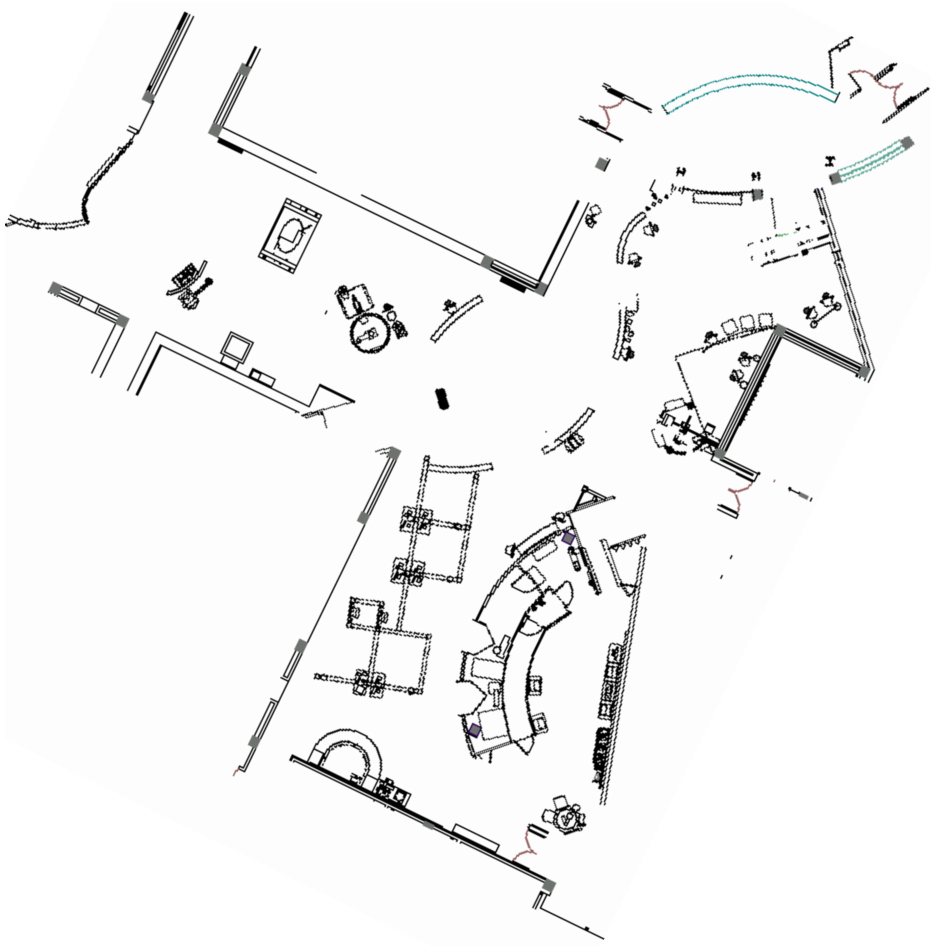
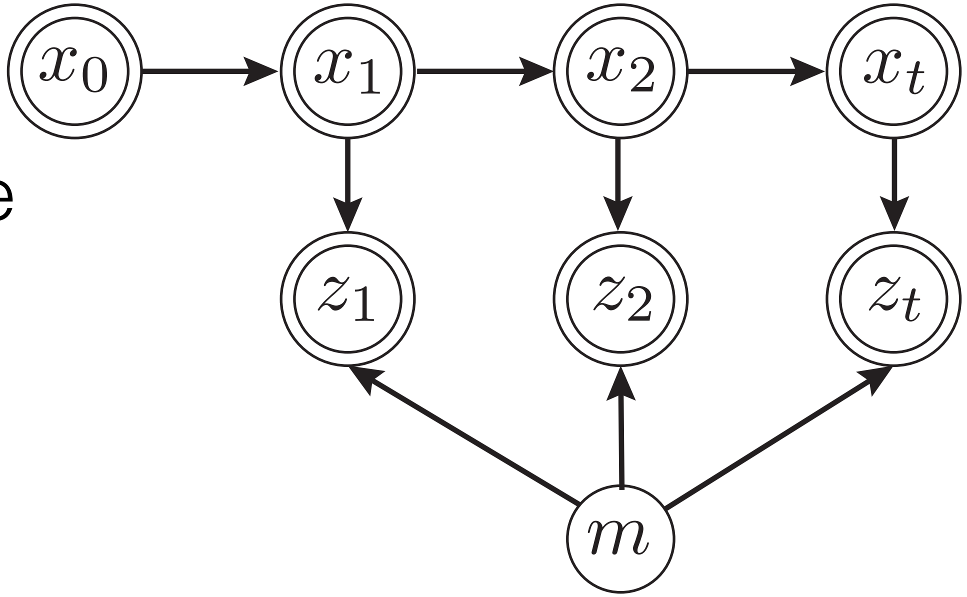
Occupancy Grid Mapping

Occupancy Grid Map



Using Conditional Probability to make a map

- Given robot poses $x_{1:t}$ and laser rangefinder measurements $z_{1:t}$ infer a map of the environment
- Expressed probabilistically as $p(m|z_{1:t}, x_{1:t})$



- Double circle is known variable
- Arrow indicate generation
- Single circle is latent variable

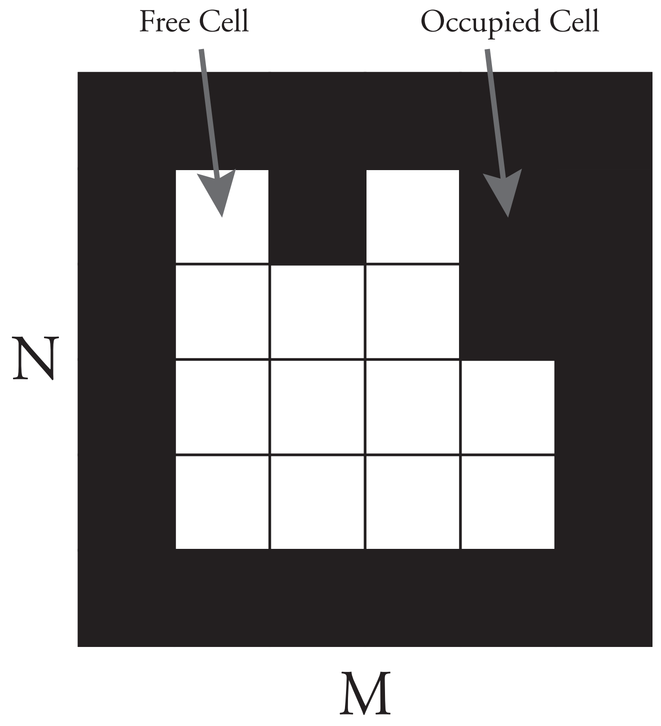
Occupancy Grid Mapping

- Map is a $M \times N$ matrix of cells
- Cell is either occupied or unoccupied
- Probability $p(m_i)$ is probability cell is occupied

$$p(m|x_{1:t}, z_{1:t}) = \prod_i p(m_i|x_{1:t}, z_{1:t})$$

- Map can be inferred from a Bayes filter with a static state

$$m = \{m_i\}_{M \times N}$$



Odds Ratio & Log Odds

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

- A is binary state $occ(i,j)$ and B is sensor reading $r = D$
- **Probability** a cell is occupied $p(occ(i,j)) = p(A)$ has range $[0,1]$
- Probability a cell is free $p(\neg A)$
- **Odds** of being occupied $o(occ(i,j)) = p(A)/p(\neg A)$ has range $[0,\infty]$
- **Log odds** $\log o(occ(i,j))$ has range $[-\infty,\infty]$
- Each cell $C(i,j)$ holds the value of log $o(occ(i,j))$
- $C(i,j) = 0$ corresponds to $p(occ(i,j)) = 0.5$

Bayes' Law using Odds

- Bayes' Law:

$$p(A|B) = \frac{p(B|A)p(A)}{p(B)}$$

- Likewise:

$$p(\neg A|B) = \frac{p(B|\neg A)p(\neg A)}{p(B)}$$

- So:

$$o(A|B) = \frac{p(A|B)}{p(\neg A|B)} = \frac{p(B|A)p(A)}{p(B|\neg A)p(\neg A)}$$

$$= \lambda(B|A)o(A)$$

- Where:

$$\lambda(B|A) = \frac{p(B|A)}{p(B|\neg A)}$$

Updating the map using Bayes' Law

- Bayes' Law can be written as

$$o(A|B) = \lambda(B|A)o(A)$$

posterior Sensor update prior

- Take log odds to make multiplication into addition

$$\log o(A|B) = \log \lambda(B|A) + \log o(A)$$

- For each cell add the evidence $\log \lambda(B|A)$ into the cells log odds

Prior and Sensor Update

- Prior
- Initially 0 if map unknown

$$l_0 = \log \frac{p(\mathbf{m}_i = 1)}{p(\mathbf{m}_i = 0)} = \log \frac{p(\mathbf{m}_i)}{1 - p(\mathbf{m}_i)}$$

