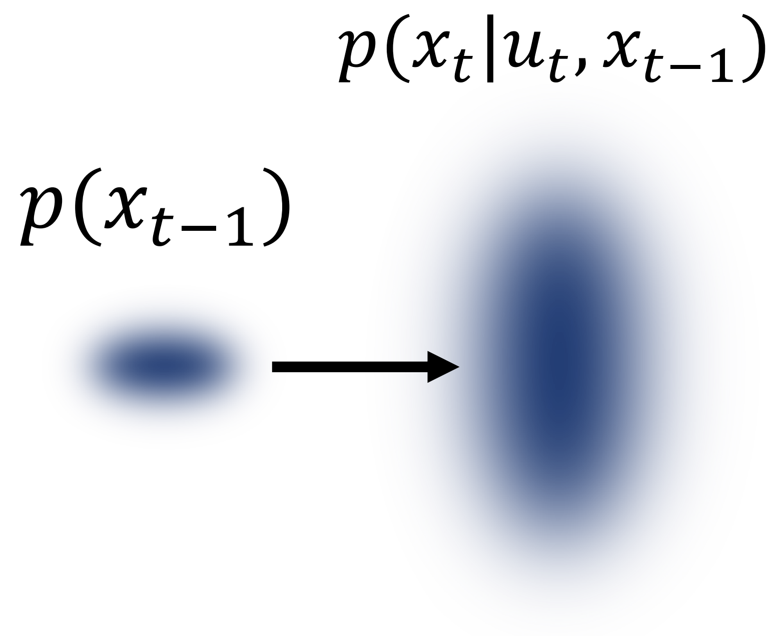
# Action Model

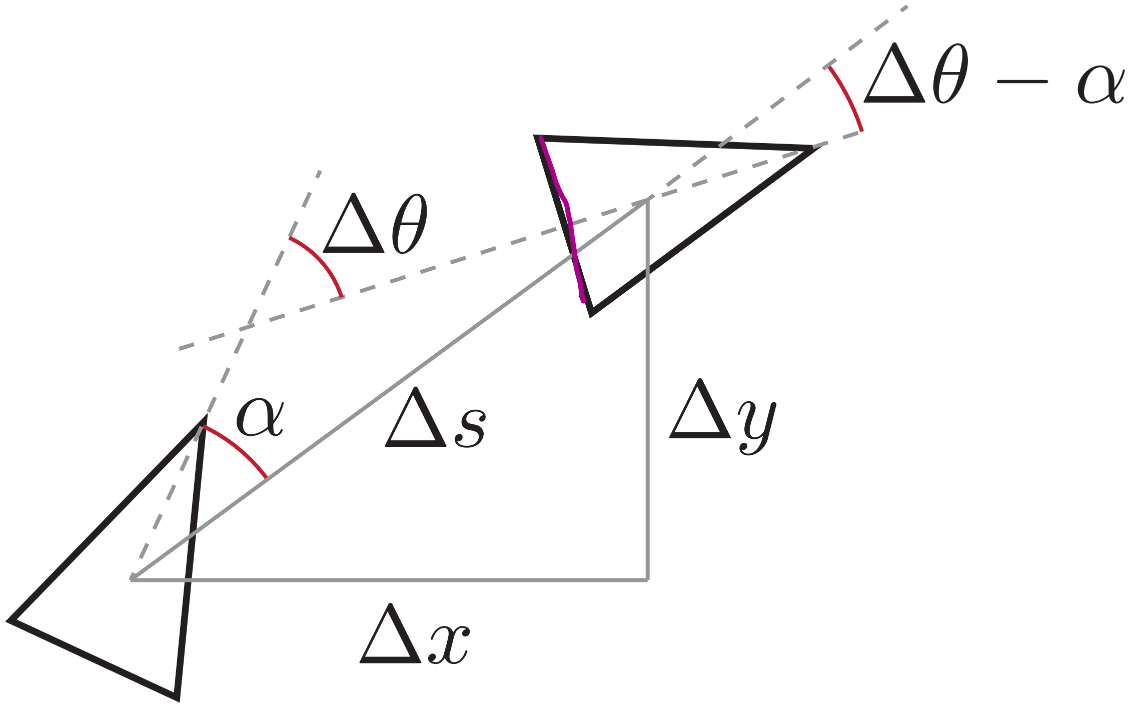
# Action Model

- 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
  - $(x_{t-1}, y_{t-1}, \theta_{t-1})$
  - $(x_t, y_t, \theta_t)$
  - $(\Delta x, \Delta \mathrm{y}, \Delta\theta)$
- $\Delta s^2 = \Delta x^2 + \Delta y^2$
- $\alpha = \mathrm{atan2}(\Delta y, \Delta x) - \theta_{t-1}$

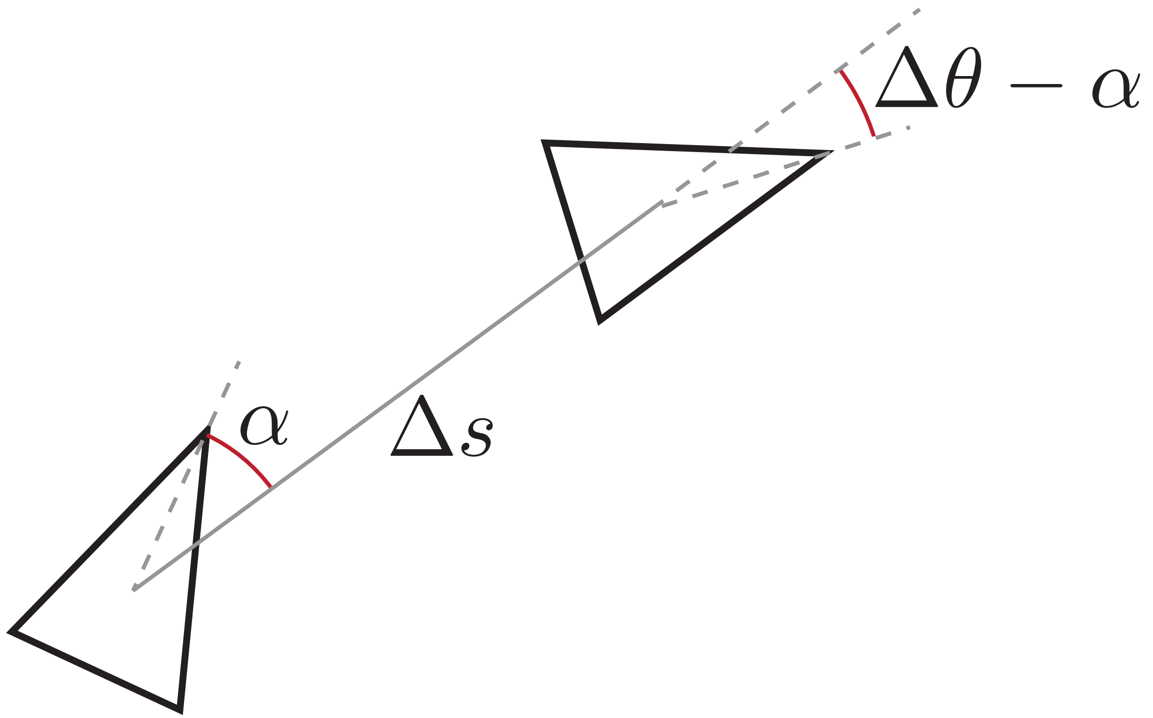



- Model action as a rotation, translation and rotation:

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

# 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 $P(x_t|u_{t-1}, x_{t-1})$

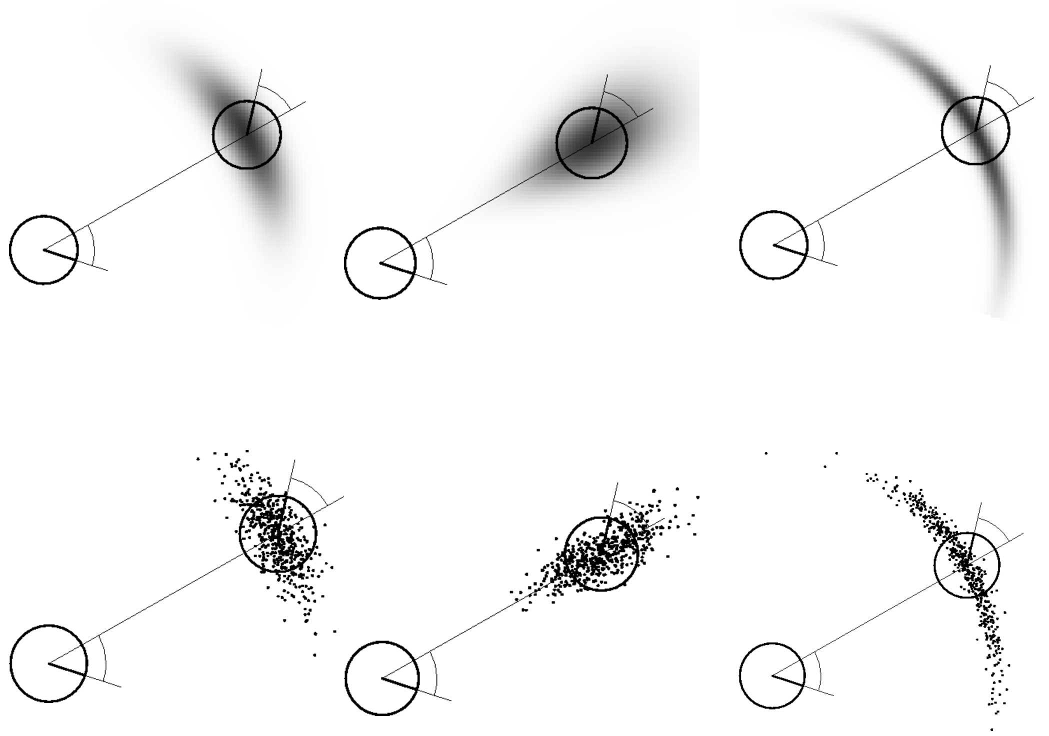
- Given action: $u = \begin{bmatrix}\alpha & \Delta s & \Delta\theta - \alpha\end{bmatrix}$

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

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

$\delta_{\text{rot1}} = \text{atan2}(\bar{y}' - \bar{y}, \bar{x}' - \bar{x}) - \bar{\theta}$
$\delta_{\text{trans}} = \sqrt{(\bar{x} - \bar{x}')^2 + (\bar{y} - \bar{y}')^2}$
$\delta_{\text{rot2}} = \bar{\theta}' - \bar{\theta} - \delta_{\text{rot1}}$

$\hat{\delta}_{\text{rot1}} = \delta_{\text{rot1}} - \textbf{sample}(\alpha_1 \delta_{\text{rot1}}^2 + \alpha_2 \delta_{\text{trans}}^2)$
$\hat{\delta}_{\text{trans}} = \delta_{\text{trans}} - \textbf{sample}(\alpha_3 \delta_{\text{trans}}^2 + \alpha_4\ \delta_{\text{rot1}}^2 + \alpha_4\ \delta_{\text{rot2}}^2)$
$\hat{\delta}_{\text{rot2}} = \delta_{\text{rot2}} - \textbf{sample}(\alpha_1 \delta_{\text{rot2}}^2 + \alpha_2 \delta_{\text{trans}}^2)$

$x' = x + \hat{\delta}_{\text{trans}}\ \cos(\theta + \hat{\delta}_{\text{rot1}})$
$y' = y + \hat{\delta}_{\text{trans}}\ \sin(\theta + \hat{\delta}_{\text{rot1}})$
$\theta' = \theta + \hat{\delta}_{\text{rot1}} + \hat{\delta}_{\text{rot2}}$

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

- Book uses 4 error coefficients $\alpha_1$ to $\alpha_4$
- $\alpha_1$ and $\alpha_3$ are same as $k_1$ and $k_2$ in our model
- $\alpha_2$ and $\alpha_4$ represent cross correlation
- You can use either model

# Action Model Implementation

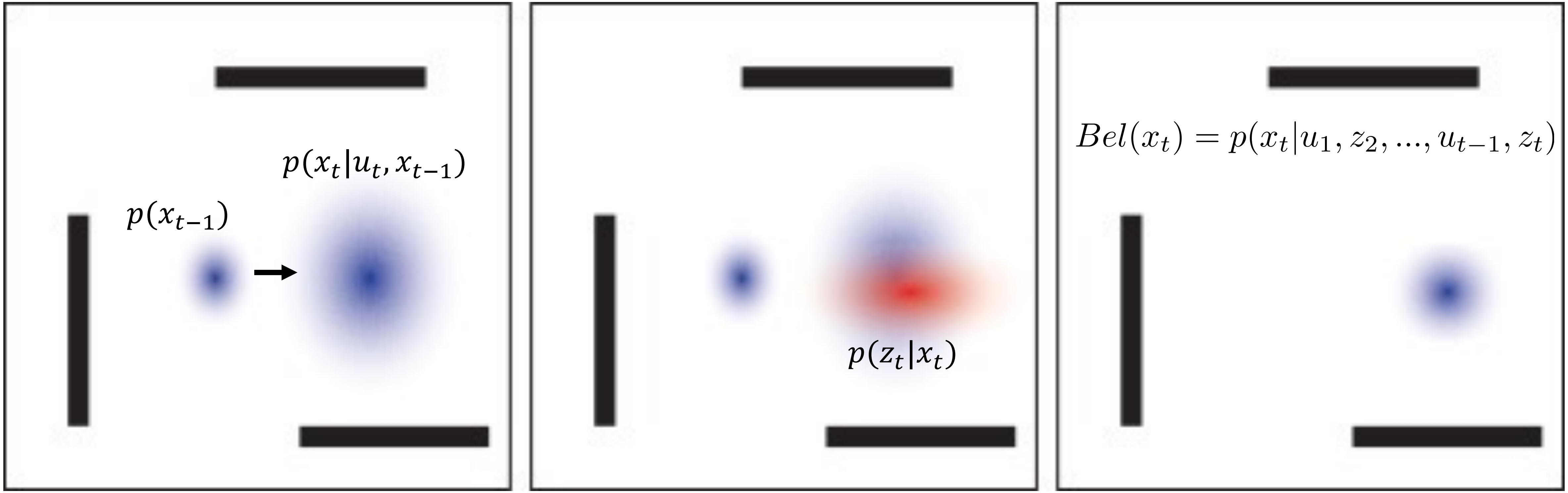
- Implemented in `action_model.cpp`
- Use `updateAction()` to check motion, calculate deltas and calculate standard deviations
- Use `applyAction()` to sample from normal distributions with previously calculated stdev and apply them to a particle.
- Lookup `std::normal_distribution` for how to sample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

