Particle Filter Example

Joey Wilson, Maani Ghaffari

February 23, 2024

Overview

1. Review Markov Filter
2. Particle Filter
3. Walk through Example
4. Notebook

Markov Localization Review

Bayes Filter Review

- In a previous recitation, we discussed Bayes filters
  - We applied Markov assumption and found a simple algorithm

Algorithm `Bayes-filter`

Require: Belief $bel(x_{t-1}) = p(x_{t-1}|z_{1:t-1}, u_{1:t-1})$, action u_t, measurement z_t;

1: **for** all state variables **do**

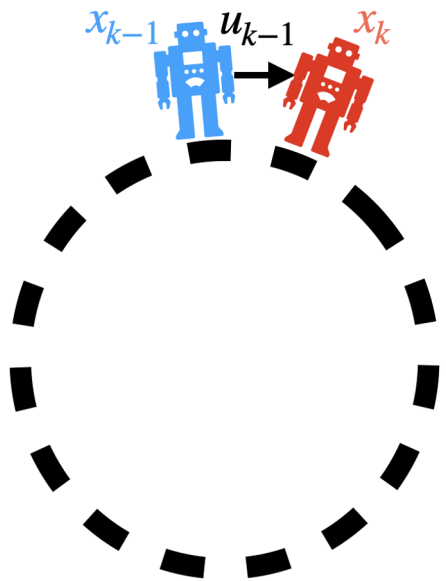
2: $\overline{bel}(x_t) = \int p(x_t|x_{t-1}, u_t) bel(x_{t-1}) \mathrm{d}x_{t-1}$ // Predict using action/control input u_t

3: $bel(x_t) = \eta p(z_t|x_t) \overline{bel}(x_t)$ // Update using perceptual data z_t

4: **return** $bel(x_t)$

Bayes Filter Review

- We then applied the filter to a simple problem
  - Robot walks around circle
  - Robot position discretized
  - Three different landmarks



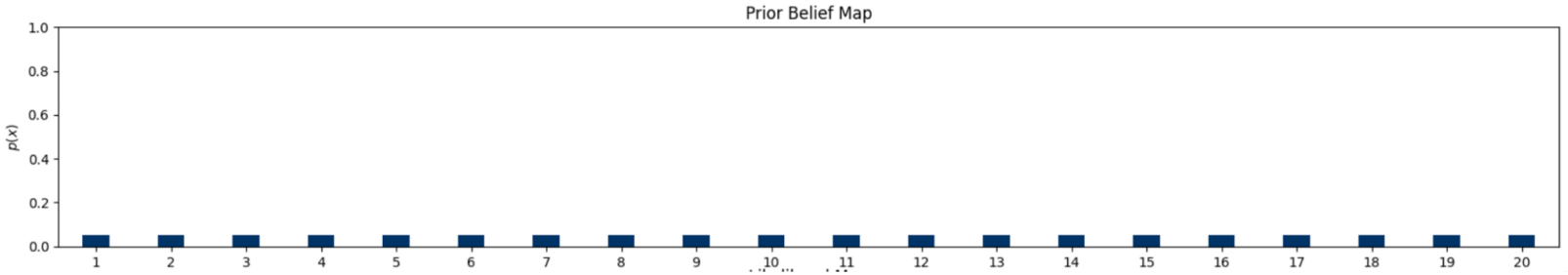
Update

- Check how likely each state is (likelihood)
- Update belief based on likelihood and normalize

```
eta = 0  # normalization constant
for i in range(len(X)):
    likelihood = measurement_model(X[i], likelihood_map, z=1)  # get measurement likelihood
    bel[:, i] = likelihood * bel_predicted[:, i]  # unnormalized Bayes update
    eta = eta + bel[:, i]
bel = bel / eta  # normalize belief
```

First Step

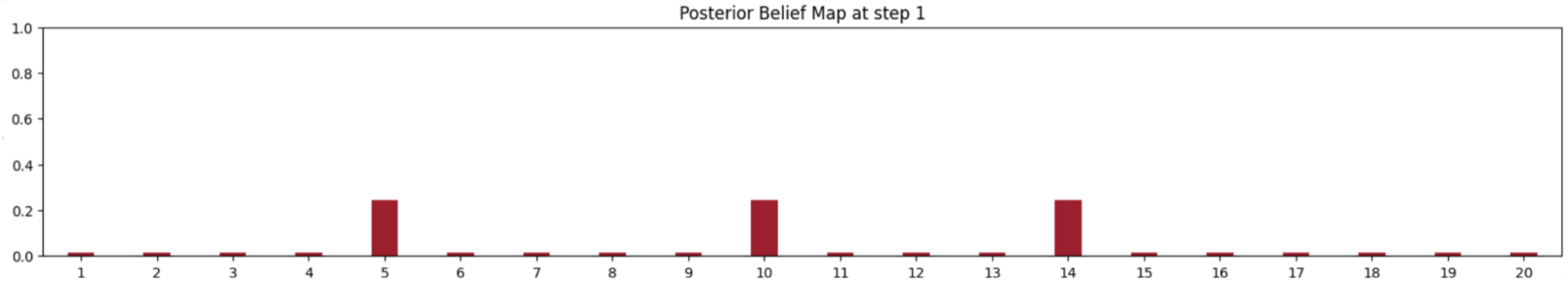
- Initialize state to uniform prior



- Suppose we detect a positive measurement and move forward. How will our posterior change?