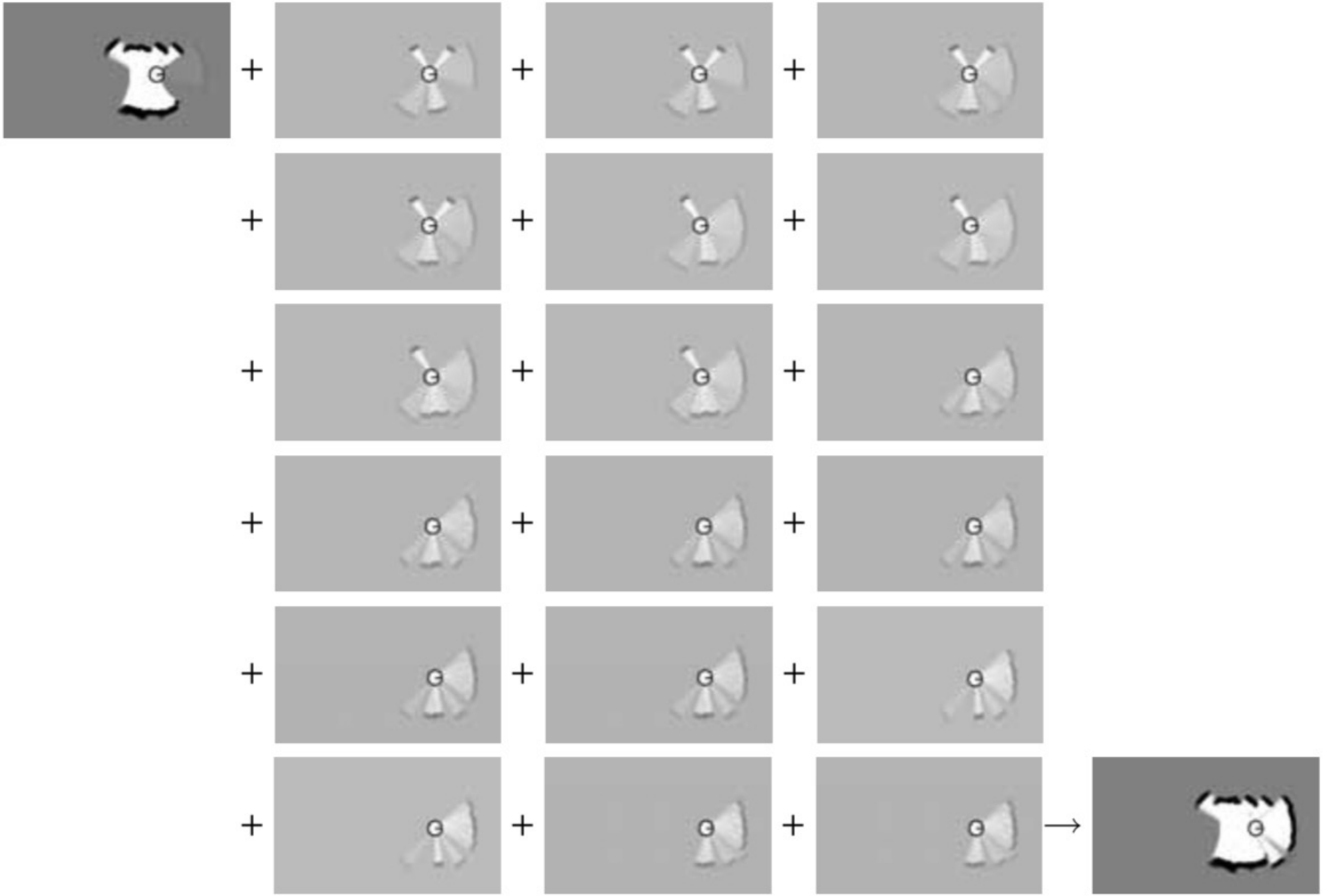
- Sensor Update

$$l_{t,i} = \log \frac{p(\mathbf{m}_i \mid z_{1:t}, x_{1:t})}{1 - p(\mathbf{m}_i \mid z_{1:t}, x_{1:t})}$$

Map Update Algorithm

Algorithm occupancy_grid_mapping($\{l_{t-1,i}\}, x_t, z_t$**):**

  for all cells $\mathbf{m}_i$ *do*
    if $\mathbf{m}_i$ *in perceptual field of* z_t *then*
      $l_{t,i} = l_{t-1,i} + \textbf{inverse\_sensor\_model}(\mathbf{m}_i, x_t, z_t) - l_0$
    else
      $l_{t,i} = l_{t-1,i}$
    endif
  endfor
  return $\{l_{t,i}\}$



Inverse Sensor Model

$$\textbf{inverse\_sensor\_model}(\mathbf{m}_i, x_t, z_t) \quad = \quad \log \frac{p(\mathbf{m}_i \mid z_t, x_t)}{1 - p(\mathbf{m}_i \mid z_t, x_t)}$$

```
Algorithm inverse_range_sensor_model(m_i, x_t, z_t):
    Let x_i, y_i be the center-of-mass of m_i
    r = sqrt((x_i − x)^2 + (y_i − y)^2)
    φ = atan2(y_i − y, x_i − x) − θ
    k = argmin_j |φ − θ_{j,sens}|
    if r > min(z_max, z_t^k + α/2)
        return l_0
    if z_t^k < z_max and |r − z_t^k| < α/2
        return l_occ
    if r ≤ z_t^k
        return l_free
    endif
```

