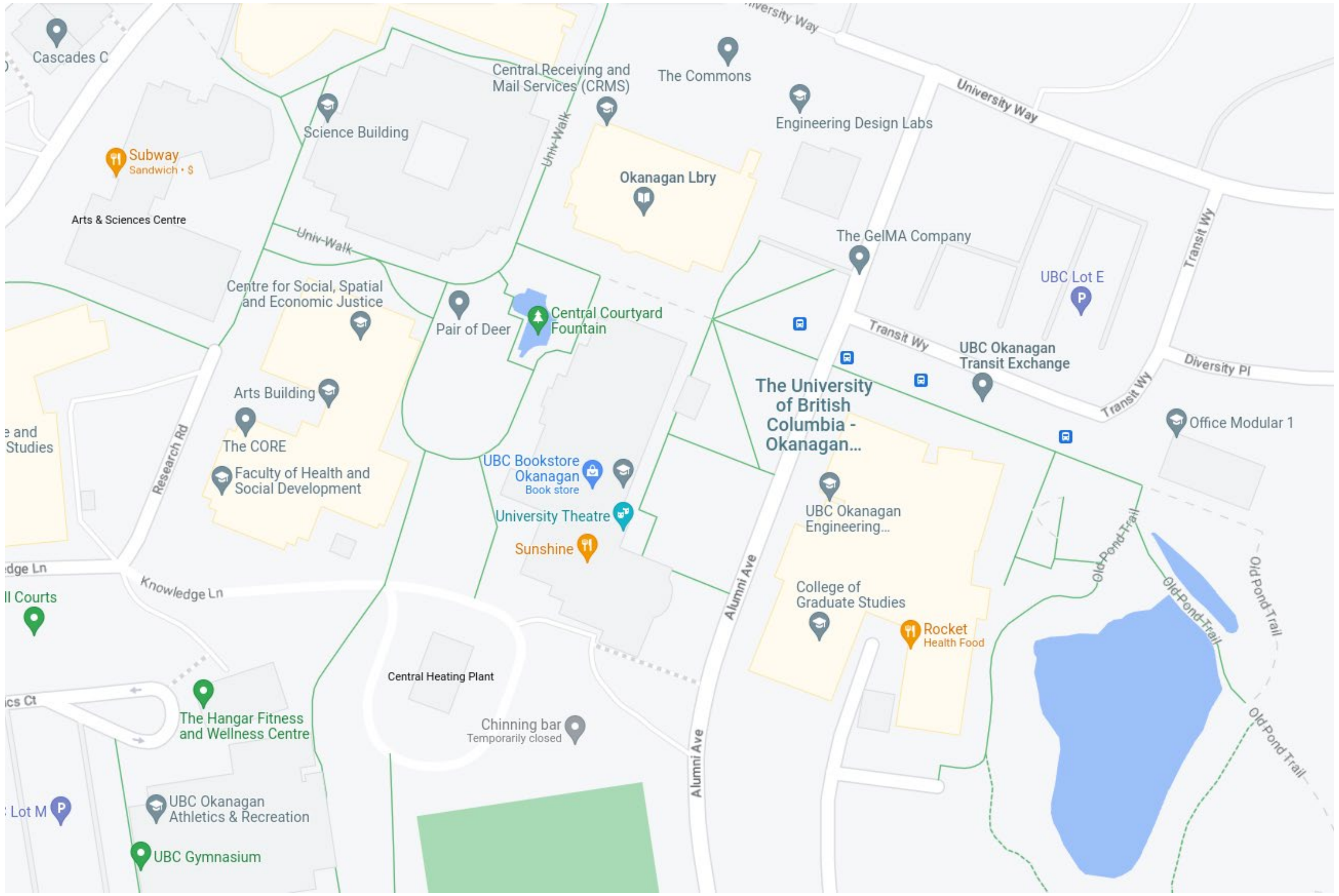
$$\underbrace{Bel(x_t)}_{\text{current location estimate}} = \eta\, \underbrace{p(z_t \mid x_t)}_{\text{observation model}} \int \underbrace{p(x_t \mid x_{t-1}, u_t)}_{\text{motion model}}\, \underbrace{Bel(x_{t-1})}_{\text{previous location estimate}}\, dx_{t-1}$$

## Examples


Kelowna
DOWNTOWN
CENTRAL CITY
RUTLAND
GLENMORE
HIGHWAY 97
McKinley Landing
Stephens Coyote Ridge Regional Park
Ancient Hill Winery
The University of British Columbia - Okanagan
Kelowna Pet Resort
Traders Cove
Lochview Trail
Bear Creek Provincial Park
Kelowna Springs Golf Club
Save-On-Foods
Knox Mountain Park
The Greenery Garden Centre
Tower Ranch Golf & Country Club
Scandia Golf & Games
Energyplex Entertainment Centre
Sandhill Wines
Delta Hotels by Marriott by Marriott Grand
Bluetail Sushi
Ben Lee Park
Walmart Supercentre
Kelowna Farmers' and Crafters' Market
Orchard Park Shopping Centre
Costco Wholesale
Black Mountain Golf Club
Mission Creek Regional Park
Google