Second Step

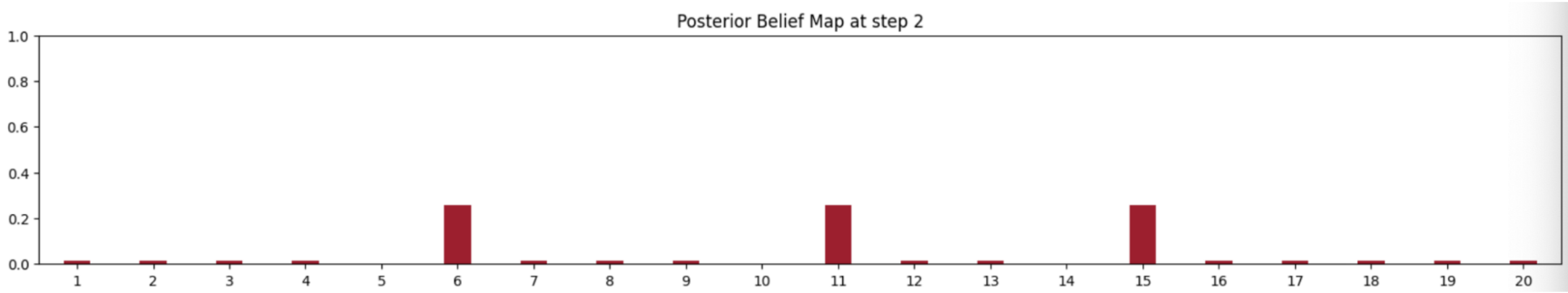
- More likely to be at locations 4, 9, or 13 before motion
- After motion, more likely locations 5, 10, 14



- Next, suppose we detect no measurement and move forward.

Third Step

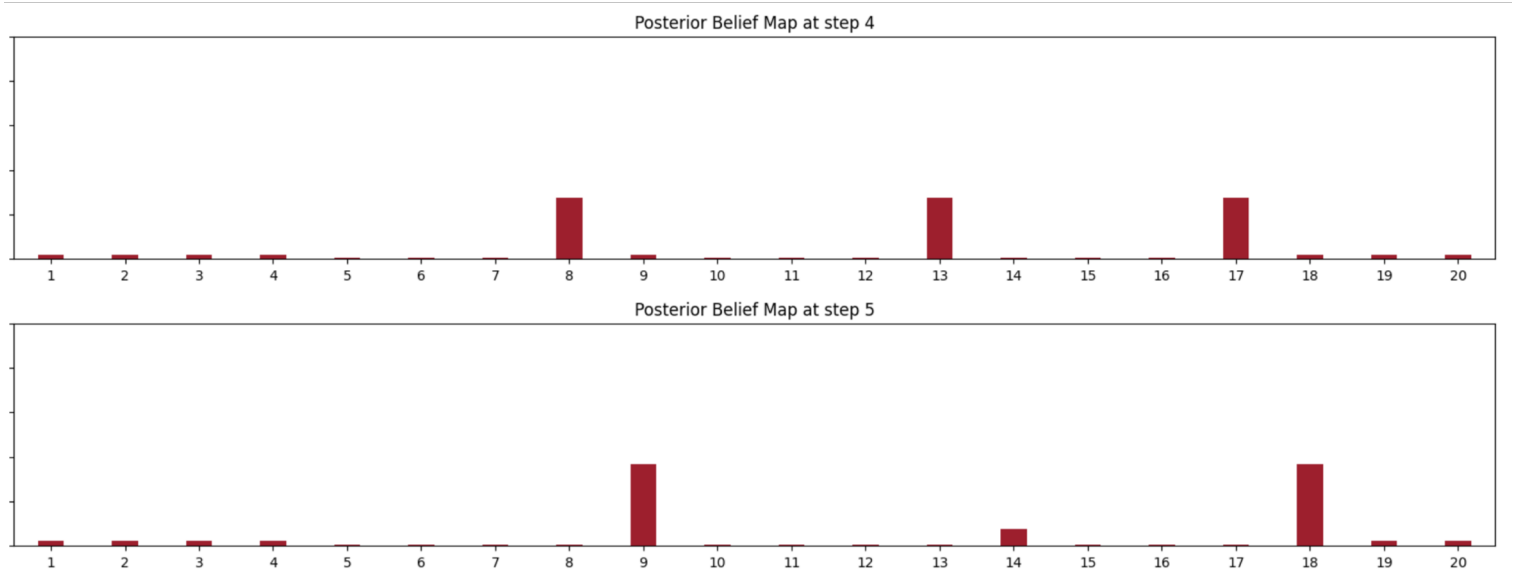
- Why do locations 5, 10, 14 have low probability?



- Next, we move a few more steps.