r – distance to cell

ϕ – angle to cell

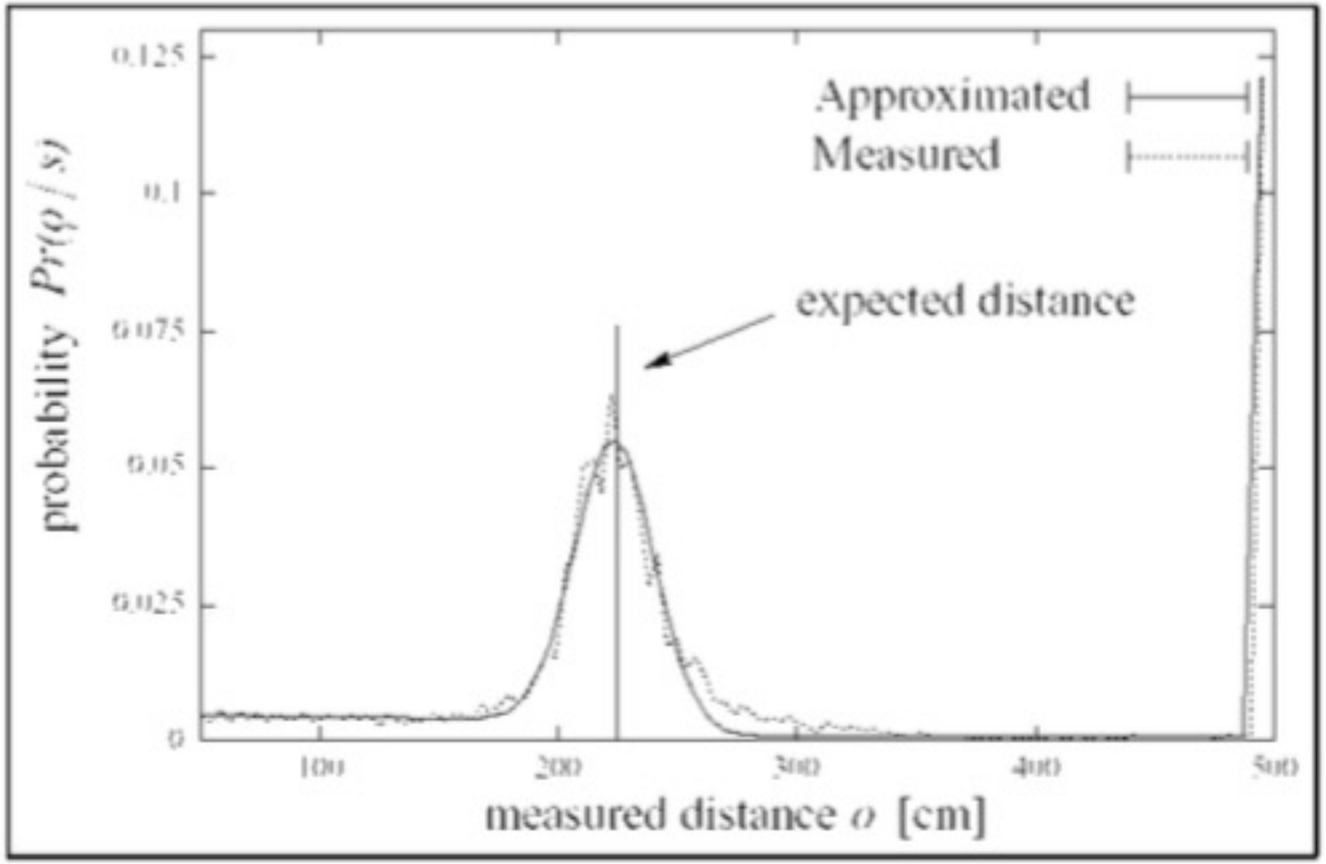
α – dimension of cell

k – beam index

z_t^k - reading k from scan at time t

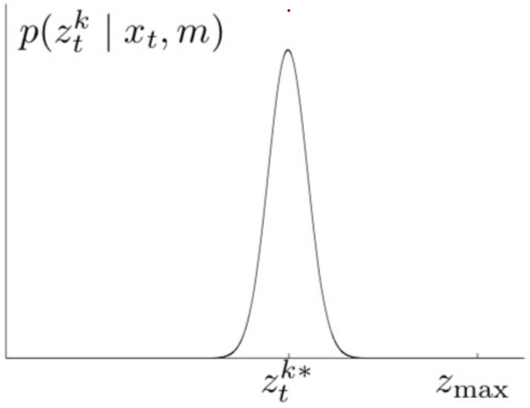
Sensor Model

Probability of reading a range given known occupancy at known distance

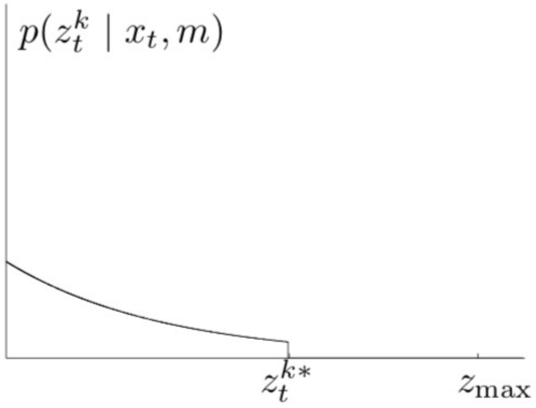


Sensor Model: Beam Model

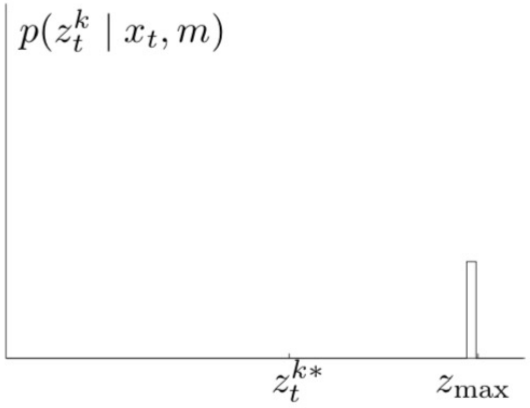
(a) Gaussian distribution p_{hit}



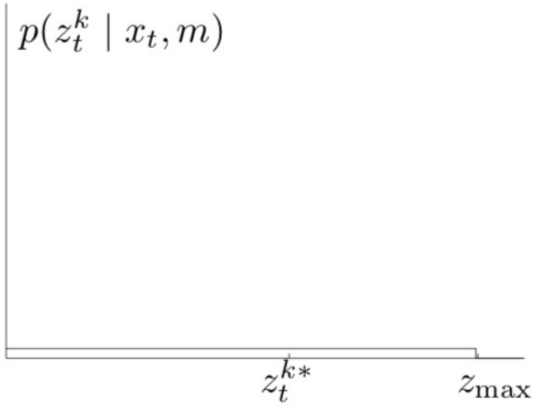
(b) Exponential distribution p_{short}



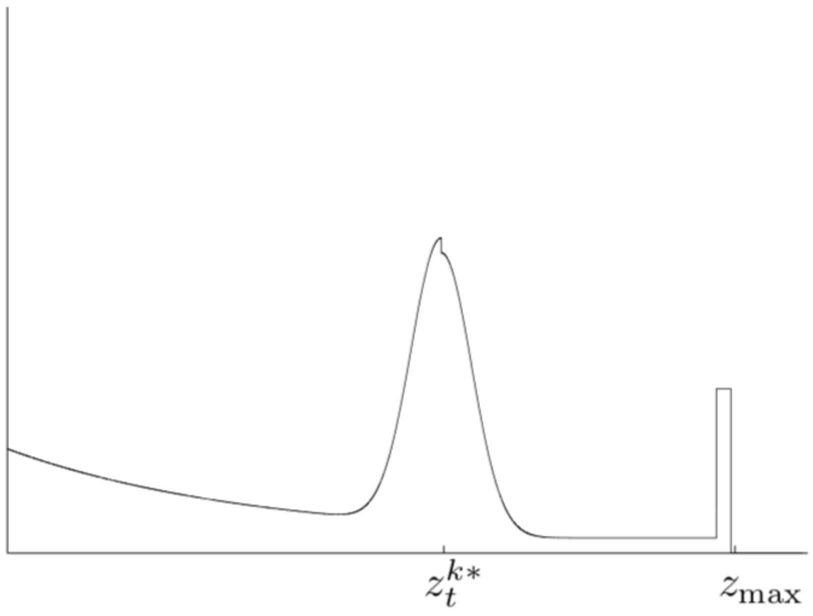
(c) Uniform distribution $p_{\max}$



(d) Uniform distribution p_{rand}



$$p(z_t^k \mid x_t, m) \quad = \quad \begin{pmatrix} z_{\text{hit}} \\ z_{\text{short}} \\ z_{\max} \\ z_{\text{rand}} \end{pmatrix}^T \cdot \begin{pmatrix} p_{\text{hit}}(z_t^k \mid x_t, m) \\ p_{\text{short}}(z_t^k \mid x_t, m) \\ p_{\max}(z_t^k \mid x_t, m) \\ p_{\text{rand}}(z_t^k \mid x_t, m) \end{pmatrix}$$



Inverse Sensor Model

- If laser terminates at C_{ij} at distance D

$$\lambda(z = D|occ(i,j)) = \frac{p(z = D|occ(i,j))}{p(z = D|\neg occ(i,j))} \approx \frac{.06}{.005} = 12$$

$$\log_2 \lambda \approx +3.5$$

- If the laser passes through C_{ij}

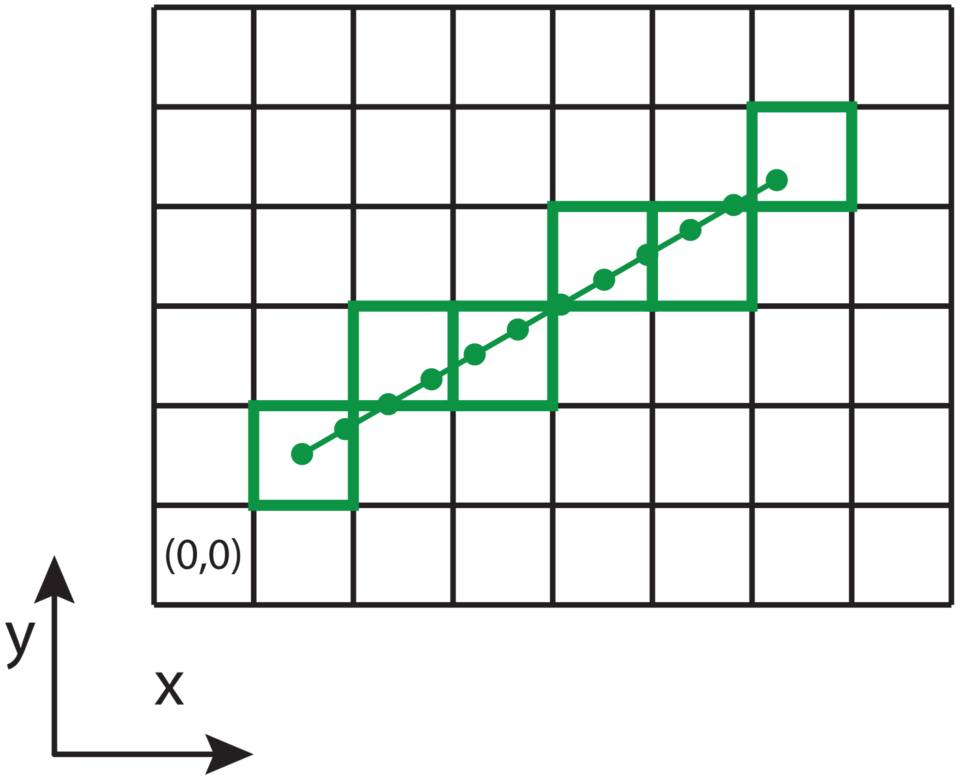
$$\lambda(z > D|occ(i,j)) = \frac{p(z > D|occ(i,j))}{p(z > D|\neg occ(i,j))} \approx \frac{.45}{.9} = 0.5$$

$$\log_2 \lambda \approx -1.0$$

Implementation

- Find endpoint of each ray on map grid and update
- Rasterize each laser ray into the map to determine cells that are currently visible and free or occupied
- Convert known pose (x, y, θ) to start cell and reading (θ,d) to end cell in the map
- Compute new log odds for each cell the ray touches
- Can divide ray into steps and check each cell along the ray
- Can use Breshenham's algorithm to update cells along the ray

Divide and Step Along Ray

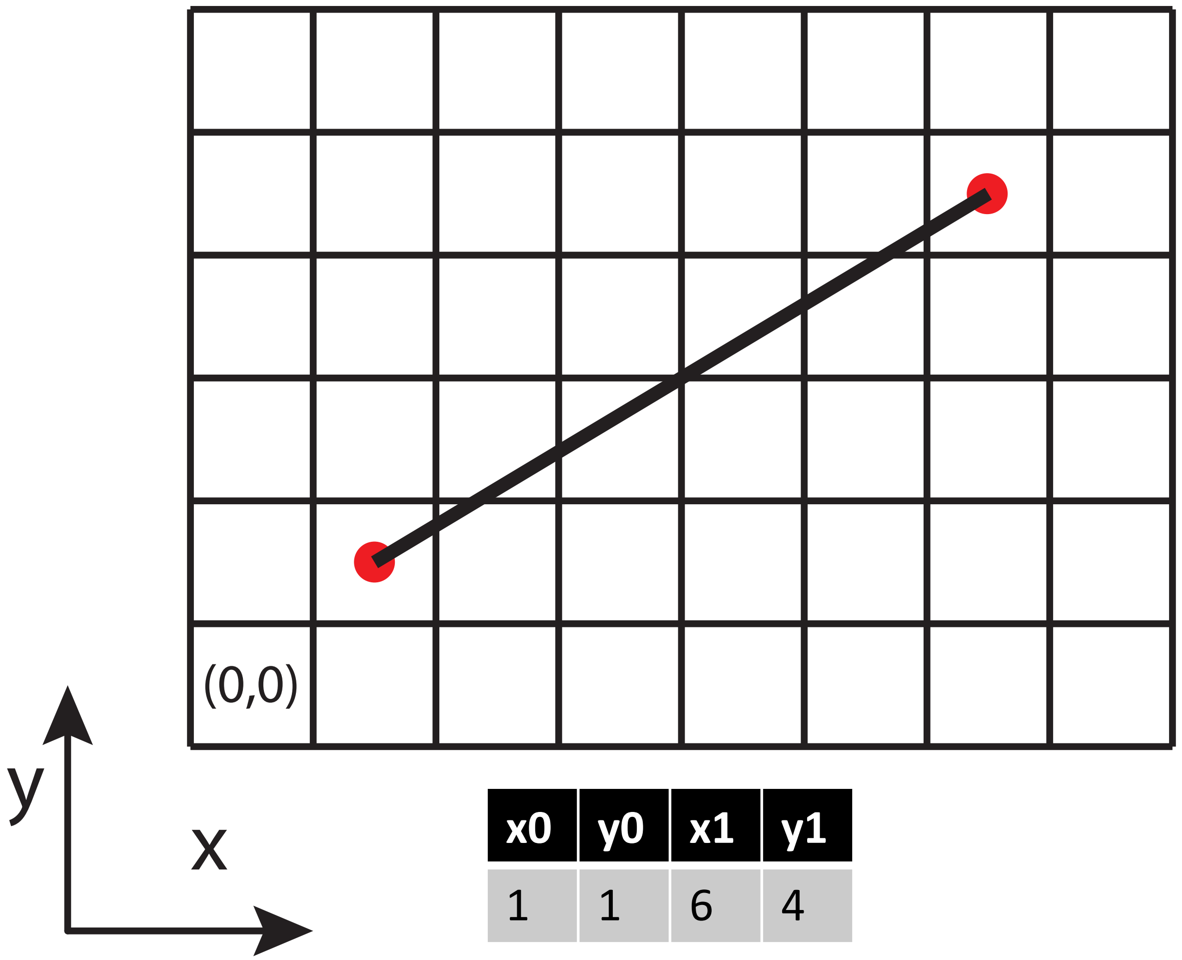


- Divide ray into ½ cell steps
- Check cell each step touches, but ensure you don't update cell twice
- In this case 12 iterations of loop
- Floating point math