# Sensor Model

# Markov Localization




Prediction: apply action model

Sensor model

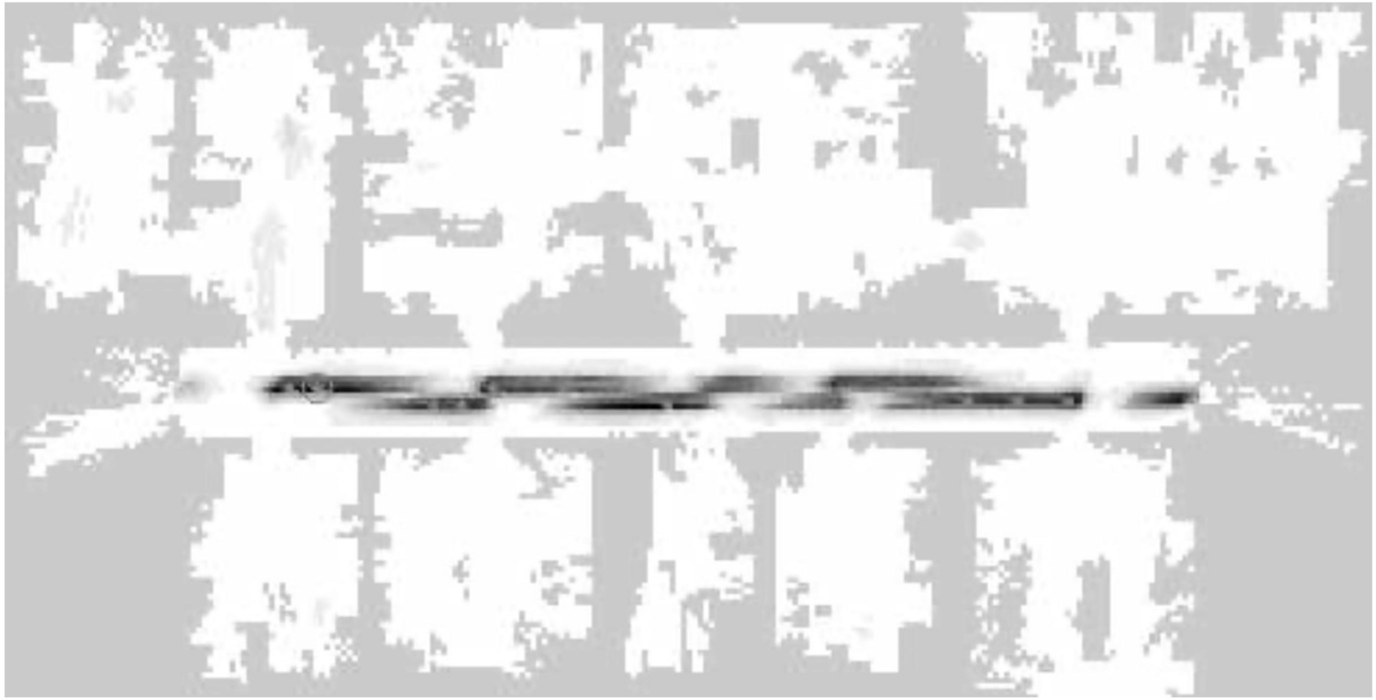
Correction: apply sensor model

# Sensor Model $p(z_t|x_t)$

- Purpose: find the probability that a lidar scan matches the hypothesis pose (particle) given a map
- Give a score to each ray in the scan, sum for scan score
- Normalize score across particles to get particle weight
- Resample particle distribution based on weights before applying action model again.