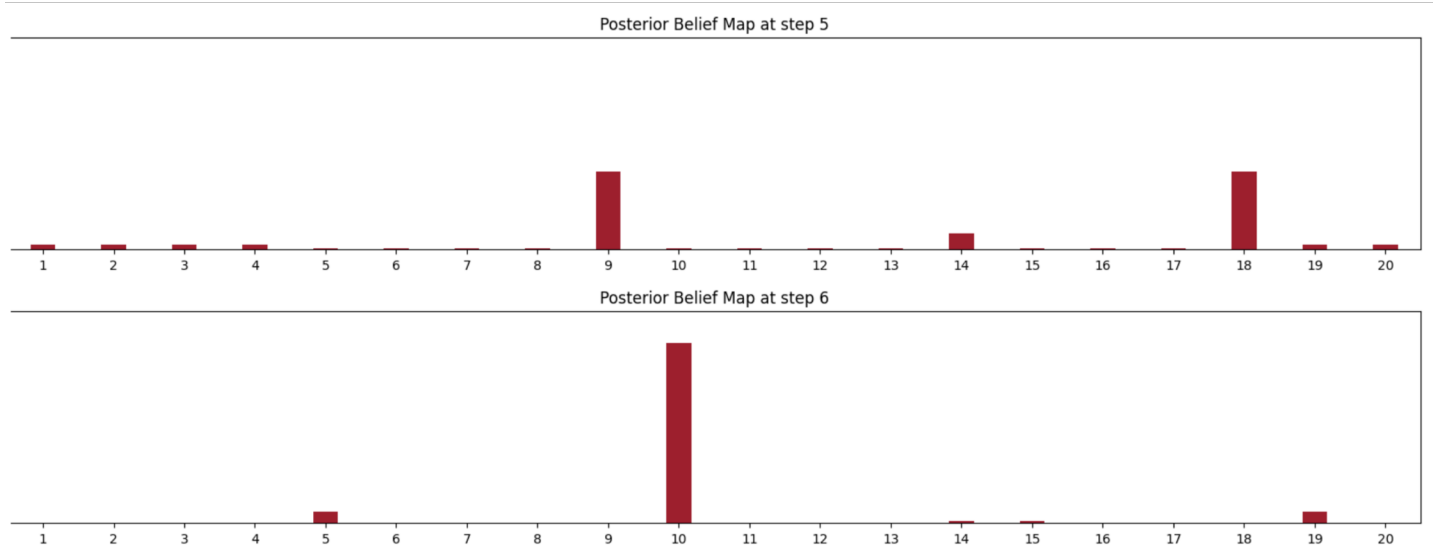
Fifth Step

- Negative sensor reading, which is unlikely at positions 4, 9, and 13



Sixth Step

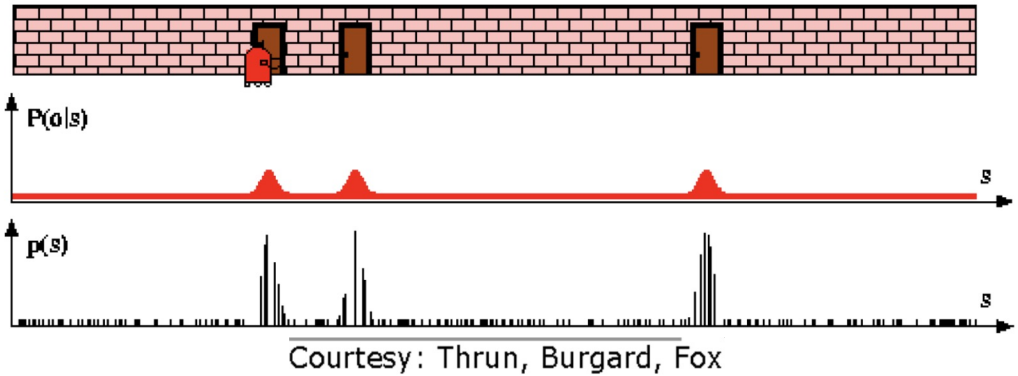
- Positive sensor reading, which is likely at positions 4, 9, and 13



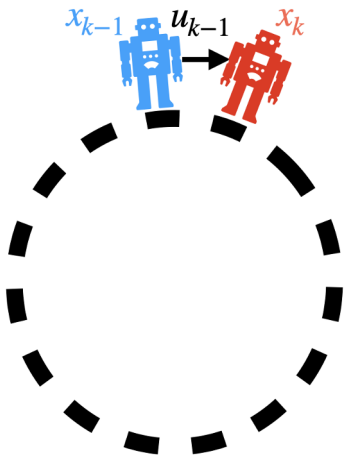
Particle Filter (Monte Carlo) Localization

Motivation

- Markov localization model was discretized
- What if we want a continuous representation of state?
  - Kalman filter provides this, but what if the model is highly nonlinear?
- Also, what if there are multiple modes?
- Solution: Represent the state using samples

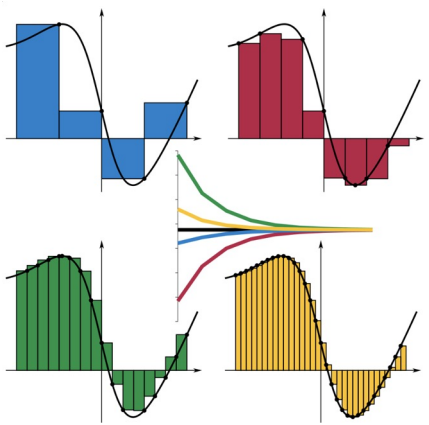


Courtesy: Thrun, Burgard, Fox



Monte Carlo Sampling Review

- Monte Carlo approaches approximate through random sampling
- Can be used for integration
  - Similar idea to Riemann sums
  - Except with random samples



https://en.wikipedia.org/wiki/Riemann_sum

Suppose we can simulate n independent and identically distributed (i.i.d.) random samples (particles), $\{x_i\}_{i=1}^n$ according to $p(x)$. An empirical estimate of this distribution is given by

$$p_n(x) = \frac{1}{n}\sum_{i=1}^{n}\delta_{x_i}(x)$$

Then, the following integral can be computed:

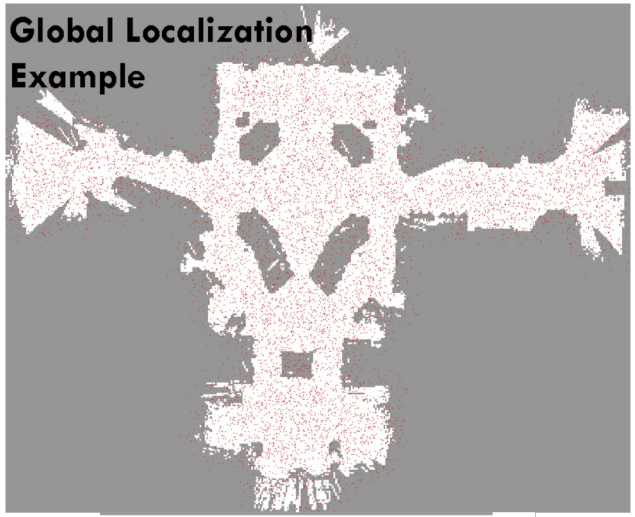
$$\mathrm{I}_n(f) = \int f(x)p_n(x)\mathrm{d}x = \frac{1}{n}\sum_{i=1}^{n} f(x_i),$$

which is an approximation of $\mathbb{E}[f(X)]$ under the true distribution $p(x)$:

$$\mathbb{E}[f(X)] \equiv \mathrm{I}(f) = \int f(x)p(x)\mathrm{d}x \approx \mathrm{I}_n(f)$$

Motivating Example

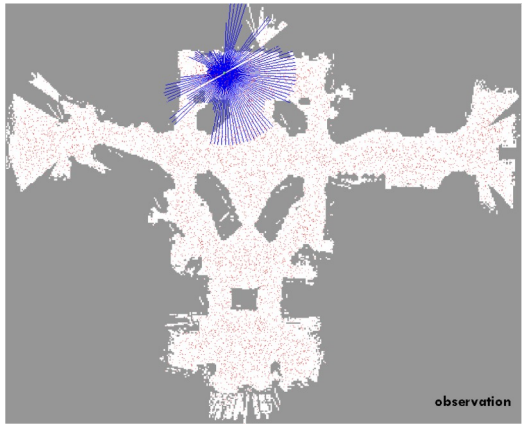
- Kidnapped robot problem (seen in ROB 550)
  - Robot is placed somewhere in environment with a map
  - Try to localize within the map
- Initialize to uniform distribution
  - Monte Carlo approximation of state
  - Importance weights initialized equally

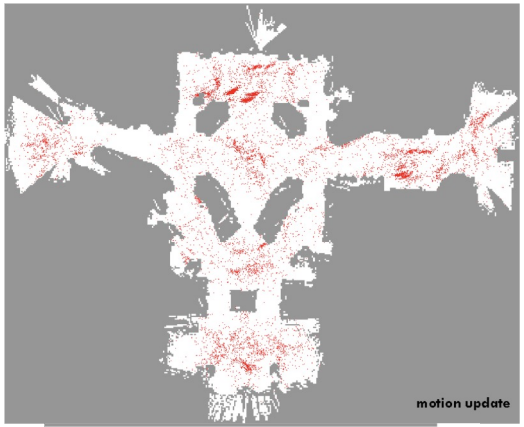


Courtesy: Thrun, Burgard, Fox

Motivating Example

- Over time, importance weights of each particle evolve as measurements are obtained
  - Similar to our Markov Localization example



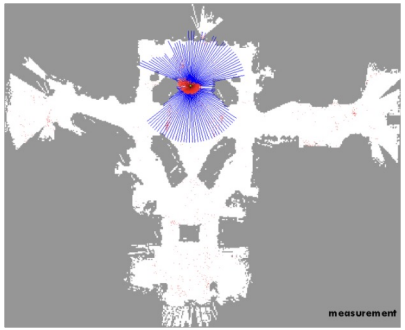


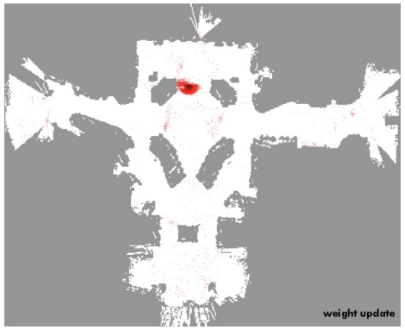
Courtesy: Thrun, Burgard, Fox

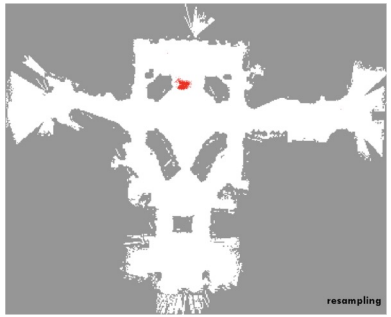
Particle Filter Localization

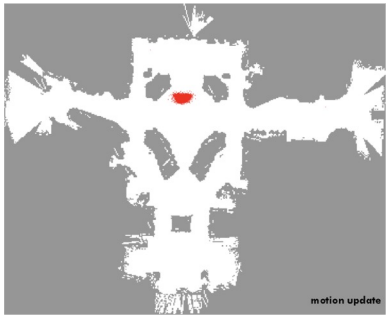
- Represent state as particles (samples), initialized to uniform weight
- At each time step:
  - **Propagate** the particles with motion
  - **Update** particle weights using measurement likelihood
    - What does the measurement tell us about each particle?
  - **Resample** to avoid weight degeneracy
    - Otherwise, weights conglomerate to one particle