Breshenham's Algorithm – Integer Math

```
dx = abs(x1-x0);
dy = abs(y1-y0);
sx = x0<x1 ? 1 : -1;
sy = y0<y1 ? 1 : -1;
err = dx-dy;
x = x0;
y = y0;

while(x != x1 || y != y1){
    updateOdds(x,y);
    e2 = 2*err;
    if (e2 >= -dy){
           err -= dy;
           x += sx;
    }
    if (e2 <= dx){
           err += dx
           y += sy
    }
}
```

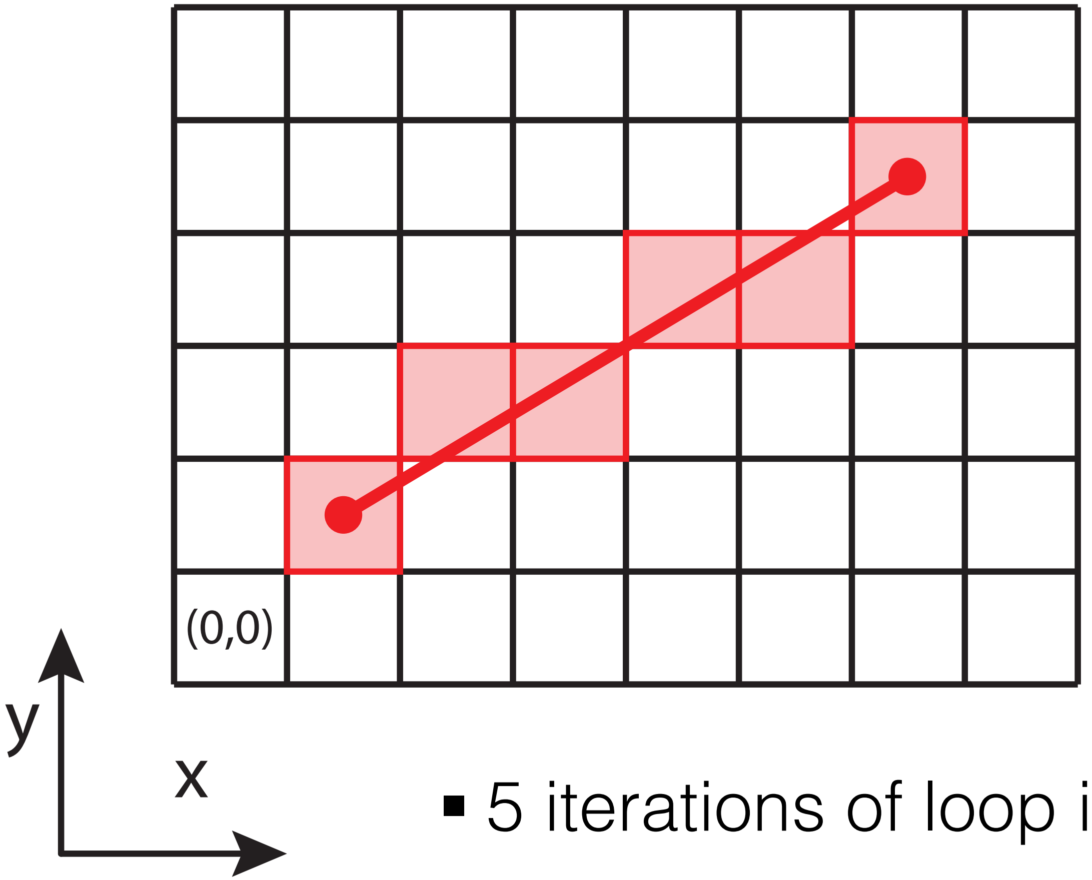


x0	y0	x1	y1
1	1	6	4

Breshenham's Algorithm

```
dx = abs(x1-x0);
dy = abs(y1-y0);
sx = x0<x1 ? 1 : -1;
sy = y0<y1 ? 1 : -1;
err = dx-dy;
x = x0;
y = y0;

while(x != x1 || y != y1){
    updateOdds(x,y);
    e2 = 2*err;
    if (e2 >= -dy){
           err -= dy;
           x += sx;
    }
    if (e2 <= dx){
           err += dx
           y += sy
    }
}
```

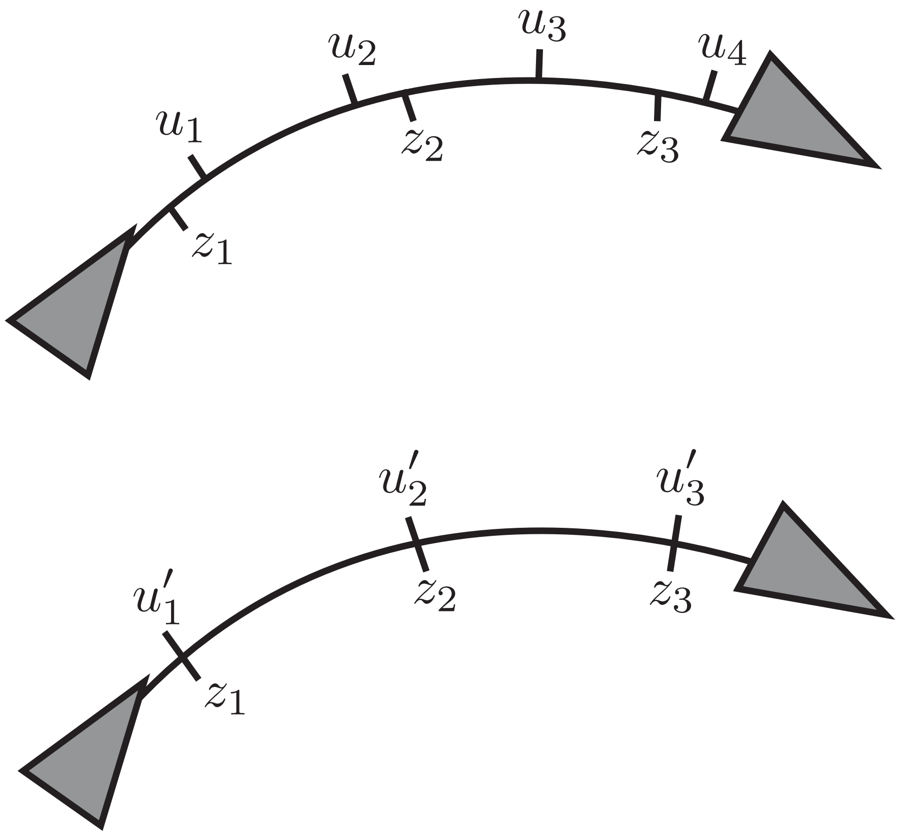


e2	x	y	err
4	2	2	4
8	3	2	1
2	4	3	3
6	5	3	0

- 5 iterations of loop in this case
- All integer math

Interpolating Observations

- Odometry and laser scans happen at different rates and arrive at different times.
- Interpolation gives odometry readings aligned with the laser scan times.



Interpolating Poses – Pose Trace

- Scans and Odometry happen at different rates…
- Already implemented in `OccupancyGridSlam::copyDataForSLAMUpdate()`
- Found in `common/pose_trace.cpp`
- Adds `pose_xyt_t` from odometry to a vector
- Interpolates x, y and θ linearly and assigns new timestamp
- Access this vector later to get interpolated poses that match the timestamps of the LIDAR scan

Moving Laser Scan in Mapping

```
struct adjusted_ray_t
{
    Point<float> origin;  ///< Position of the robot when the ray was measured
    float        range;   ///< Range of the measurement
    float        theta;   ///< Angle in global frame, not robot frame
};
```

```
void Mapping::updateMap(...){
        MovingLaserScan movingScan(scan, previousPose, pose);

        for(auto& ray : movingScan){
                scoreEndpoint(ray, map);
        }

        for(auto& ray : movingScan){
                scoreRay(ray, map);
        }
        previousPose_ = pose;
}
```

Localization, Particle Filter & Action Model

Localization: *"Where am I?"*

- The occupancy grid mapping method we covered assumes we know precisely the location (x, y, θ) in the reference frame of the map.
- The method we have used for localization in the past, odometry, is not accurate over time, as you should have discovered.
- We need to use other sensors to re-localize at each step and determine a sufficiently accurate pose

Localization Problem

Given:

- Map of features: $\mathbf{m} = \{m_1, m_2, ..m_n\}$
- Sensor Measurements: $\mathbf{Z}_{0:k} = \{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_k\}$

Desired:

- Path of robot: $\mathbf{X}_{0:k} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k\}$

$$p(\mathbf{X}_{0:k} | m, \mathbf{Z}_{0:k})$$

SLAM Problem

Given:

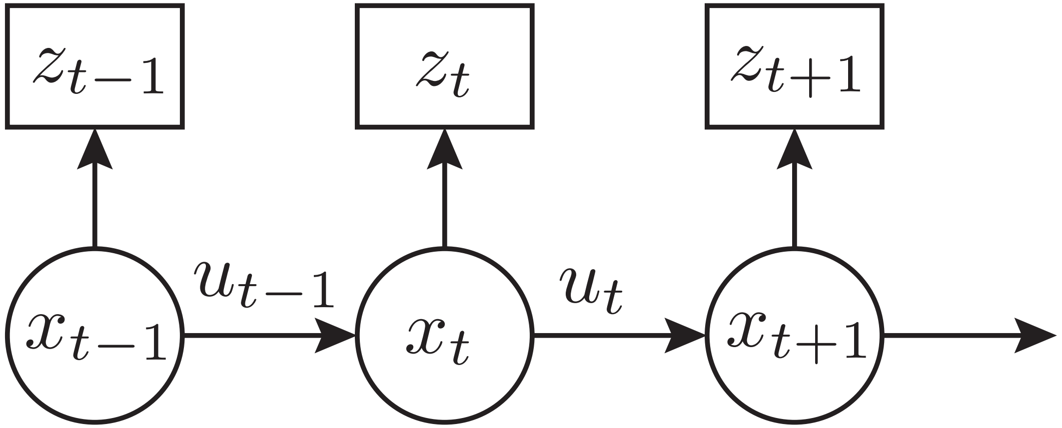
- Robot Commands: $\mathbf{U}_{0:k} = \{u_1, u_2, ..., u_k\}$
- Sensor Measurements: $\mathbf{Z}_{0:k} = \{\mathbf{z}_1, \mathbf{z}_2, ..., \mathbf{z}_k\}$