# Sensor Model

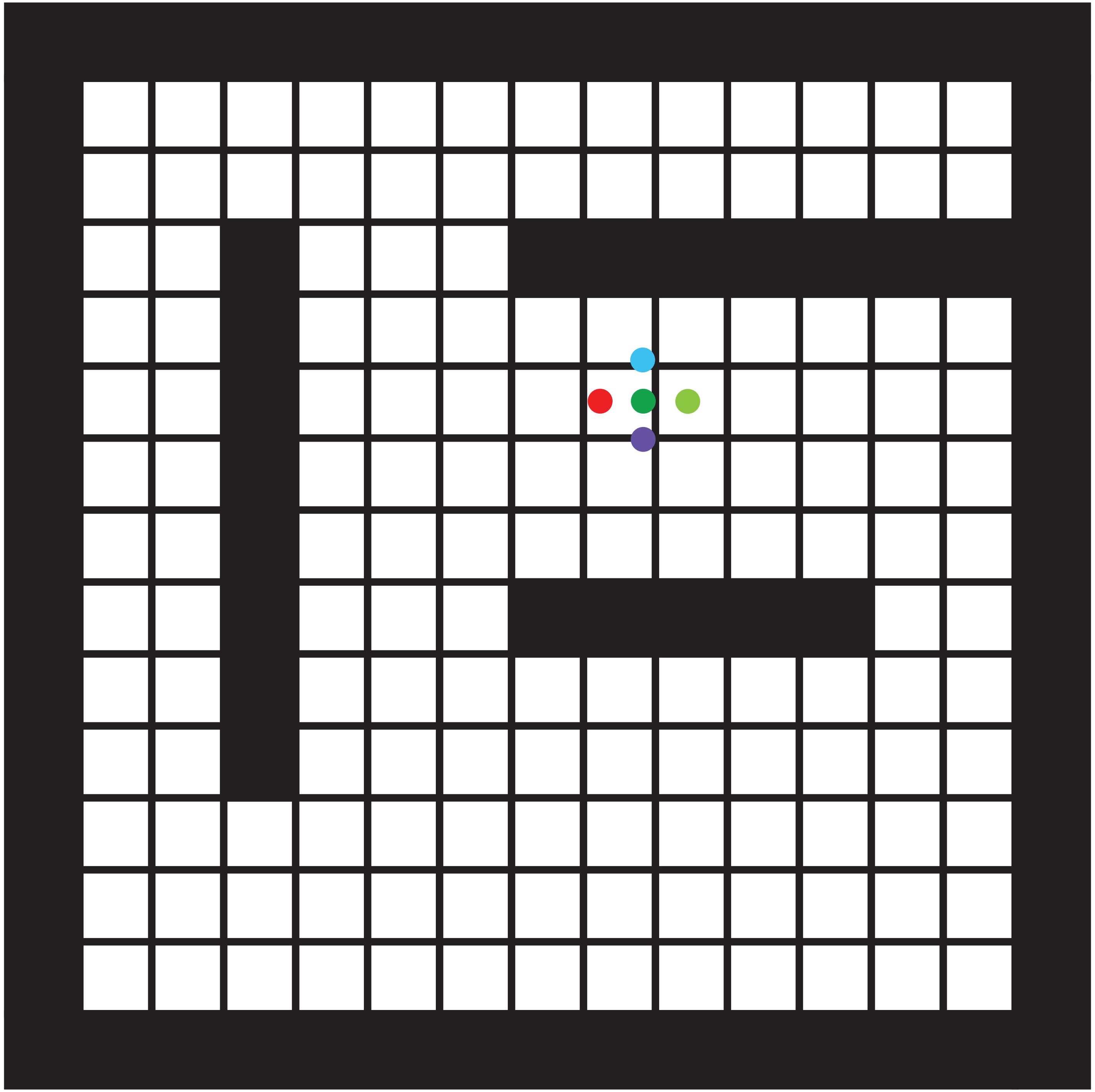
$z_t, m$

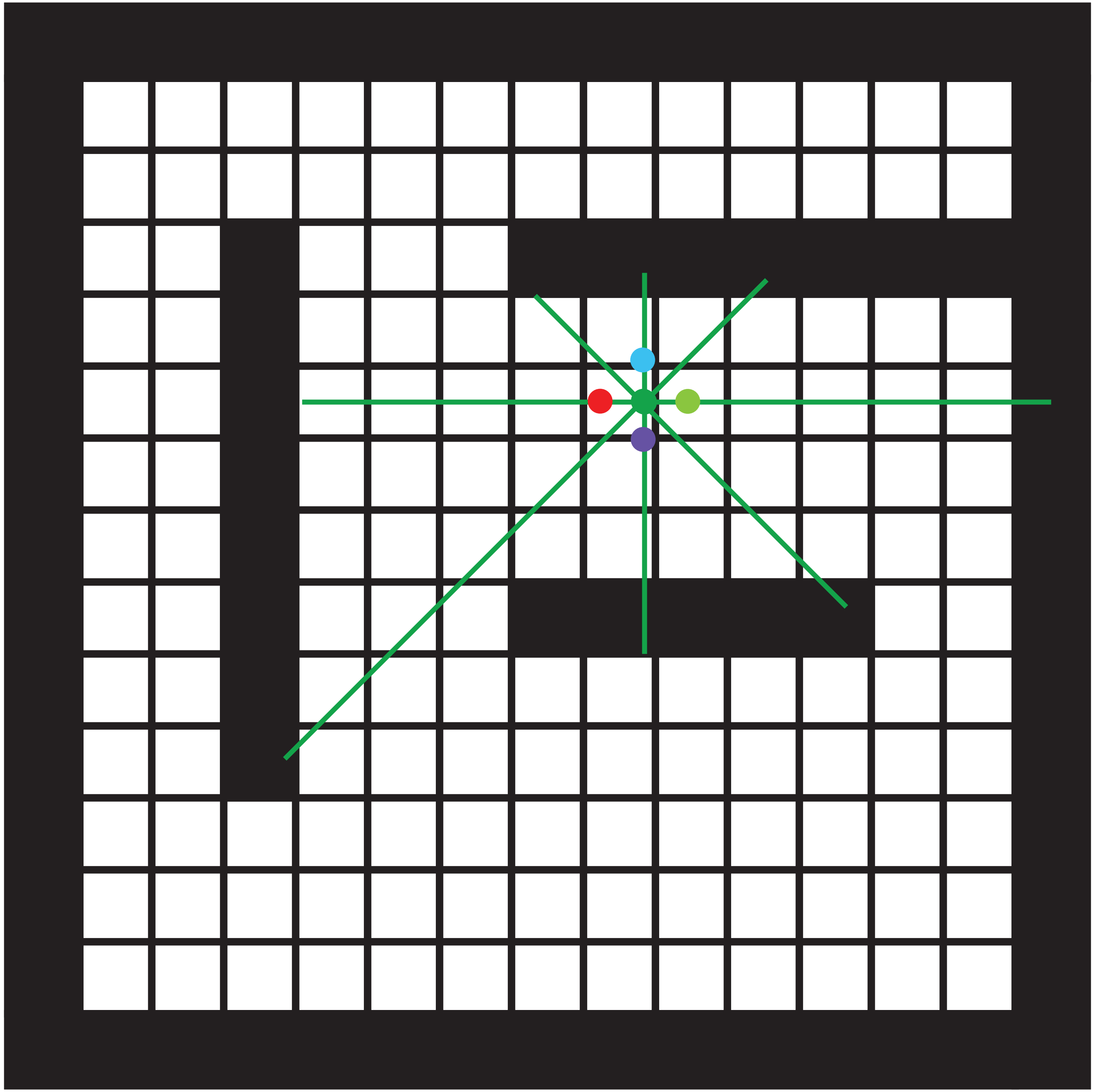


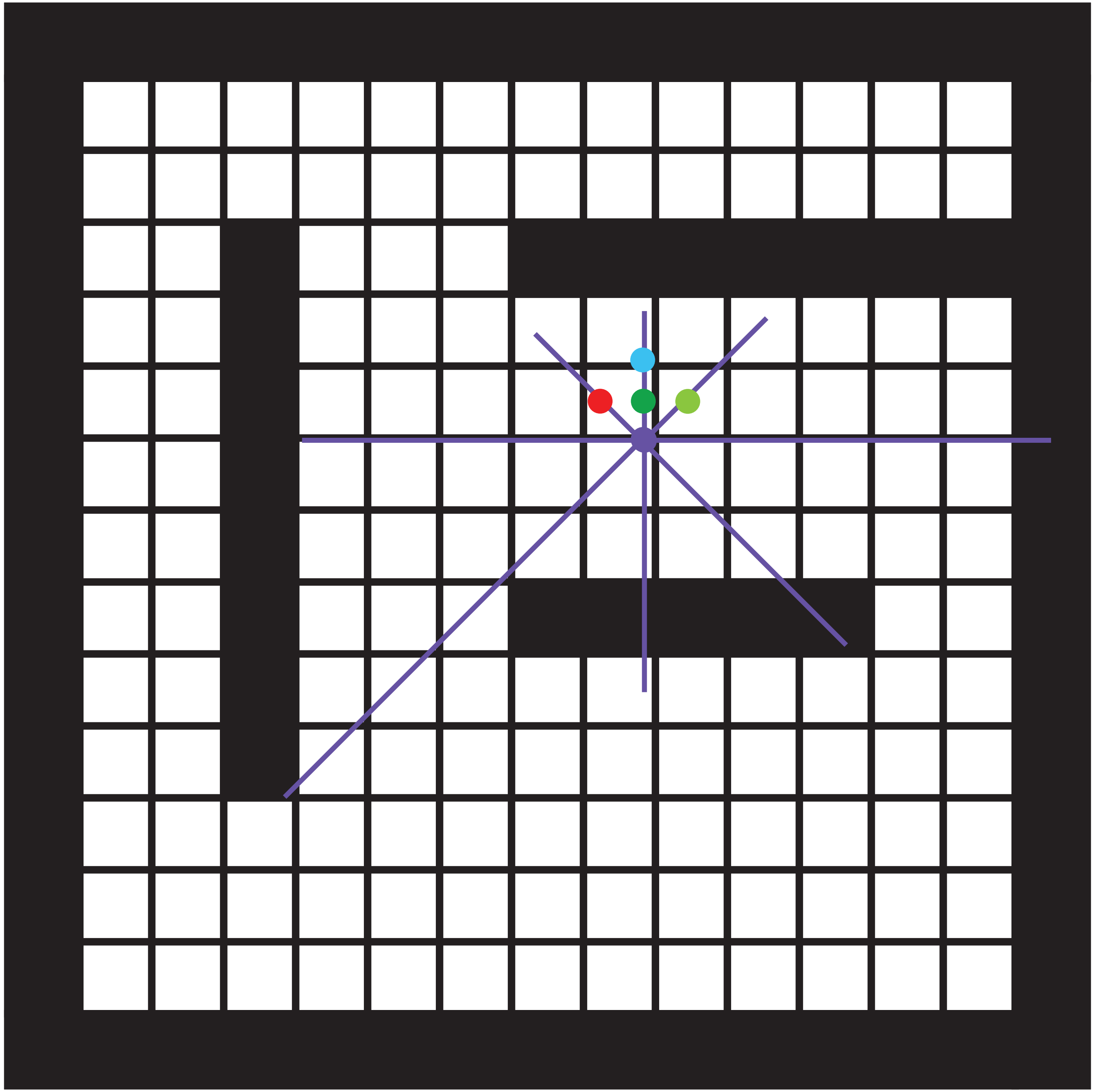
$p(z_t|x_t, m)$

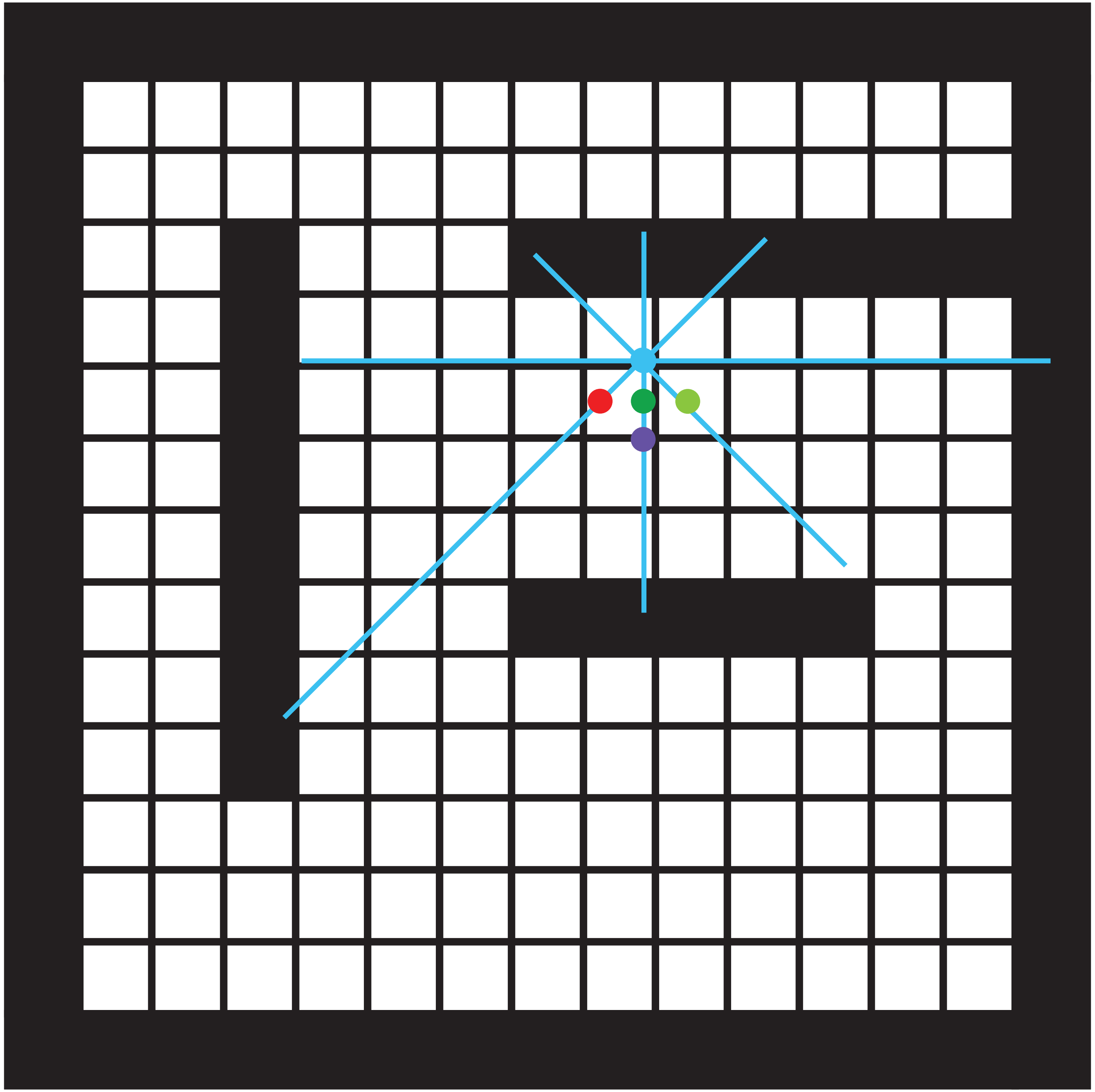
# Sensor Model

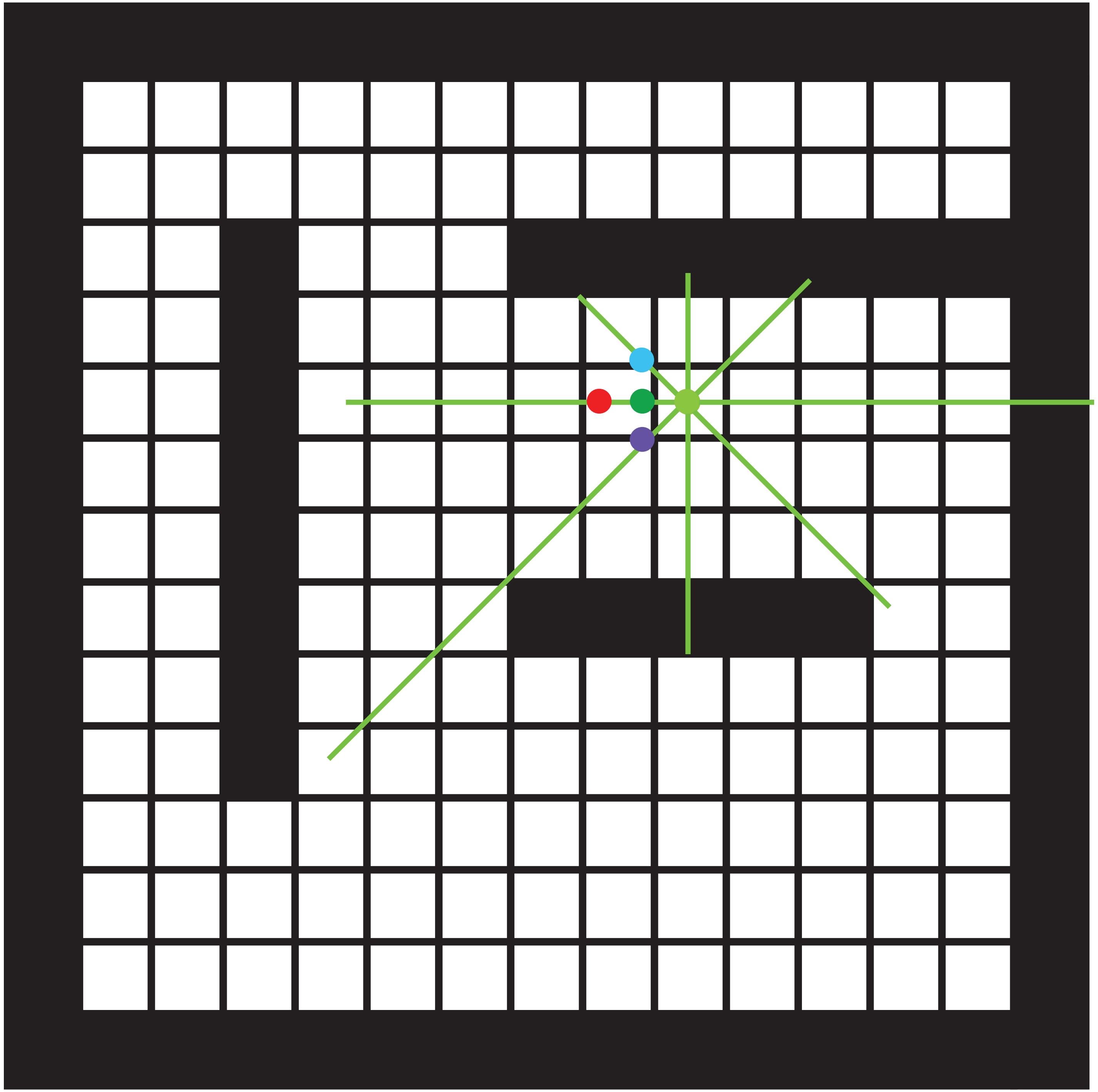
- Instead of calculating $p(z_t|x_t, m)$ for every possible pose in the map, only need to evaluate for each particle.
- Particle effectively represents hypothesis robot took the path from previous particle pose to new pose given by action model
- Sensor model will then test each ray in the laser scan and provide a fitness score based on the particles hypothesis
- Fitness scores normalized over all particles become particle weights
- You will decide how to assign fitness score based on accuracy/computational considerations