Courtesy: Thrun, Burgard, Fox









Particle Filter Algorithm

Algorithm 3 `particle-filter`

Require: particles $\mathcal{X}_{k-1} = \{x_{k-1}^i, \tilde{w}_{k-1}^i\}_{i=1}^n$, action u_k, measurement z_k, resampling threshold n_t (e.g. $n/3$);

1: $\mathcal{X}_k \leftarrow \varnothing$
2: **for** each $x_{k-1}^i \in \mathcal{X}_{k-1}$ **do**
3: draw $x_k^i \sim p(x_k | x_{k-1}^i, u_k)$ ▷ sample from motion model
4: $w_k^i \leftarrow \tilde{w}_{k-1}^i p(z_k | x_k^i)$ ▷ update importance weights
5: $w_\mathrm{total} \leftarrow \sum_{i=1}^n w_k^i$ ▷ compute total weight to normalize importance weights
6: $\mathcal{X}_k \leftarrow \mathcal{X}_k \cup \{x_k^i, w_k^i / w_\mathrm{total}\}_{i=1}^n$ ▷ add weighted samples to the new set
7: $n_\mathrm{eff} \leftarrow 1 / \sum_{i=1}^n (\tilde{w}_k^i)^2$ ▷ compute effective sample size
8: **if** $n_\mathrm{eff} < n_\mathrm{t}$ **then**
9: $\mathcal{X}_k \leftarrow$ resample using $\mathcal{X}_k$ ▷ use a resampling algorithm to draw particles with higher weights
10: **return** $\mathcal{X}_k$

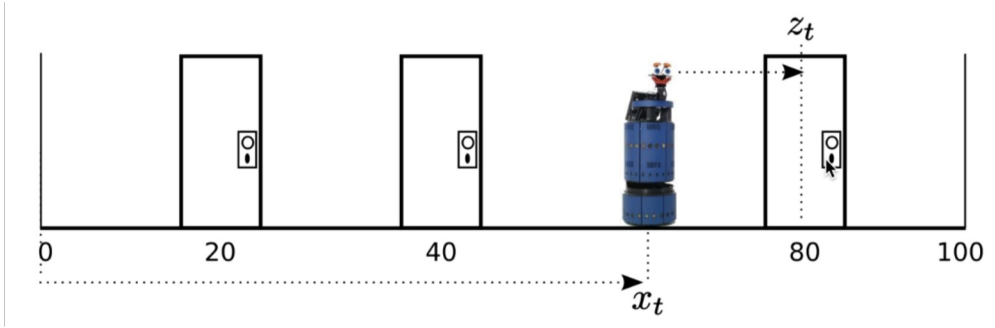
Additional Considerations

- When to resample?
  - Usually use effective sample size
- Particle degeneration: eventually, no particles near correct state due to resampling
  - Can add randomly selected particles during resampling
- How many particles?
  - Can adaptively adjust based on uncertainty

Implementation

Problem Statement

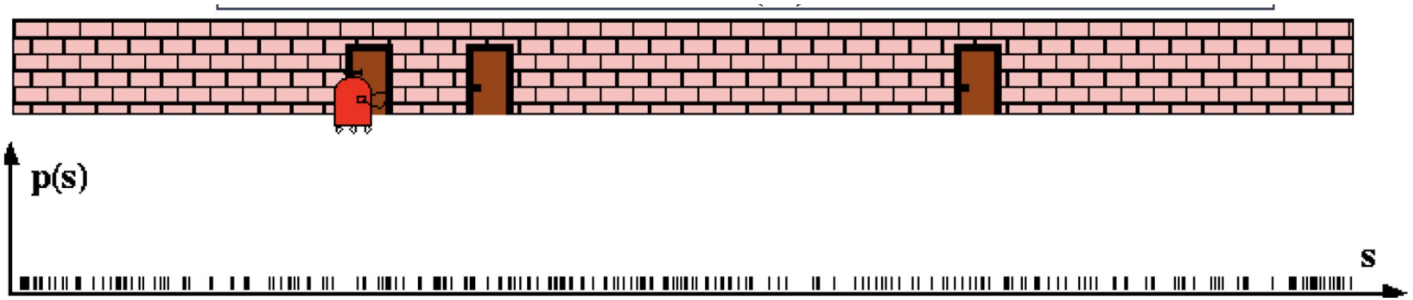
- Estimate robot state (x) given sequence of measurements (z) and controls (u)
- Robot in a corridor 100 m long
- Sensor measures distance to nearest door with Gaussian noise
- Motion also corrupted by Gaussian noise



Step 1: Initialize

- Uniformly distribute particles with equal weights 1/N

```
# Uniformly distribute particles
def initialize_particles(self):
    ## initialize the particles with uniform distributions.
    # N samples
    self.X_t = np.random.uniform(self.space[0], self.space[1], size=self.N)
    # Equal weights
    self.W_t = np.ones(self.N)/self.N
```