Desired:

- Map of features: $\mathbf{m} = \{m_1, m_2, ..m_n\}$
- Path of robot: $\mathbf{X}_{0:k} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_k\}$

$$p(m, \mathbf{X}_{0:k} | \mathbf{U}_{0:k}, \mathbf{Z}_{0:k})$$

Modeling Actions and Sensing



- Action model: $P(x_t \mid x_{t-1}, u_{t-1})$
- Sensor model: $P(z_t \mid x_t)$
- Find Belief: $Bel(x_t) = P(x_t \mid u_1, z_2, \ldots, u_{t-1}, z_t)$

  the posterior probability distribution of x_t given

  the past history of actions and sensor inputs

Markov Localization

- Evaluate $Bel(x_t)$ for every possible state x_t

- **Prediction** step:

$$Bel^{-}(x_t) = \int P(x_t|u_{t-1}, x_{t-1}) Bel(x_{t-1}) dx_{t-1}$$

  - Integrate over every possible state x_{t-1} to apply the probability that action u_{t-1} could reach state x_t from x_{t-1}

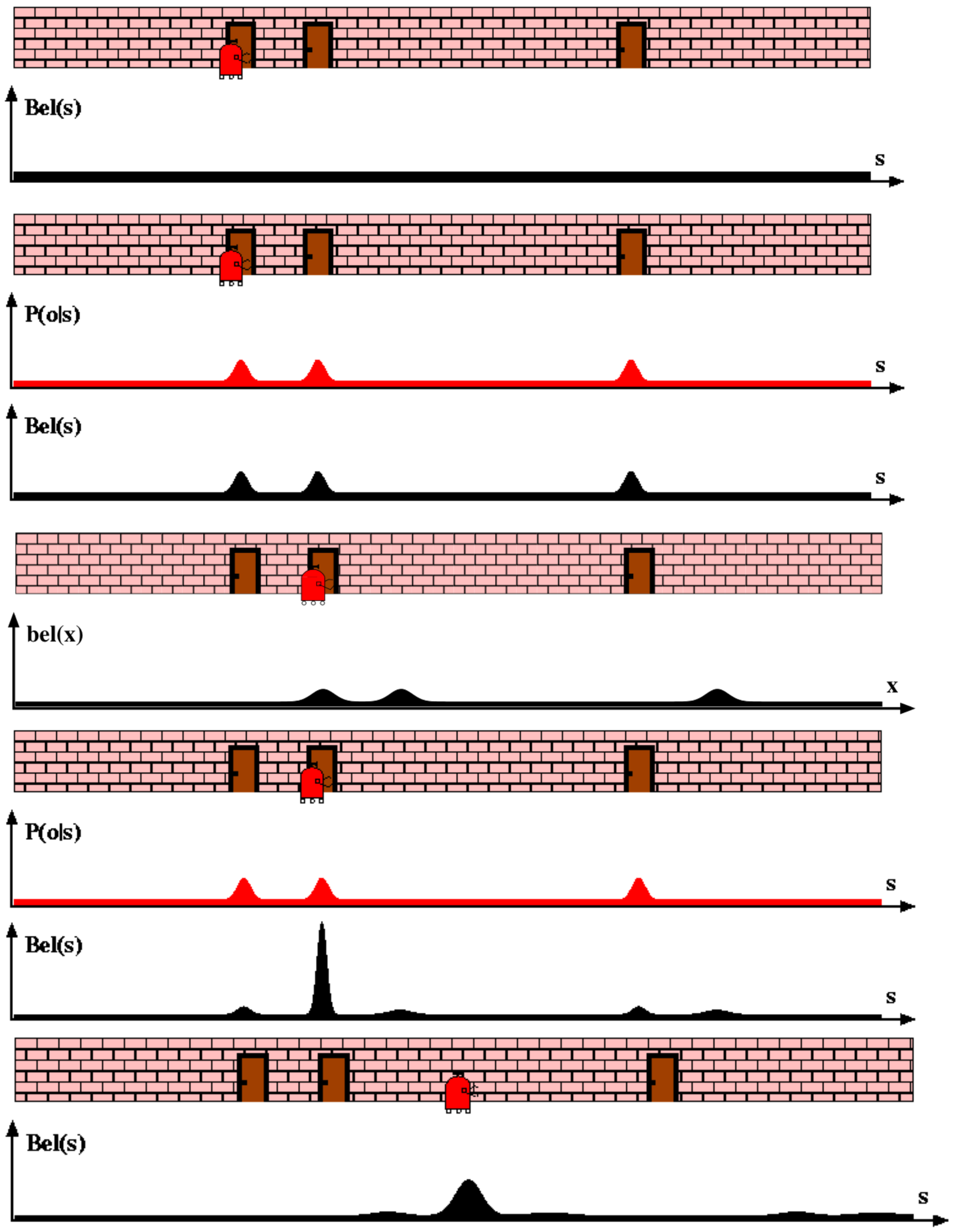
- **Correction** step:

$$Bel(x_t) = \eta\, P(z_t|x_t) Bel^{-}(x_t)$$

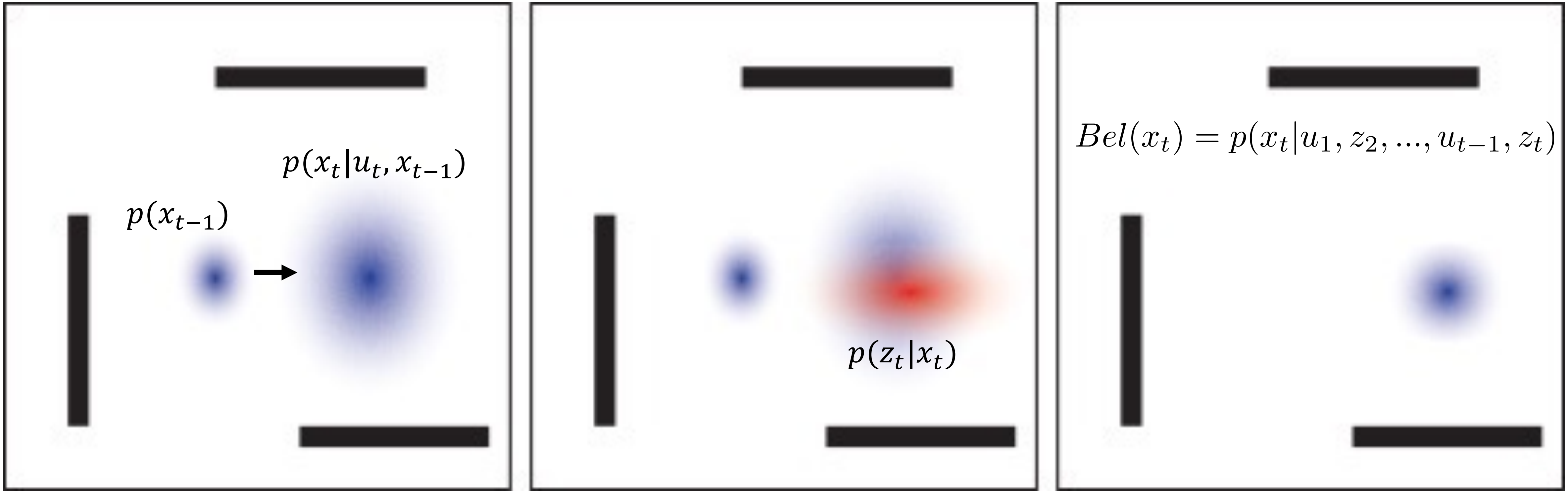
  - Weight each state x_t with likelihood of observation z_t

Localization Filter

- Prior
  - $Bel^{-}(x_0)$
- Sensor Model (update posterior)
  - $Bel(x_0) = \eta P(z_0|x_0) Bel^{-}(x_0)$
- Action Model (update new prior):
  - $Bel^{-}(x_1) = \int P(x_1|\, u_0, x_0)\, Bel\,(x_0) dx_0$
- Sensor Model
  - $Bel(x_1) = \eta P(z_1|x_1) Bel^{-}(x_1)$
- Action Model
  - $Bel^{-}(x_2) = \int P(x_2|\, u_1, x_1)\, Bel\,(x_1) dx_1$



Markov Localization



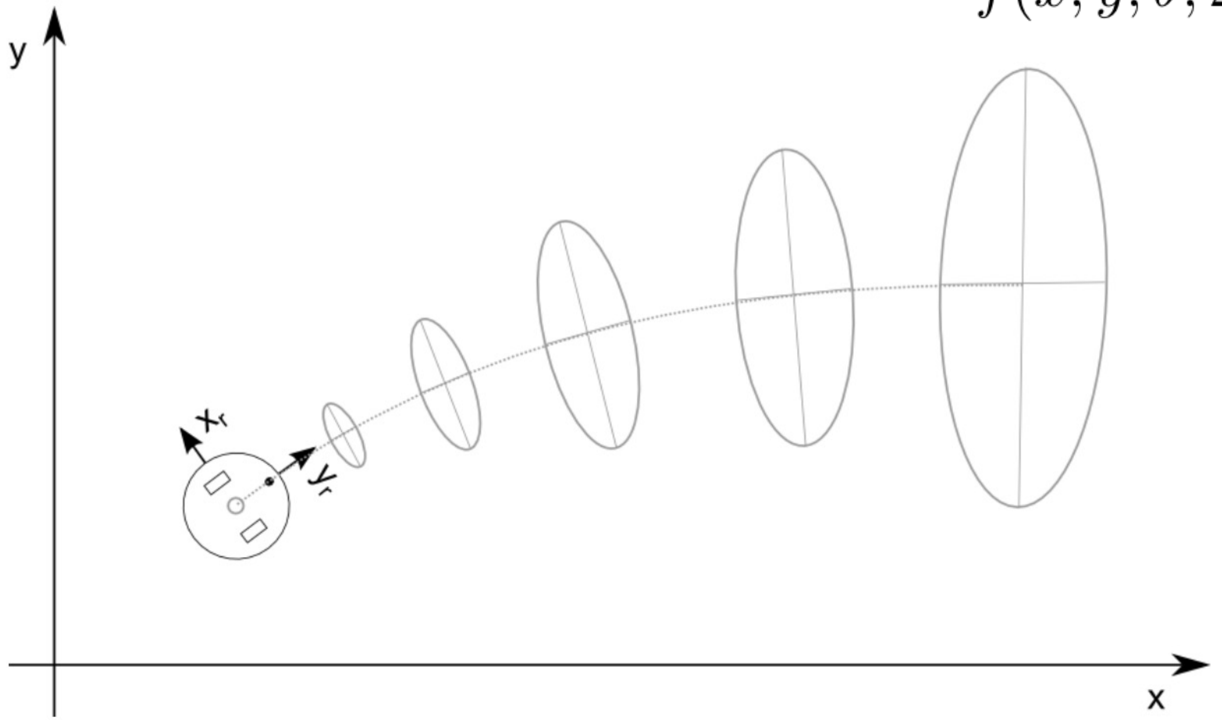
Prediction: apply action model

Sensor model

Correction: apply sensor model

Action Model - Error Propagation

$$f(x, y, \theta, \Delta s_r, \Delta s_l) = [x, y, \theta]^T + [\Delta x \qquad \Delta y \qquad \Delta\theta]^T$$