Living Room
Kitchen
Entryway
Dining Room

# Map



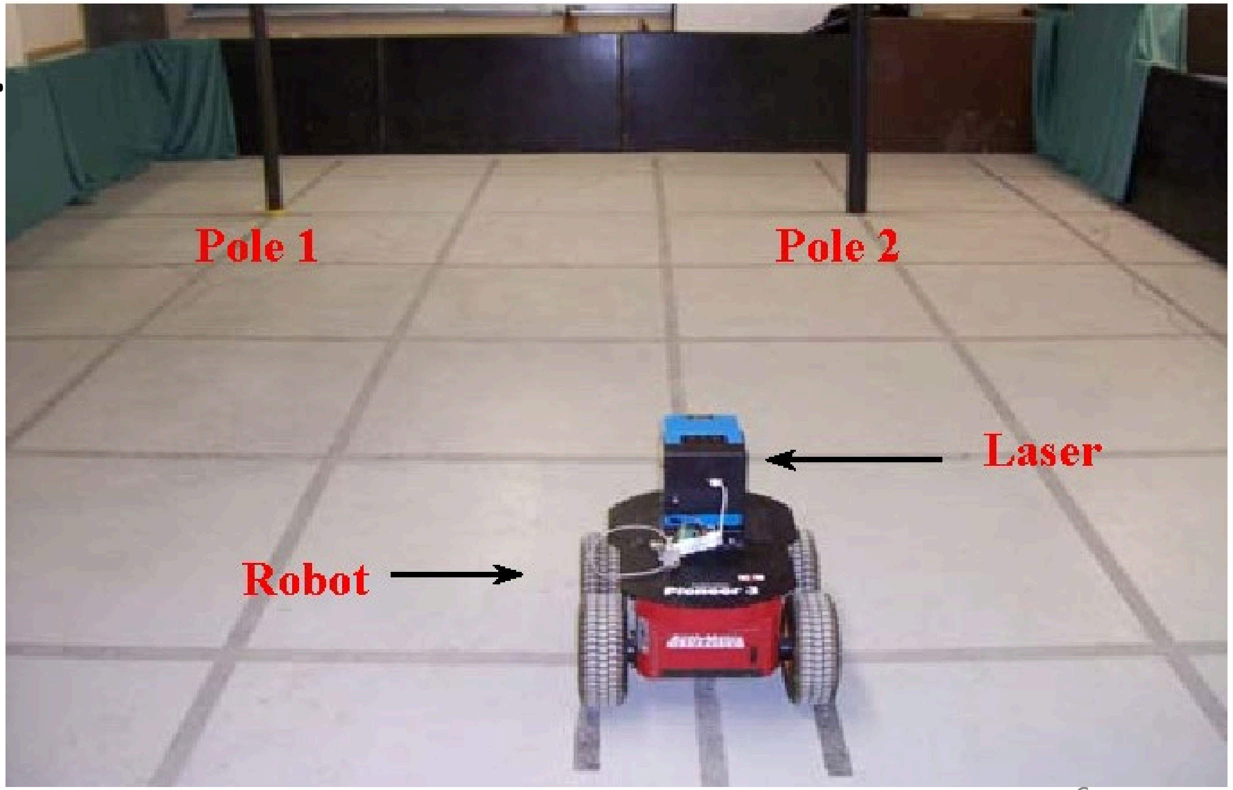
What are the information on the map that we need to do localization?

- locations of landmarks

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

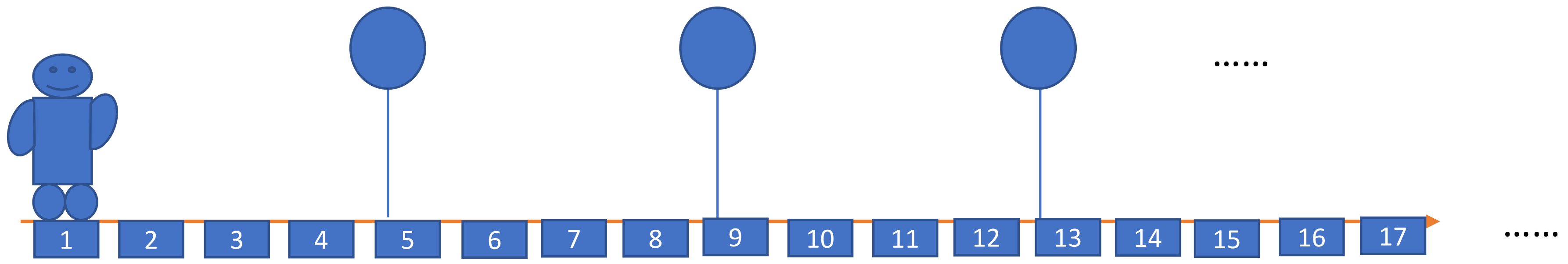
## Our implementation

- 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

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

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

draw $x^i_t$ from $p(x_t \mid x^i_{t-1}, u_t)$

Importance factor for $x^i_t$:

$$w^i_t = \frac{\text{target distribution}}{\text{proposal distribution}}$$

$$= \frac{\eta\, p(z_t \mid x_t)\, p(x_t \mid x_{t-1}, u_t)\, Bel\ (x_{t-1})}{p(x_t \mid x_{t-1}, u_t)\, Bel\ (x_{t-1})}$$

$$\propto p(z_t \mid x_t)$$

# Particle Filter for 1D localization

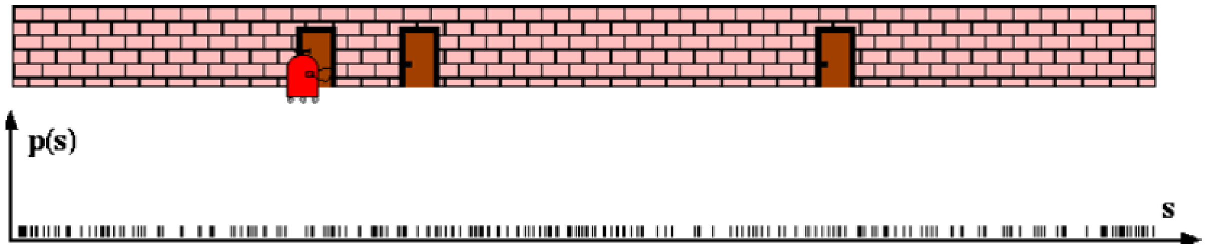
- Sample the next generation for particles using the proposal distribution
- Compute the importance weights :