Step 2: Propagate Motion

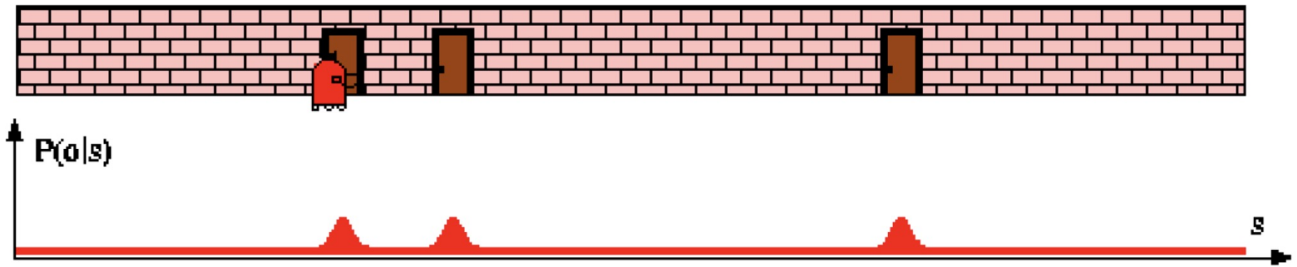
- For each particle, sample a new location

```
# Next pose of a particle
def sample_pose(self, x_tm1, u_t):
    ## sample the particles based on the odometry models and noise distributions.
    # Sample of noise
    x_t = x_tm1 + u_t + np.random.normal(0, self.odom_sig)
    # Clip at boundaries
    x_t = np.clip(x_t, self.space[0], self.space[1])
    return x_t
```

Step 3: Calculate Likelihood

- For each particle, calculate likelihood of measurement

```
def likelihood_of_z_given_x(self, z_t, x_t):
    # Which door is nearest
    i = np.argmin(np.abs(self.map - x_t))
    # Location of nearest door
    door_loc = self.map[i]
    # Distance
    z_hat = abs(door_loc - x_t)
    # Probability of measurement given distance
    p = norm.pdf(z_t, loc=z_hat, scale=self.sens_sig)
    return p
```



Step 4: Update Weights

- Calculate new weights using likelihood

```
## for each particle
for i in range(self.N):
    ## update the weights based on measurement
    self.W_t[i] = self.W_t[i] * self.likelihood_of_z_given_x(z_t, self.X_t[i])
w_tot = np.sum(self.W_t)
self.W_t /= w_tot ## normalize the weights so they sum to 1
```

Step 5: Resample

- Resample particles according to updated weights

Algorithm 4 `low-variance-resampling`

Require: particles $\mathcal{X}_k = \{x_k^i, \tilde{w}_k^i\}_{i=1}^n$;

1: $w_c \leftarrow$ compute the vector of cumulative sum of the weights using $\{\tilde{w}_k^i\}_{i=1}^n$ ▷ w_c is the Cumulative Distribution Function (CDF)
2: $r \leftarrow \mathbf{rand}(0, n^{-1})$ ▷ draw a uniform random number between 0 and n^{-1}
3: $j \leftarrow 1$ ▷ dummy index to climb the CDF and select particles
4: **for all** $i \in \{1:n\}$ **do**
5: $\quad u \leftarrow r + (i-1)n^{-1}$ ▷ move along the CDF
6: $\quad$ **while** $u > w_c^j$ **do**
7: $\quad\quad j \leftarrow j + 1$
8: $\quad x_k^i \leftarrow x_k^j$ ▷ replicate the survived particle
9: $\quad \tilde{w}_k^i \leftarrow n^{-1}$ ▷ set the weight to n^{-1} (uniform distribution)
10: **return** $\mathcal{X}_k$

```python
def resample(self):
    M = len(self.W_t)
    Minv = 1/M
    X_t_bar = np.zeros(M)
    W_t_bar = np.ones(M)*Minv
    # r is a random offset so sampling is not deterministic
    r = np.random.uniform(0, Minv)
    print("r:", r)
    c = self.W_t[0]
    i = 0
    # For each particle
    for m in range(M):
        U = r + m*Minv
        print("U:", U)
        # Skip over particles until c >= U
        while U > c:
            i += 1
            c += self.W_t[i]
        # Add current particle
        X_t_bar[m] = self.X_t[i]
        print("Added particle", i, " c ==", c)
    self.X_t = X_t_bar
    self.W_t = W_t_bar
```

Step 5: Resample

- r is a random offset so the algorithm is not deterministic
- Samples assigned based on cumulative CDF

Algorithm 4 `low-variance-resampling`

Require: particles $\mathcal{X}_k = \{x_k^i, \tilde{w}_k^i\}_{i=1}^n$;

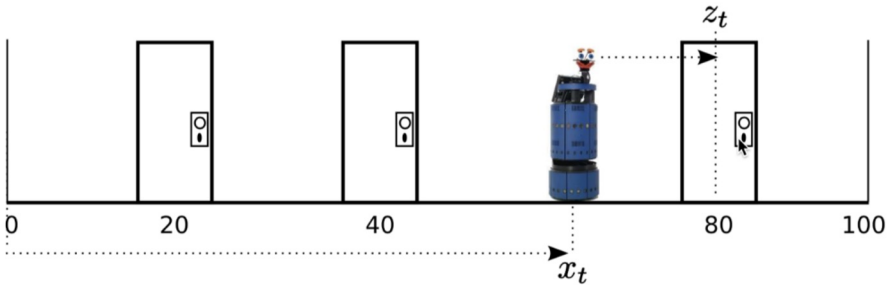
1: $w_c \leftarrow$ compute the vector of cumulative sum of the weights using $\{\tilde{w}_k^i\}_{i=1}^n$
▷ w_c is the Cumulative Distribution Function (CDF)
2: $r \leftarrow \texttt{rand}(0, n^{-1})$ ▷ draw a uniform random number between 0 and n^{-1}
3: $j \leftarrow 1$ ▷ dummy index to climb the CDF and select particles
4: **for** all $i \in \{1 : n\}$ **do**
5: $u \leftarrow r + (i-1)n^{-1}$ ▷ move along the CDF
6: **while** $u > w_c^j$ **do**
7: $j \leftarrow j + 1$
8: $x_k^i \leftarrow x_k^j$ ▷ replicate the survived particle
9: $\tilde{w}_k^i \leftarrow n^{-1}$ ▷ set the weight to n^{-1} (uniform distribution)
10: **return** $\mathcal{X}_k$