$$\Delta x = \Delta s cos(\theta + \Delta\theta/2)$$
$$\Delta y = \Delta s sin(\theta + \Delta\theta/2)$$
$$\Delta\theta = \frac{\Delta s_r - \Delta s_l}{b}$$

$$\Sigma_{p'} = \nabla_p f \Sigma_p \nabla_p f^T + \nabla_{\Delta_{r,l}} f \Sigma_\Delta \nabla_{\Delta_{r,l}} f^T$$

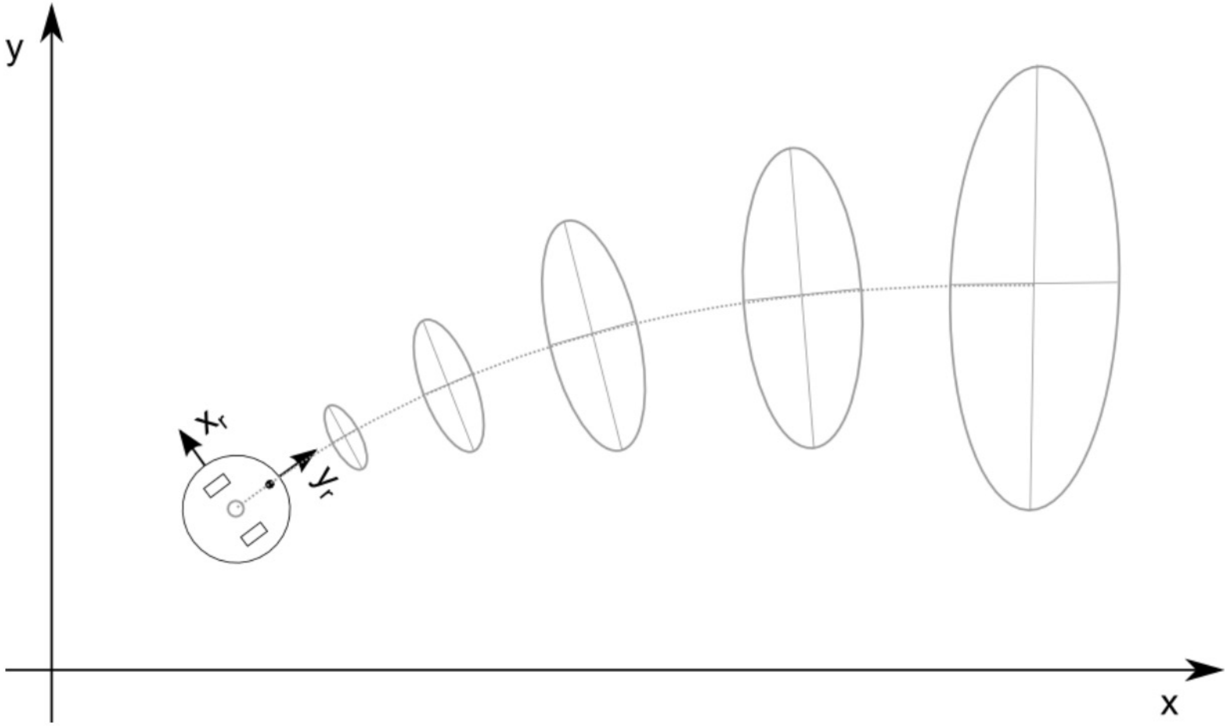
Error propagation law: Odometry example
Nikolaus Correll

https://www.youtube.com/watch?v=ubg_AAM7Zd8

Error Propagation

$$\Sigma_{p'} = \nabla_p f \Sigma_p \nabla_p f^T + \nabla_{\Delta_{r,l}} f \Sigma_\Delta \nabla_{\Delta_{r,l}} f^T$$

$$\nabla_{\Delta_{r,l}} f = \begin{bmatrix} \frac{1}{2}\cos(\theta + \frac{\Delta\theta/2}{b}) - \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) & \frac{1}{2}\cos(\theta + \frac{\Delta\theta/2}{b}) - \frac{\Delta s}{2b}\sin(\theta + \frac{\Delta\theta}{b}) \\ \frac{1}{2}\sin(\theta + \frac{\Delta\theta/2}{b}) + \frac{\Delta s}{2b}\cos(\theta + \frac{\Delta\theta}{b}) & \frac{1}{2}\sin(\theta + \frac{\Delta\theta/2}{b}) + \frac{\Delta s}{2b}\cos(\theta + \frac{\Delta\theta}{b}) \\ \frac{1}{2} & -\frac{1}{2} \end{bmatrix}$$



$$\Sigma_\Delta = \begin{bmatrix} k_r|\Delta s_r| & 0 \\ 0 & k_l|\Delta s_l| \end{bmatrix}$$

$$\nabla_p f = \begin{bmatrix} \frac{\partial f}{\partial x} & \frac{\partial f}{\partial y} & \frac{\partial f}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1\,0 & -\Delta s sin(\theta + \Delta\theta/2) \\ 0\,1 & \Delta s cos(\theta + \Delta\theta/2) \\ 0\,0 & 1 \end{bmatrix}$$

Representing a Particle Distribution

- The distribution $Bel(x_t)$ is represented by a set S_t of N weighted samples

$$S_t = \left\{ \left\langle x_t^{(i)},\ w_t^{(i)} \right\rangle | i = 1, \dots N \right\} \qquad x_t = \begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix}$$

Where

$$\sum_{i=1}^{N} w_t^{(i)} = 1$$

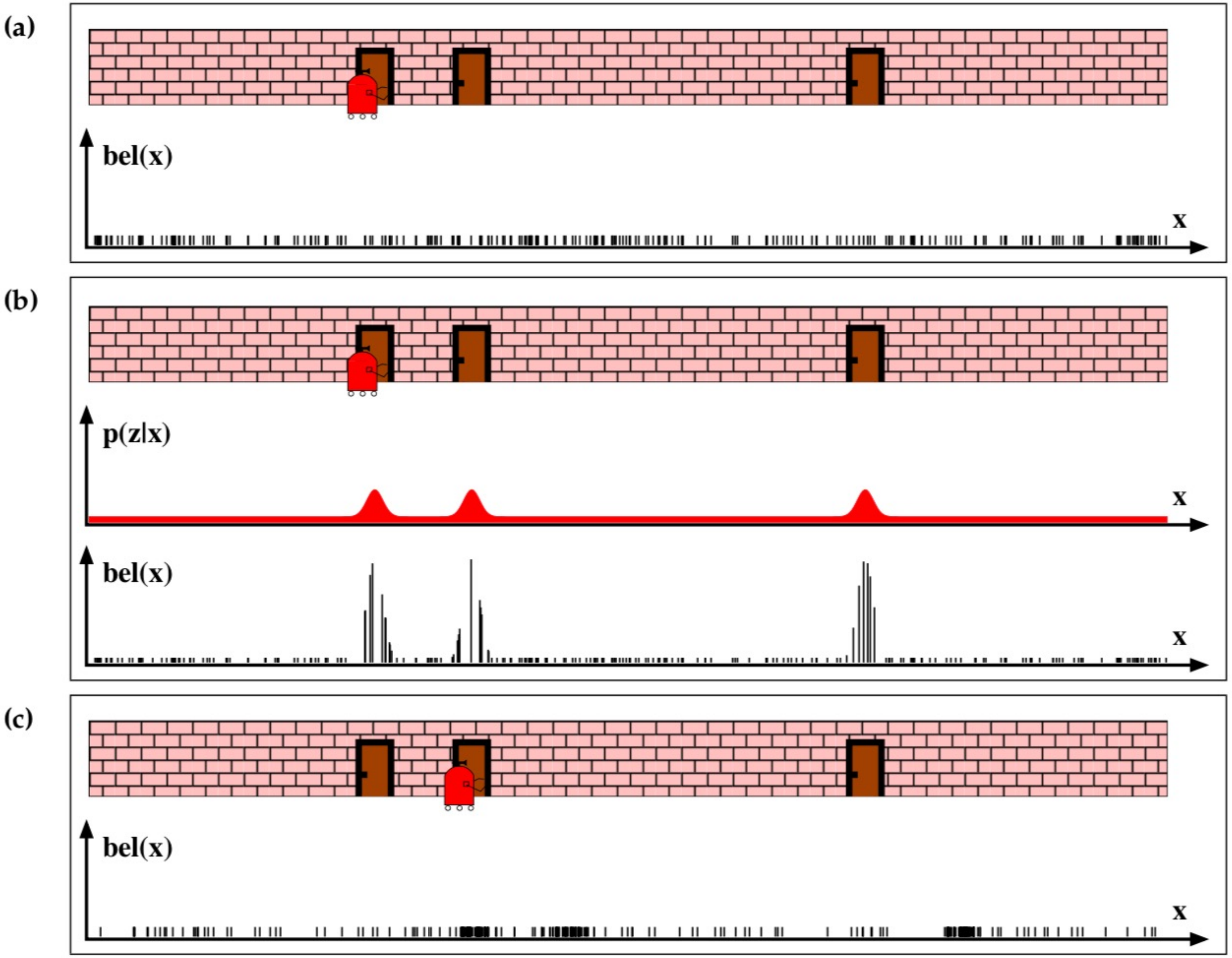
- A particle filter is a Bayes filter that uses this sample representation