  *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

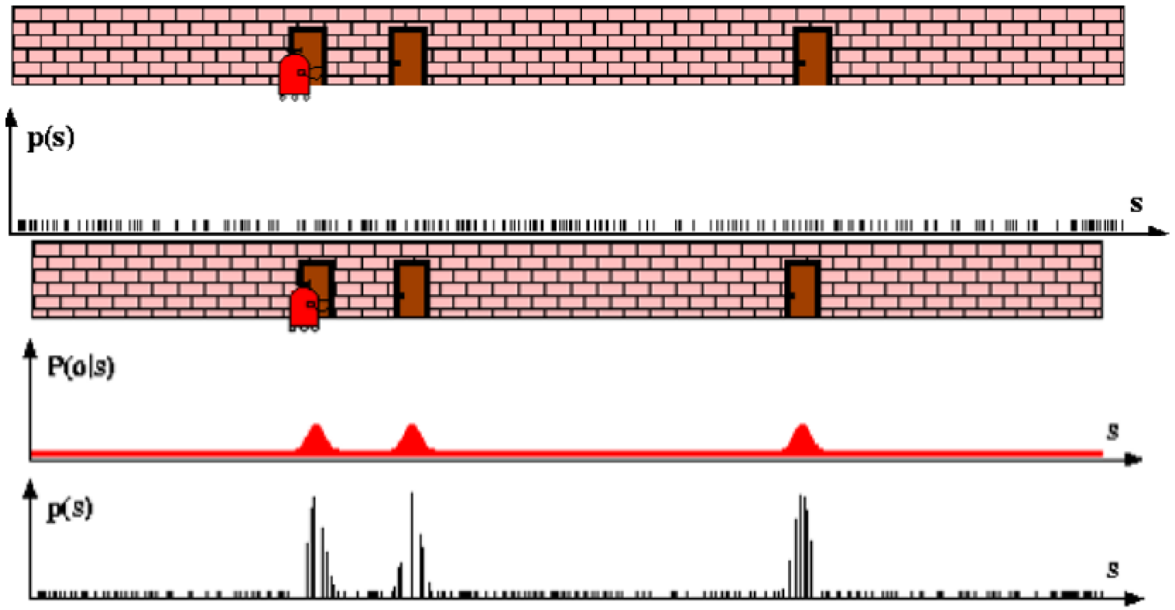
Prior:



## Particle filter for 1D localization

Prior:

Update:

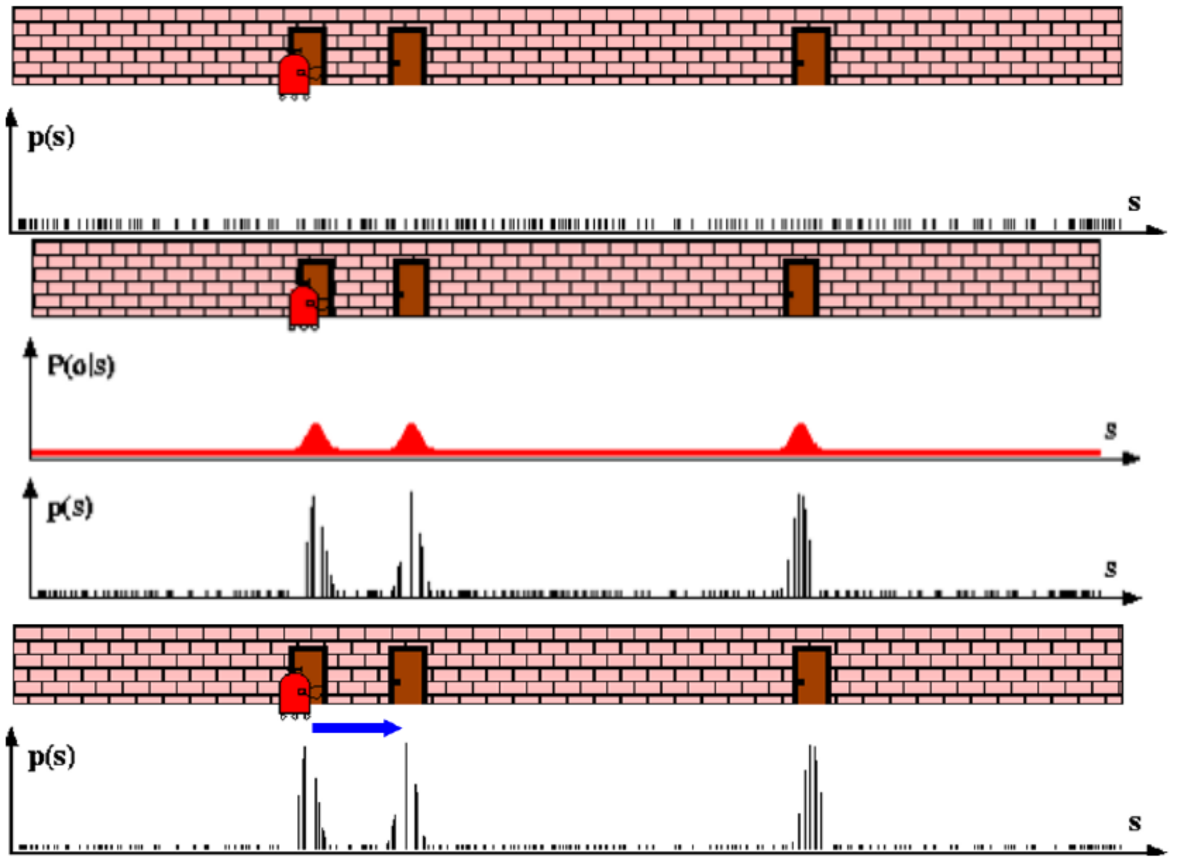


# Particle filter for 1D localization

Prior:

Update:

Predict:



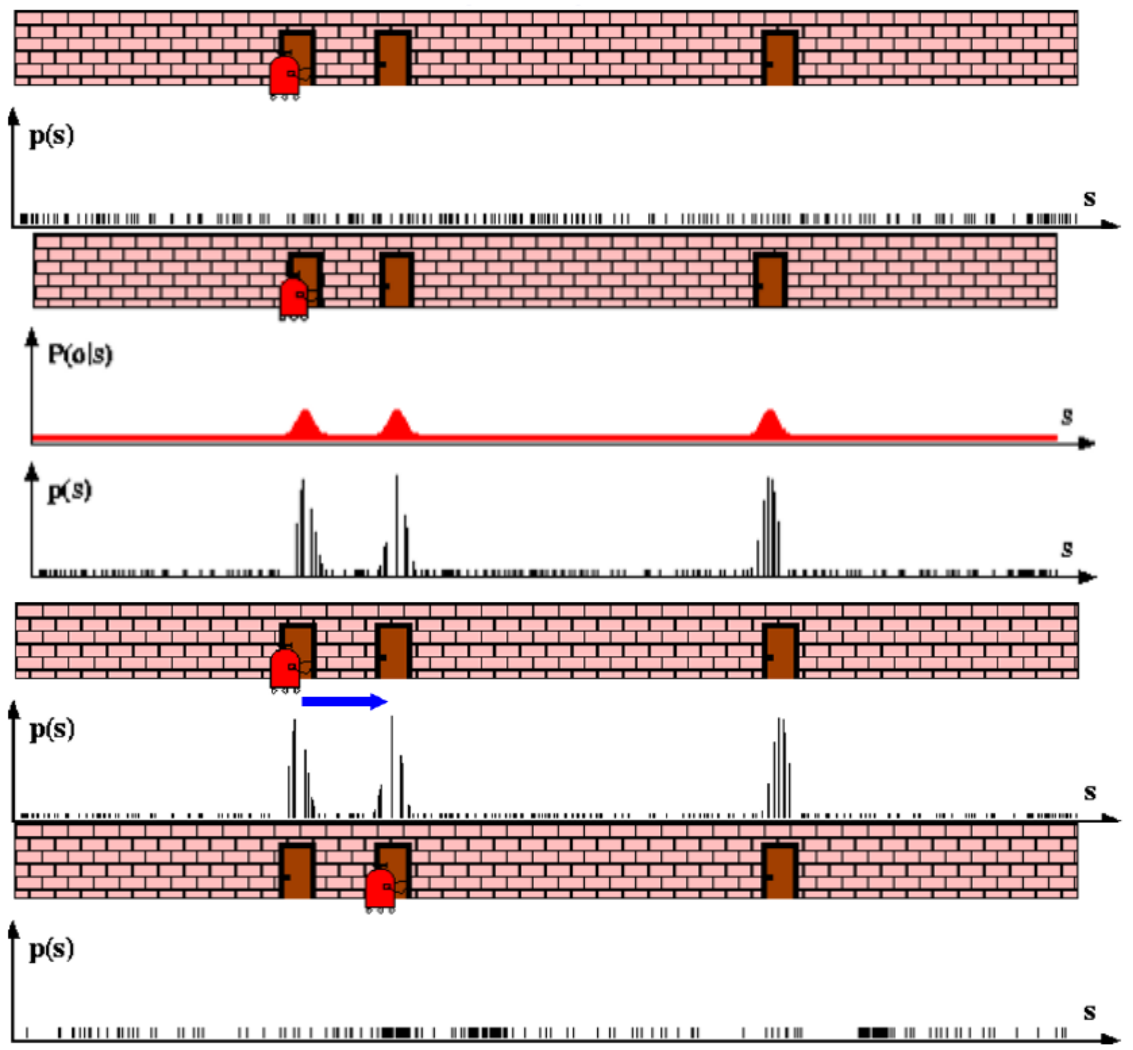
# Particle filter for 1D localization

Prior:

Update:

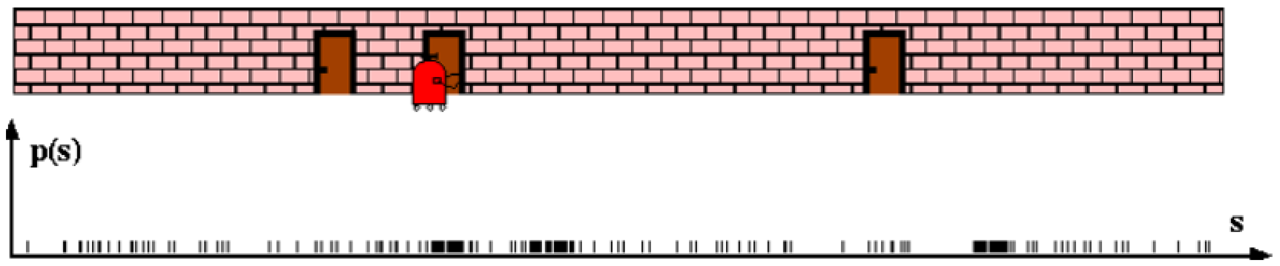
Predict:

Resample:



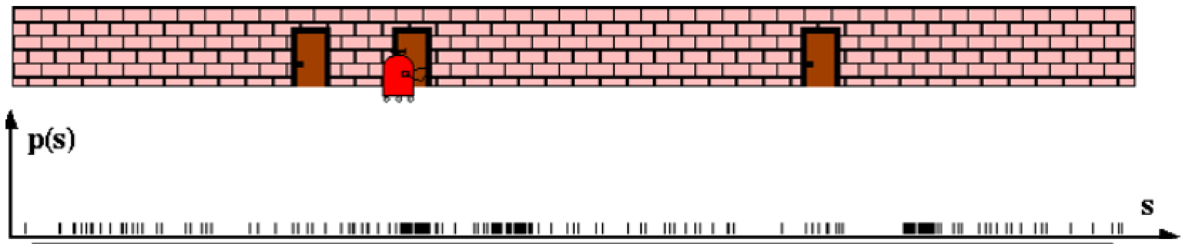
# Particle filter for 1D localization

Prior:

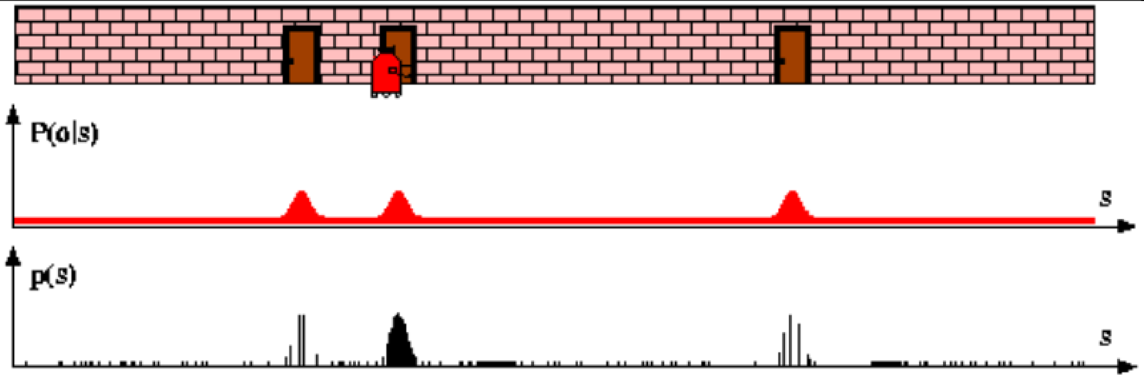


# Particle filter for 1D localization

Prior:



Update:

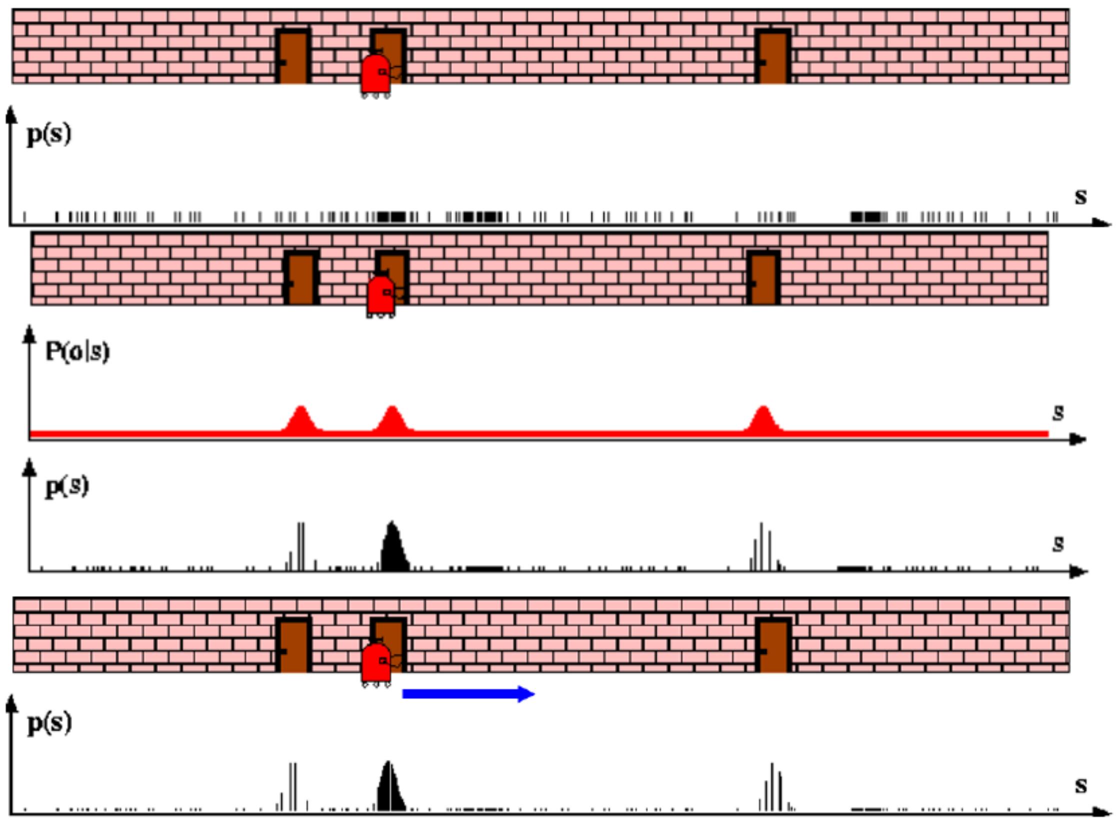


# Particle filter for 1D localization

Prior:

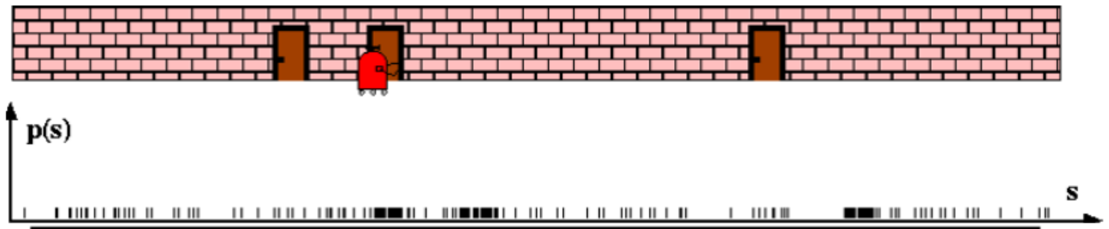
Update:

Predict:

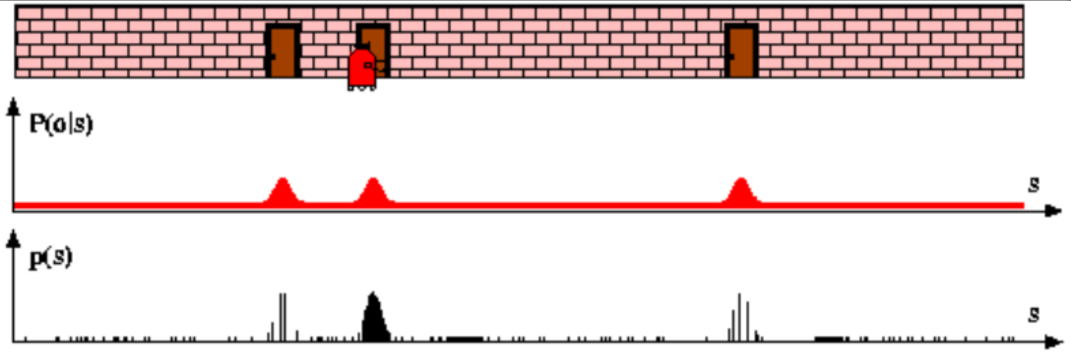


# Particle filter for 1D localization

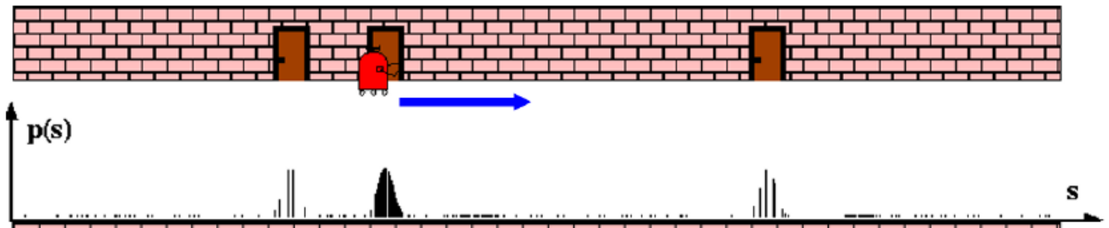
Prior:



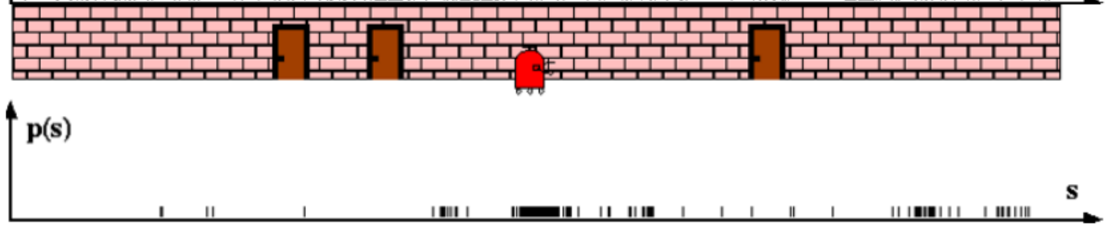
Update:



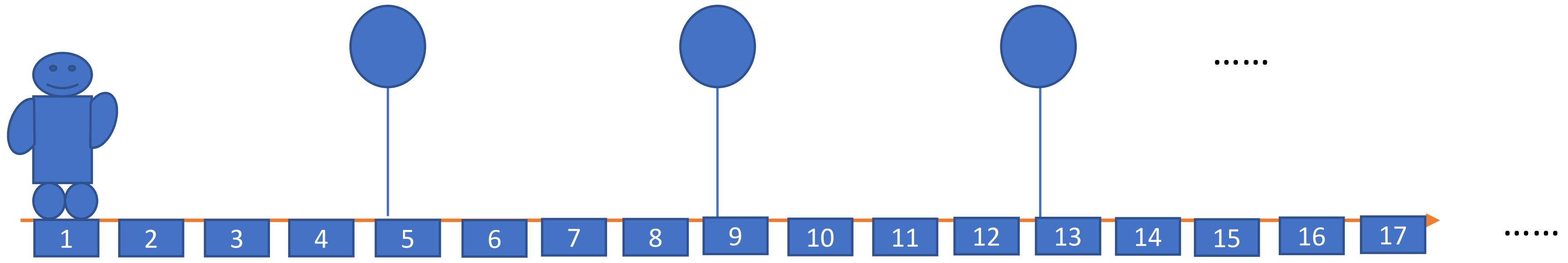
Predict:



Resample:



# 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

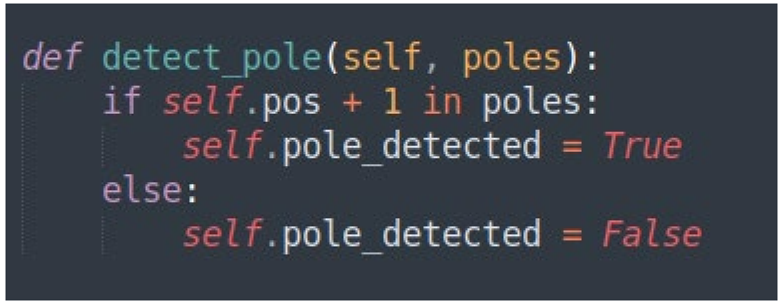
### Step 2: Add uncertainties in particles’ movements

- 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:

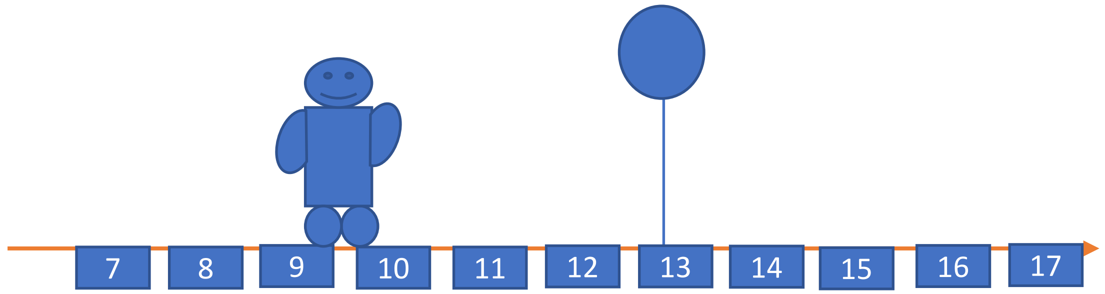
Previous measurement



```
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)

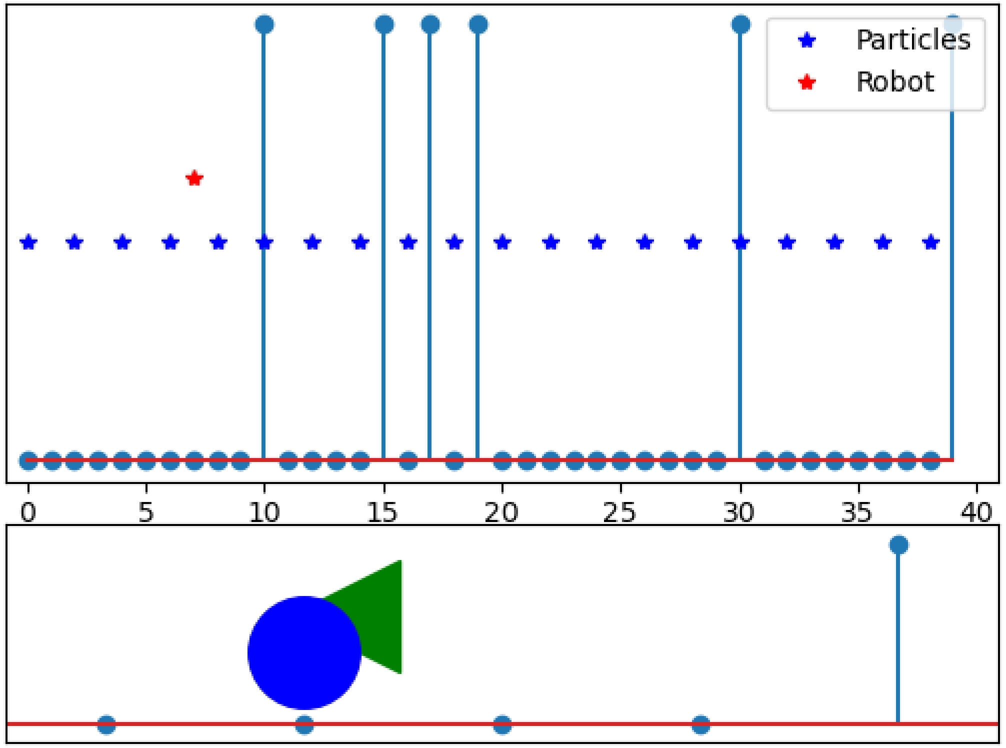
# Step by step implementing a particle filter

- 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?



## Particle Filter for 2D localization

## Extend to 2D case—Not mandatory

- I will give you files including the world definition and sensor readings, where the odometry motion model and a range-only sensor are used.

You need to do the following:

- Fulfill the sample motion function for particles

- Define the weight update for the particles

- Define the resampling function

A code skeleton with the particle filter framework is provided for you.

# Extend to 2D case—Not mandatory

- data -This folder contains files representing the world definition and sensor readings used by the filter.
- code -This folder contains the particle filter framework.
- Run the simulation- in the terminal: *python particle_filter_st.py*. It will only work properly once you filled in the blanks in the file. The blanks include

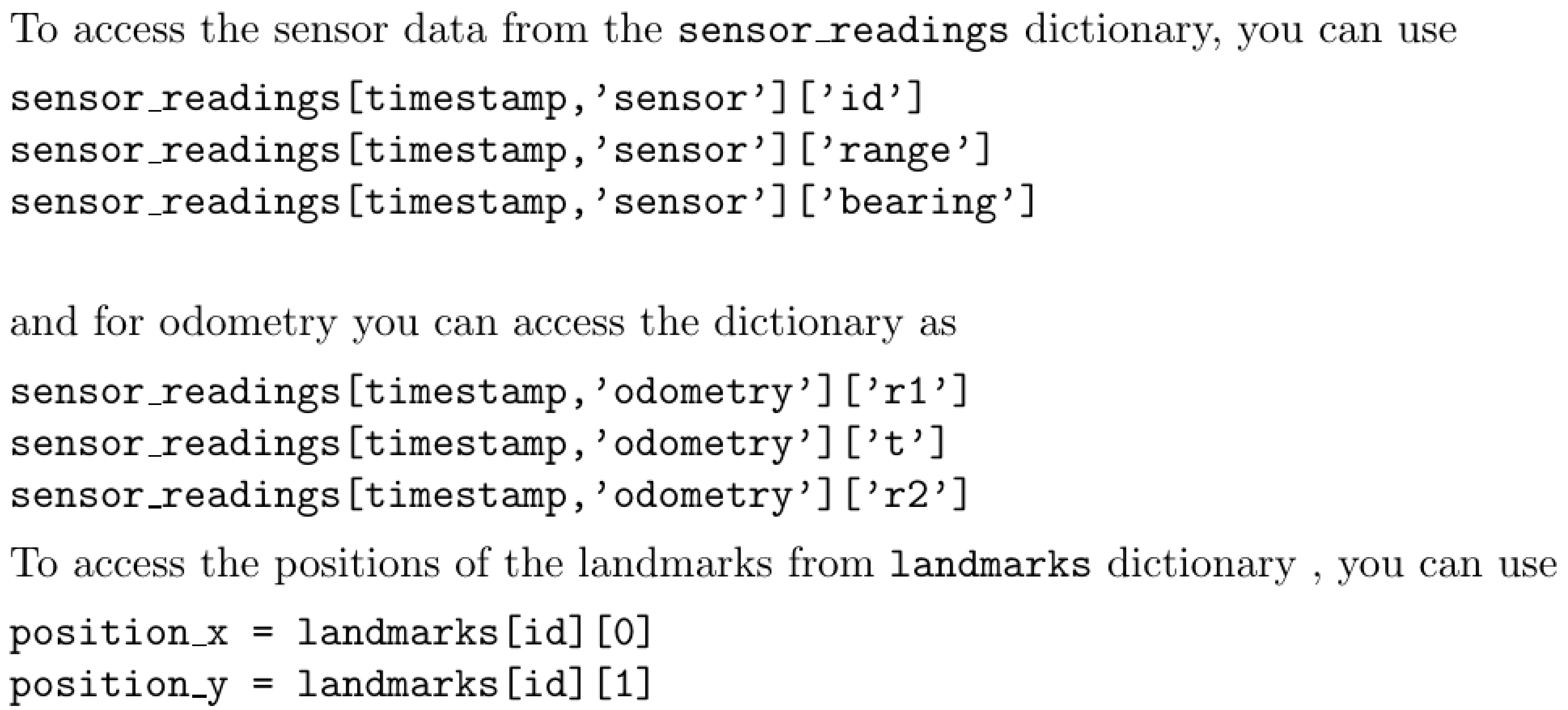
- sample_motion function: **implementing the odometry motion model and sampling from it**, then return the new set of parameters after the motion update. The function samples new particle positions based on the old positions, the odometry measurements and the motion noise. The motion noise are

$$[\alpha_1, \alpha_2, \alpha_3, \alpha_4] = [0.1, 0.1, 0.05, 0.05]$$

- weight_update function: using **a range-only sensor**, it takes the input including the landmarks positions and landmark observations (range), and returns a list of weights for the particle set. The standard deviation of the Gaussian zero-mean measurement noise is σ=0.2.
- resampling function: same as the resampling function in 1D case.

## Extend to 2D case—Not mandatory

- Tips: To read in the sensor and landmark data, we have used dictionaries.



To access the sensor data from the `sensor_readings` dictionary, you can use

```
sensor_readings[timestamp,’sensor’][’id’]
sensor_readings[timestamp,’sensor’][’range’]
sensor_readings[timestamp,’sensor’][’bearing’]
```

and for odometry you can access the dictionary as

```
sensor_readings[timestamp,’odometry’][’r1’]
sensor_readings[timestamp,’odometry’][’t’]
sensor_readings[timestamp,’odometry’][’r2’]
```

To access the positions of the landmarks from `landmarks` dictionary , you can use

```
position_x = landmarks[id][0]
position_y = landmarks[id][1]
```