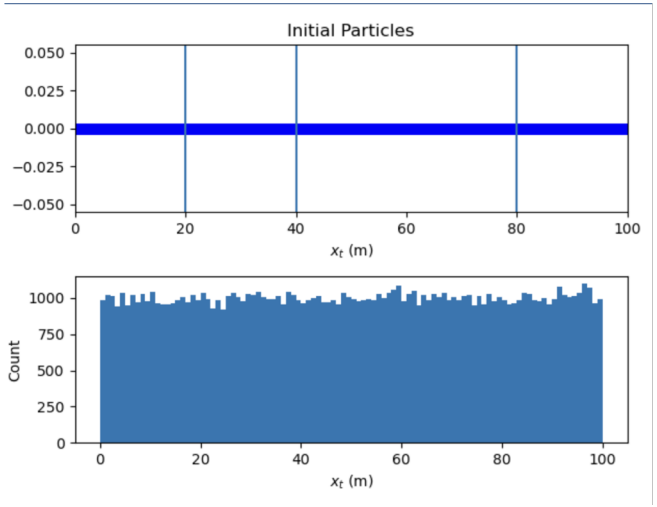
```
r: 0.00032339259849476545
U: 0.00032339259849476545
Added particle 0  c == 0.013723097225879475
U: 0.004323392598494766
Added particle 0  c == 0.013723097225879475
U: 0.008323392598494765
Added particle 0  c == 0.013723097225879475
U: 0.012323392598494765
Added particle 0  c == 0.013723097225879475
U: 0.016323392598494767
Added particle 24  c == 0.08156506282011926
U: 0.020323392598494767
Added particle 24  c == 0.08156506282011926
U: 0.024323392598494767
Added particle 24  c == 0.08156506282011926
U: 0.028323392598494767
Added particle 24  c == 0.08156506282011926
U: 0.03232339259849477
Added particle 24  c == 0.08156506282011926
U: 0.03632339259849477
Added particle 24  c == 0.08156506282011926
U: 0.04032339259849477
Added particle 24  c == 0.08156506282011926
U: 0.044323392598494764
Added particle 24  c == 0.08156506282011926
...
U: 0.9923233925984948
Added particle 234  c == 0.9925754306313397
U: 0.9963233925984948
Added particle 244  c == 1.0
```

Output

First Frame

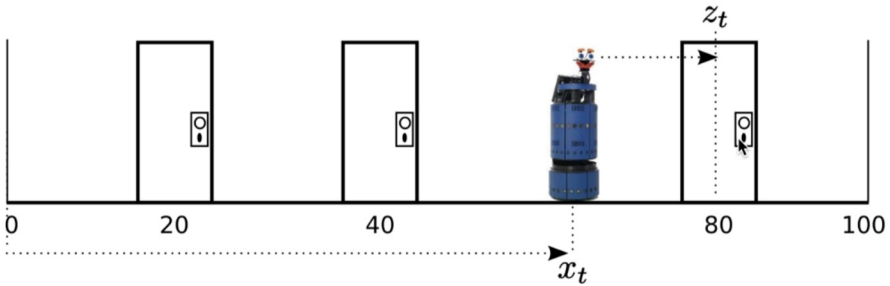
- Initialize uniformly
- At first step, move 10 meters and obtain measurement that nearest door is 10 meters away $\boldsymbol{u}_1 = 10\ \mathrm{m}, \boldsymbol{z}_1 = 10\ \mathrm{m}$

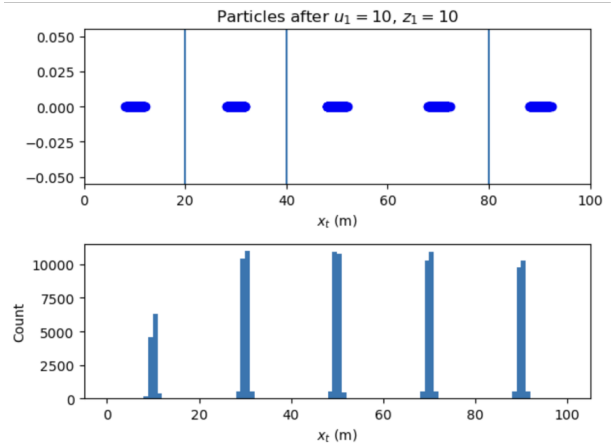




Second Frame

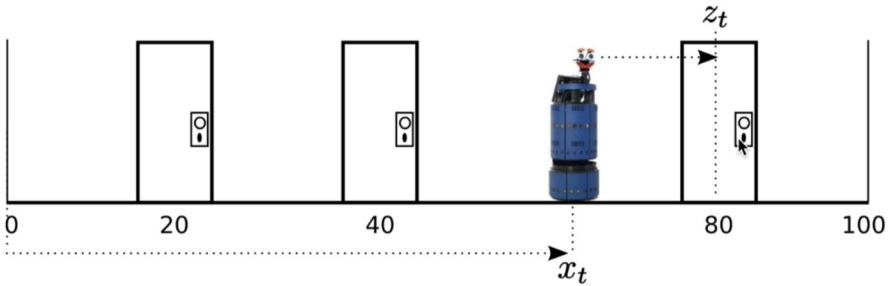
- Any position ~10 meters from a door is equally likely (except for 10 meters, since moved 10 meters right)
- Next: Move 20 meters and obtain measurement that nearest door is 10 meters away