Monte Carlo Localization with Particle Filter

- A pose is a position & orientation (x, y, θ)
- The probability distribution is represented by a collection of N poses
- Each pose has a weight (importance factor)
- Weights in the collection sum to 1
- Initialize with weight N^{-1}

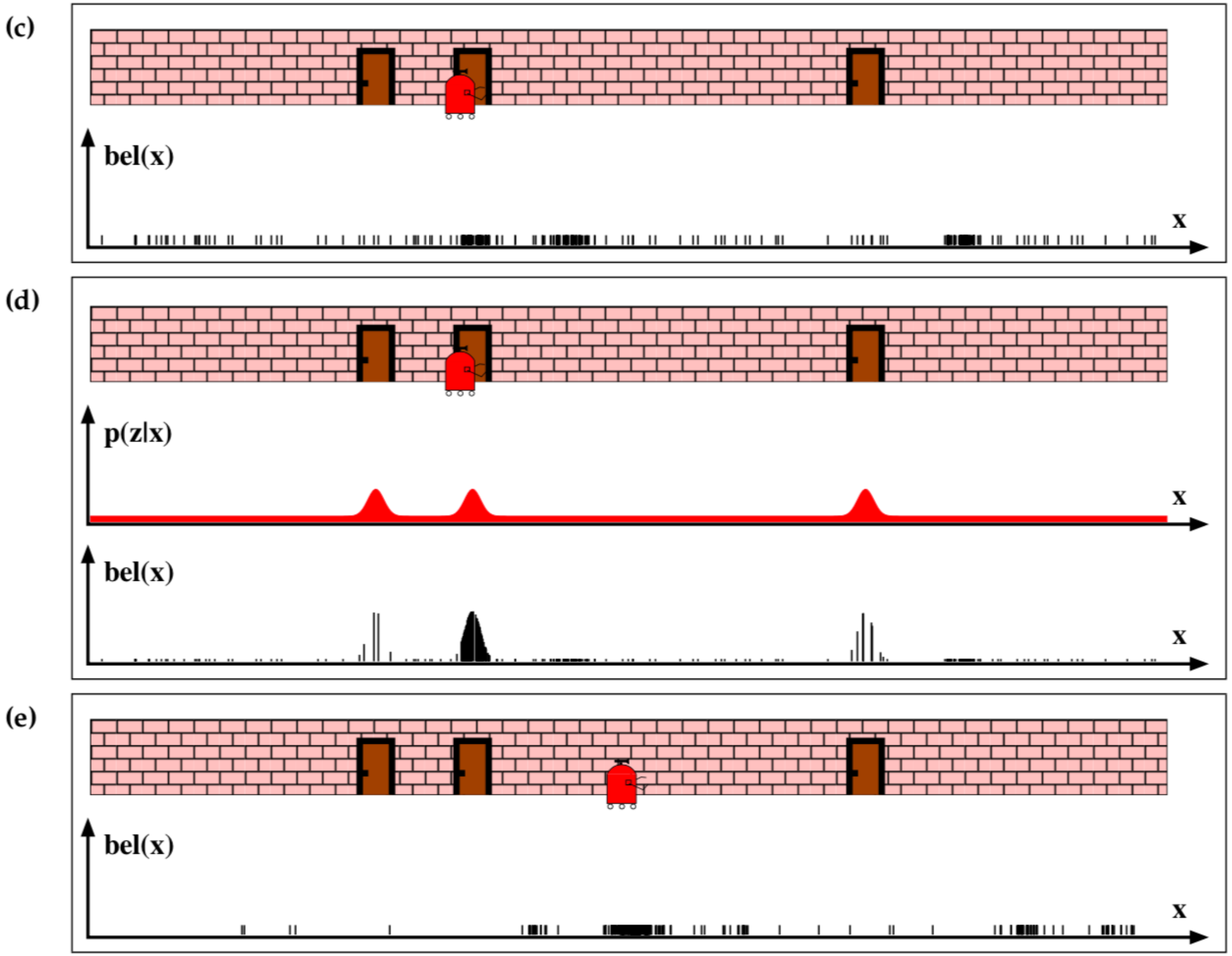
Localization with Particle Filter

- Begin with randomly sampled particles, equal weight
- Find weights of particles according to sensor model $p(z_t|x_t)$
- Resample particles according to posterior weights, apply action model $p(x_t|u_t, x_{t-1})$



Localization with Particle Filter

- Previous posterior is new prior
- Find weights of particles according to sensor model $p(z_t|x_t)$
- Resample particles according to posterior weights, apply action model $p(x_t|u_t, x_{t-1})$



Particle Filter Algorithm

Input: $u_{t-1}, z_t, S_{t-1} = \left\{\left\langle x_{t-1}^{(i)},\ w_{t-1}^{(i)}\right\rangle | i = 1, \dots, N\right\}$

Initialize: $S_t \coloneqq \emptyset, i \coloneqq 1, \ \ \alpha \coloneqq 0$

while $i \leq N$ **do**

sample j from the discrete distribution given by weights in S_{t-1}

sample $x_t^{(i)}$ from $p(x_t | u_{t-1}, x_{t-1})$ given $x_{t-1}^{(j)}$ and u_{t-1}

$w_t^{(i)} \coloneqq \ p(z_t | x_t^{(i)})$

$\alpha = \alpha + w_t^{(i)}, i \coloneqq i + 1$

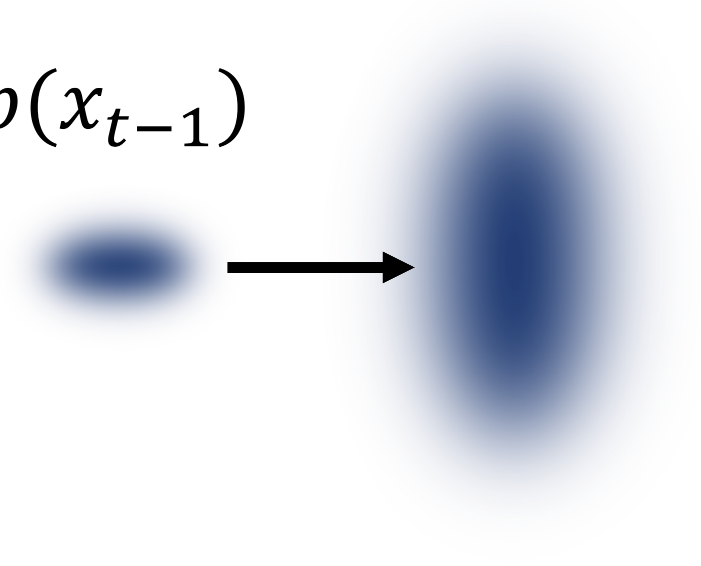
$S_t = S_t \cup \left\{\left\langle x_t^{(i)},\ w_t^{(i)}\right\rangle\right\}$

for $i \coloneqq 1$ to N **do** $w_t^{(i)} \coloneqq w_t^{(i)} / \alpha$

return S_t

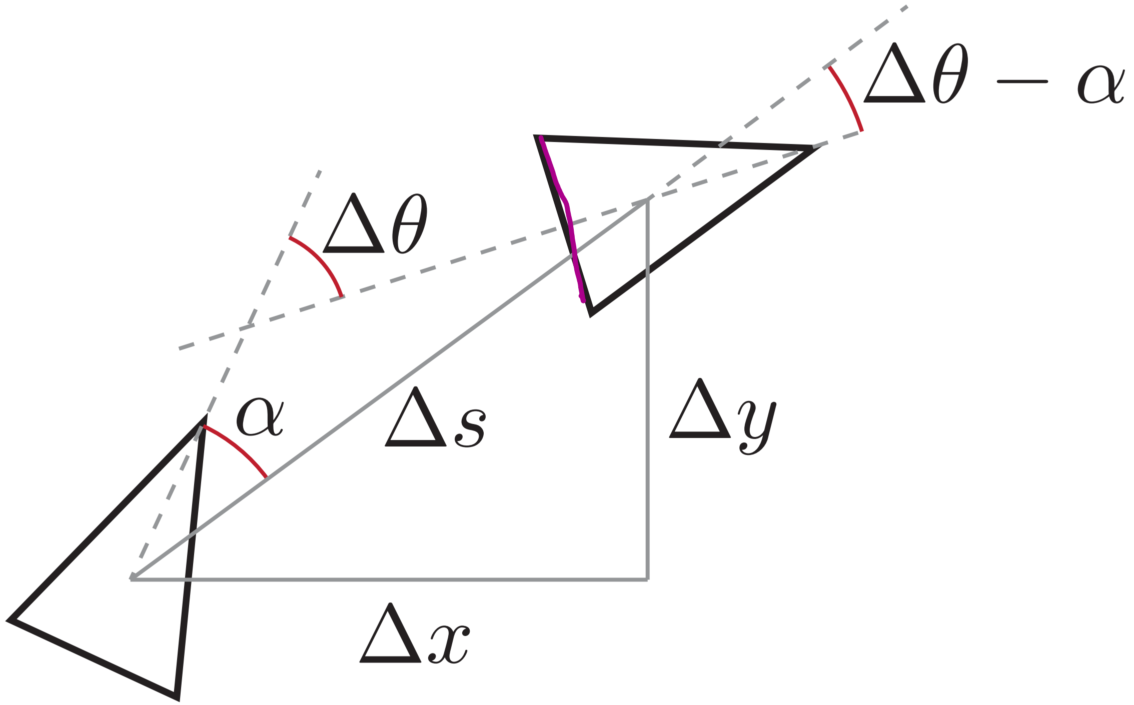
Action Model

- How certain are we of our state after a given action?
- Action disperses the probability distribution