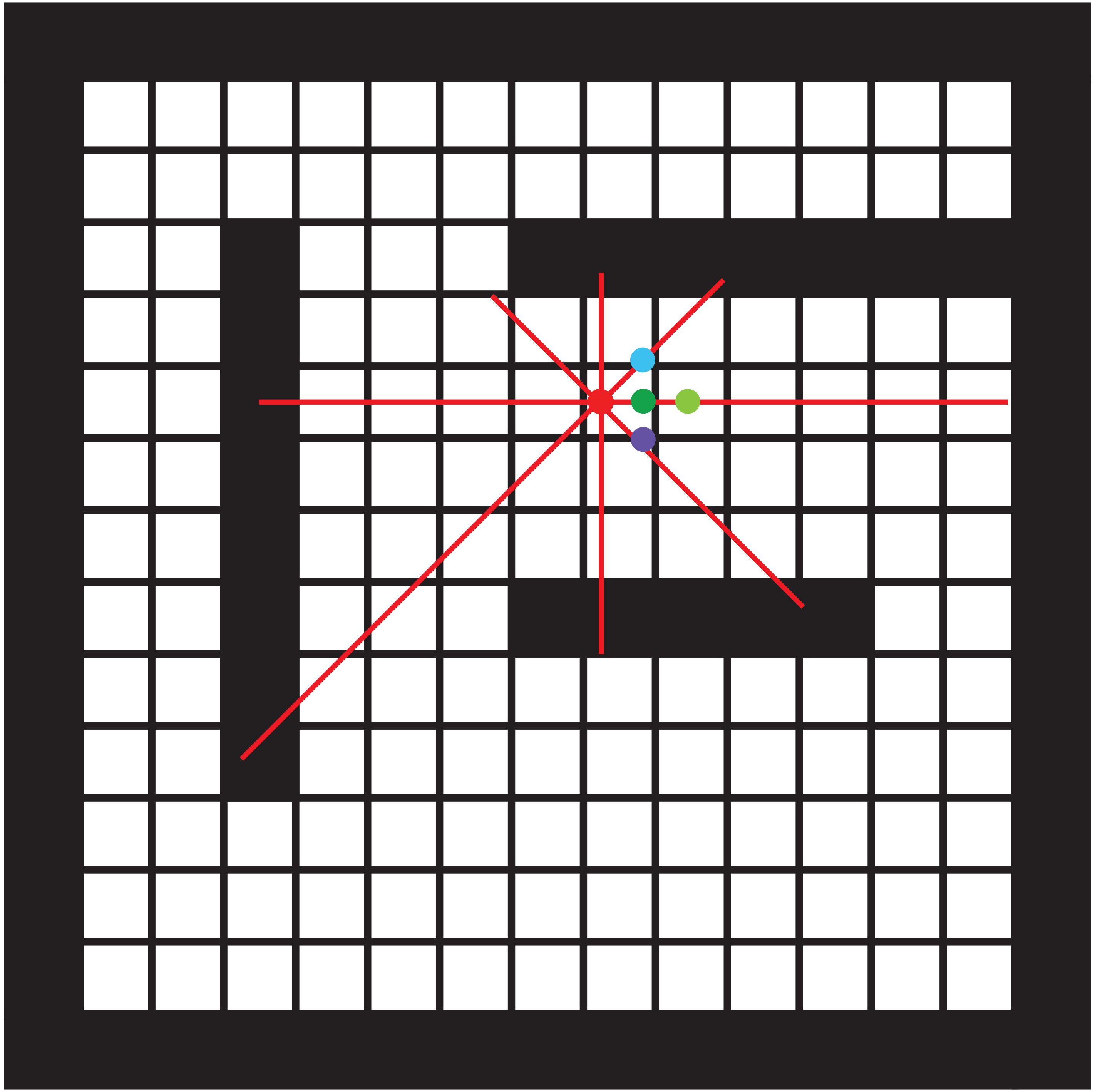






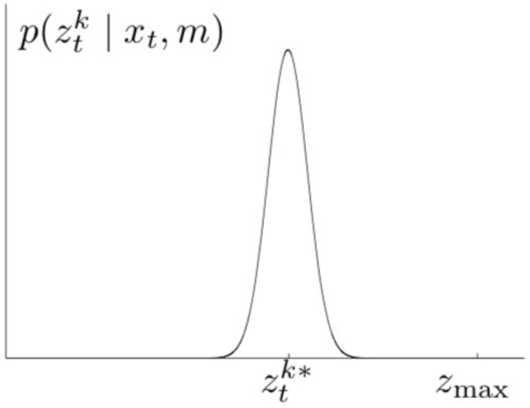




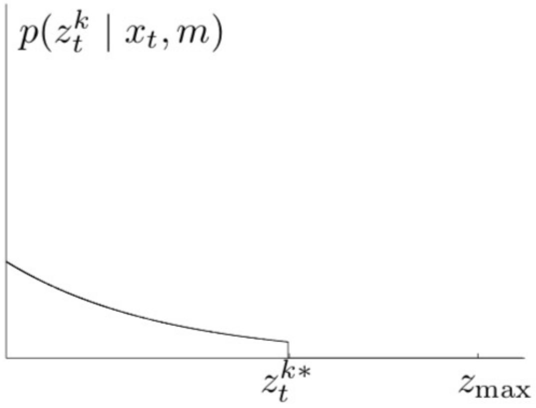


# Sensor Model: Beam Model

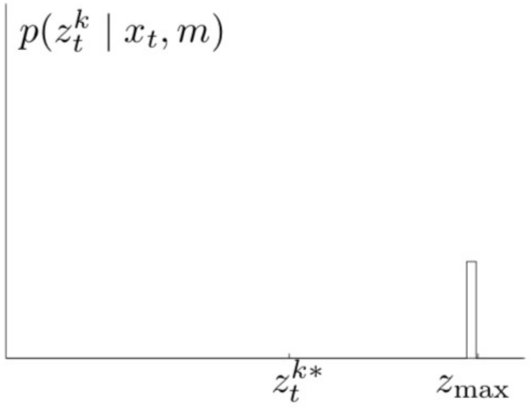
**(a)** Gaussian distribution $p_{\mathrm{hit}}$



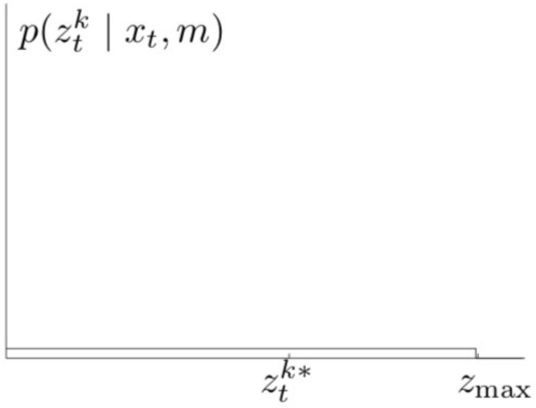

**(b)** Exponential distribution $p_{\mathrm{short}}$



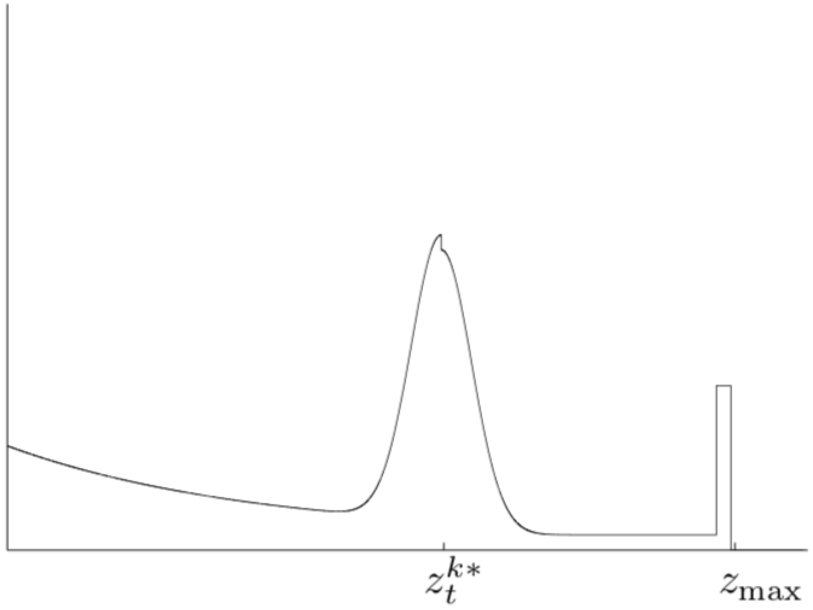

**(c)** Uniform distribution $p_{\mathrm{max}}$




**(d)** Uniform distribution $p_{\mathrm{rand}}$




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



# Sensor Model: Beam Model

- The beam model looks at each ray endpoint casted on the grid and calculates the product of probabilities of the rays in the scan

**Algorithm beam_range_finder_model(**$z_t, x_t, m$**):**

$q = 1$

*for* $k = 1$ *to* $K$ *do*

*compute* $z_t^{k*}$ *for the measurement* $z_t^k$ *using ray casting*

$p = z_{\text{hit}} \cdot p_{\text{hit}}(z_t^k \mid x_t, m) + z_{\text{short}} \cdot p_{\text{short}}(z_t^k \mid x_t, m)$