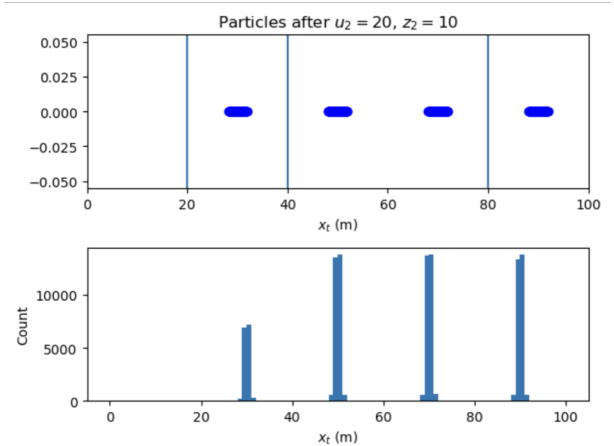




Third Frame

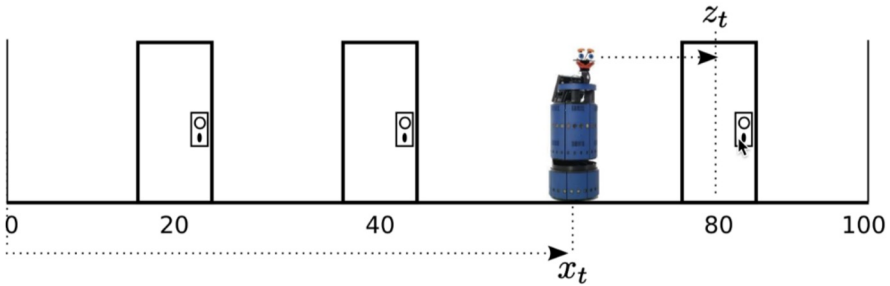
- Mode around 30m still less likely, others equally likely except final mode gone due to clipping
- Next: Move 20 meters and obtain measurement that nearest door is 10 meters away

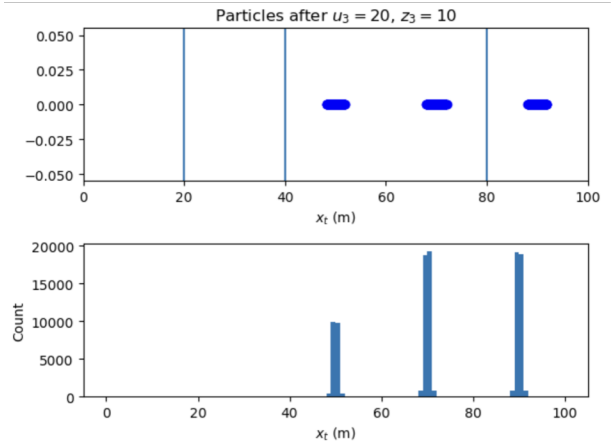




Fourth Frame

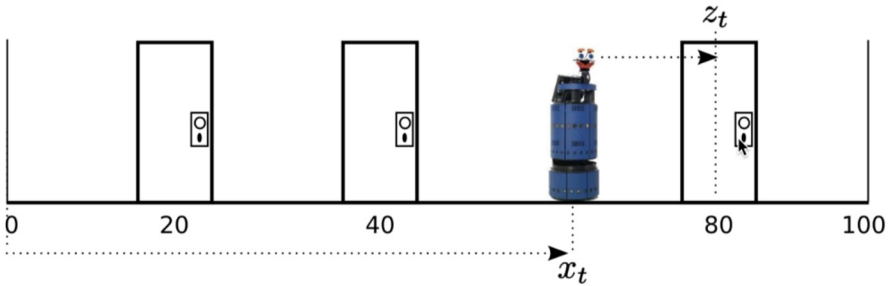
- Mode around 50m still less likely, mode from 90m gone due to clipping
- Next: Move -**10** meters and obtain measurement that nearest door is **20** meters away

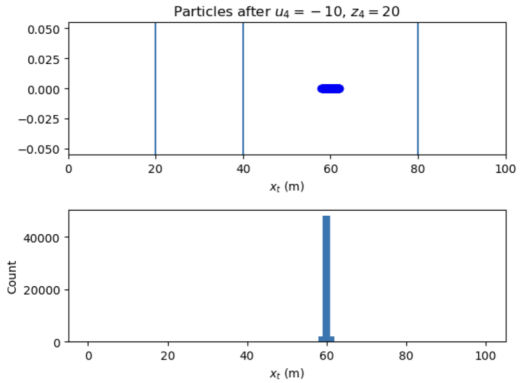




Fifth Frame

- Particles which moved from 50 m to 40 m seem possible, however the measurement would be distance of about 0, so likelihood ~0





Demo