Modeling Action

- From odometry we have previous and current pose
  - $(x_{t-1}, y_{t-1}, \theta_{t-1})$
  - (x_t, y_t, θ_t)
  - $(\Delta x, \Delta \mathrm{y}, \Delta\theta)$
- $\Delta s^2 = \Delta x^2 + \Delta y^2$
- $\alpha = \mathrm{atan2}(\Delta y, \Delta x) - \theta_{t-1}$

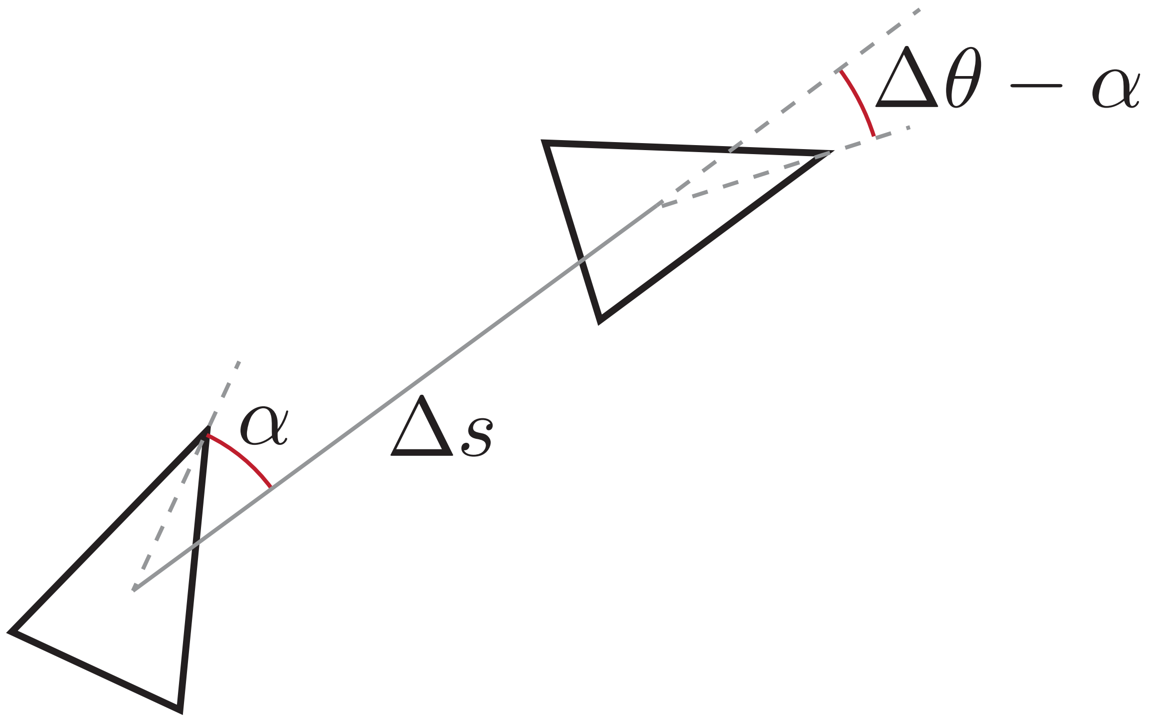


- Model action as a rotation, translation and rotation:

$$u = \begin{bmatrix} \alpha & \Delta s & \Delta\theta - \alpha \end{bmatrix}$$

Modeling Action Error

- Model the action as
  - Turn($\alpha + \varepsilon_1$)
  - Travel($\Delta s + \varepsilon_2$)
  - Turn($\Delta\theta - \alpha + \varepsilon_3$)
- Model errors as Gaussian:
  - $\varepsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$
  - $\varepsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|)$
  - $\varepsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$
- Standard Deviation proportional to action magnitude



The Action Model $P(x_t|u_{t-1}, x_{t-1})$

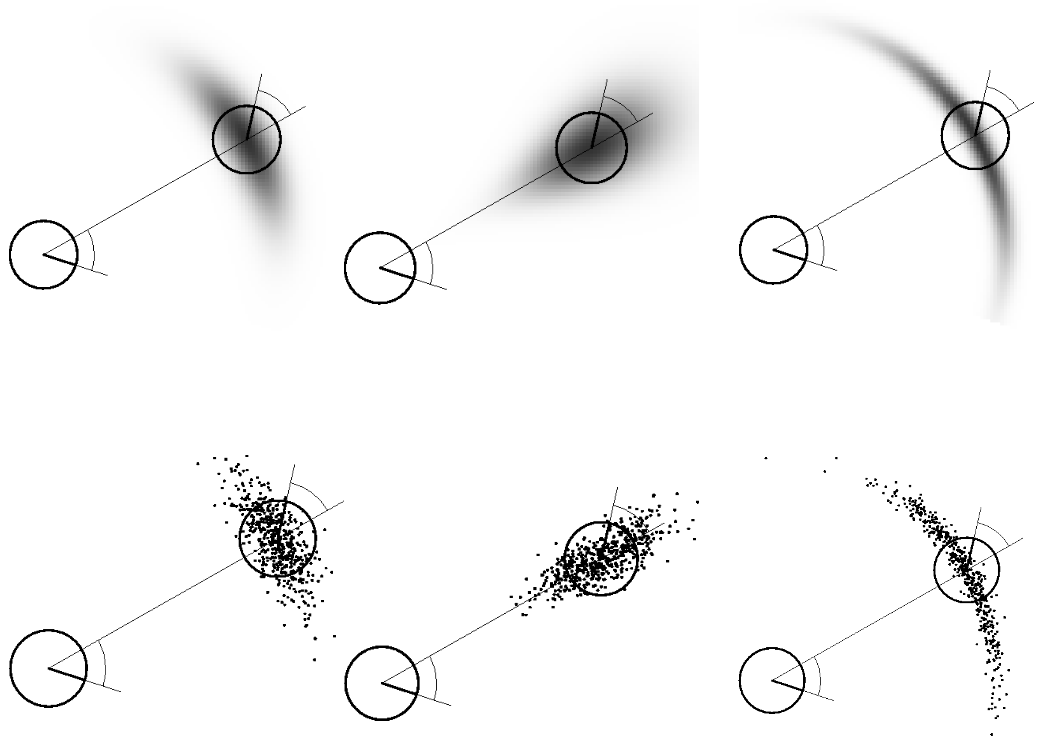
- Given action: $u = \begin{bmatrix}\alpha & \Delta s & \Delta\theta - \alpha\end{bmatrix}$

$$\begin{bmatrix} x_t \\ y_t \\ \theta_t \end{bmatrix} = \begin{bmatrix} x_{t-1} \\ y_{t-1} \\ \theta_{t-1} \end{bmatrix} + \begin{bmatrix} (\Delta s + \varepsilon_2)\cos(\theta_{t-1} + \alpha + \varepsilon_1) \\ \Delta s + \varepsilon_2)\sin(\theta_{t-1} + \alpha + \varepsilon_1) \\ \Delta\theta + \varepsilon_1 + \varepsilon_3 \end{bmatrix}$$

- Where:
  - $\varepsilon_1 \sim \mathcal{N}(0, k_1|\alpha|)$
  - $\varepsilon_2 \sim \mathcal{N}(0, k_2|\Delta s|)$
  - $\varepsilon_3 \sim \mathcal{N}(0, k_1|\Delta\theta - \alpha|)$

Actions Disperse the Distribution

- N particles is approximately a probability distribution
- The distribution disperses with action
- When tuning, look at your distribution and see if dispersion is reasonable given your experience with robot



Tuning the Action Model

- The noise in the action model captures non-systematic errors
- Calibrate odometry first to eliminate systematic errors.
- Can perform straight line experiments and rotation experiments to determine reasonable values of k_1 and k_2
- If dispersion is too small or too large, localization will fail.

Sample Odometry Action Model

Algorithm sample_motion_model_odometry(u_t, x_{t-1})**:**

$$\delta_{\mathrm{rot1}} = \mathrm{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$$

$$\delta_{\mathrm{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$$

$$\delta_{\mathrm{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\mathrm{rot1}}$$

$$\hat{\delta}_{\mathrm{rot1}} = \delta_{\mathrm{rot1}} - \mathbf{sample}(\alpha_1 \delta_{\mathrm{rot1}}^2 + \alpha_2 \delta_{\mathrm{trans}}^2)$$

$$\hat{\delta}_{\mathrm{trans}} = \delta_{\mathrm{trans}} - \mathbf{sample}(\alpha_3 \delta_{\mathrm{trans}}^2 + \alpha_4\ \delta_{\mathrm{rot1}}^2 + \alpha_4\ \delta_{\mathrm{rot2}}^2)$$

$$\hat{\delta}_{\mathrm{rot2}} = \delta_{\mathrm{rot2}} - \mathbf{sample}(\alpha_1 \delta_{\mathrm{rot2}}^2 + \alpha_2 \delta_{\mathrm{trans}}^2)$$

$$x' = x + \hat{\delta}_{\mathrm{trans}}\ \cos(\theta + \hat{\delta}_{\mathrm{rot1}})$$

$$y' = y + \hat{\delta}_{\mathrm{trans}}\ \sin(\theta + \hat{\delta}_{\mathrm{rot1}})$$

$$\theta' = \theta + \hat{\delta}_{\mathrm{rot1}} + \hat{\delta}_{\mathrm{rot2}}$$

return $x_t = (x', y', \theta')^T$

- Book uses 4 error coefficients α_1 to α_4
- α_1 and α_3 are same as k_1 and k_2 in our model
- α_2 and α_4 represent cross correlation
- You can use either model

Action Model Implementation

- Implemented in `action_model.cpp`
- Use `updateAction()` to check motion, calculate deltas and calculate standard deviations
- Use `applyAction()` to sample from normal distributions with previously calculated stdev and apply them to a particle.
- Lookup `std::normal_distribution` for how to sample from a normal distribution

Sampling from a Distribution

```
#include <random>
int main(){
    // initializing random seed
    std::random_device rd;
    // Mersenne twister PRNG, initialized with seed random device instance
    std::mt19937 gen(rd());
    int i;
    float sample;
    float mean = 0.0
    float stddev = 1.0
    // instance of class std::normal_distribution with specific mean and stddev
    std::normal_distribution<float> d(mean, stddev);
    for(i = 0; i < 1000; ++i)
    {
        // get random number with normal distribution using gen as random source
        sample = d(gen);
        do_something_with_this_value(sample);
    }
    return 0;
}
```