$+ z_{\max} \cdot p_{\max}(z_t^k \mid x_t, m) + z_{\text{rand}} \cdot p_{\text{rand}}(z_t^k \mid x_t, m)$

$q = q \cdot p$

*return* $q$

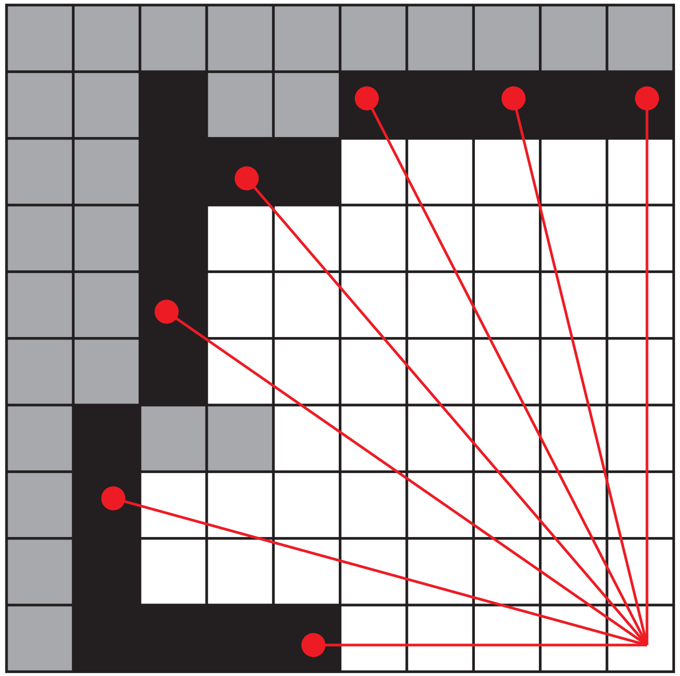
# Beam Model

- Measurement probabilities are too small to be meaningful

$$P(z_t|x_t) = \prod_{i=0}^{N} P(\mathrm{ray}_i = d_i|x_t)$$

- Log odds can be accumulated without numerical underflow

$$\log P(z_t|x_t) = \sum_{i=0}^{N} \log P(\mathrm{ray}_i = d_i|x_t)$$



# Computing $\log P(\text{ray}_i = d_i | x_t)$

- One method is to cast the ray in the grid, and assign log odds to each case
- If ray$_i$ terminates at the first obstacle

$$\log P(\text{ray}_i = d_i | x_t) = -4$$

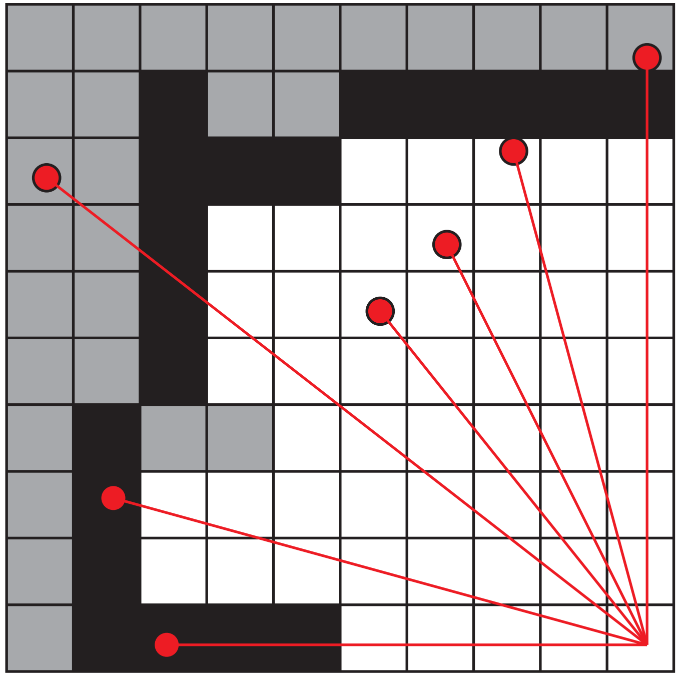
- If ray$_i$ terminates before an obstacle

$$\log P(\text{ray}_i = d_i | x_t) = -8$$

- If ray$_i$ terminates after an obstacle

$$\log P(\text{ray}_i = d_i | x_t) = -12$$

- Take log odds into scan score

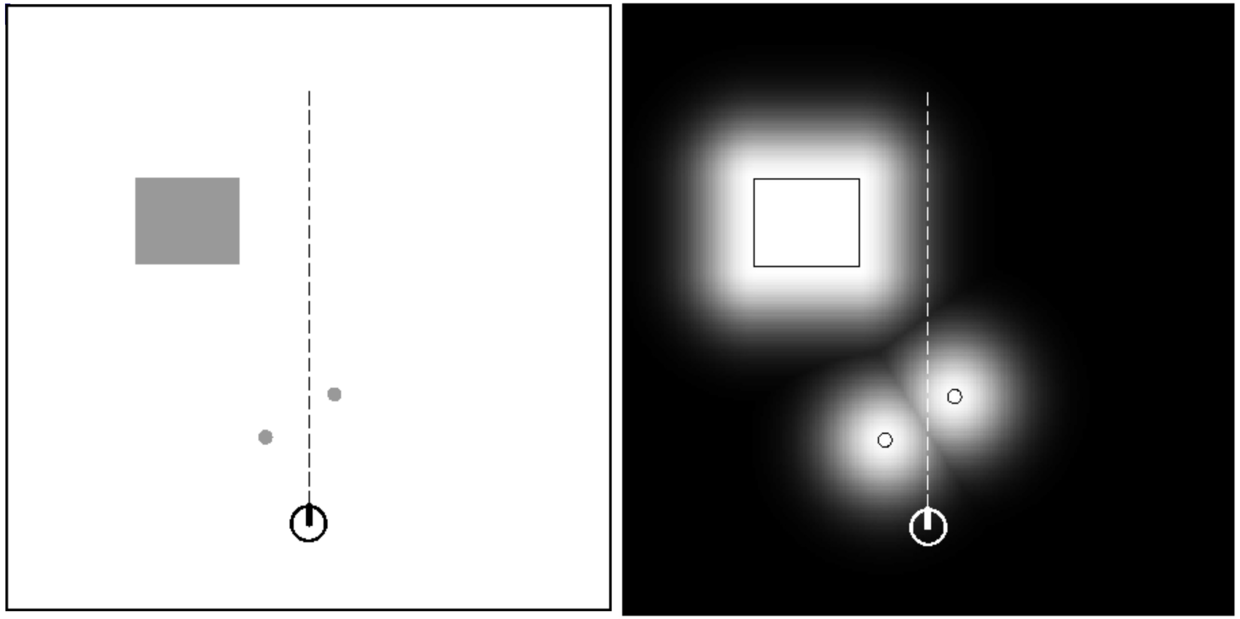


# Sensor Model: Likelihood Field

- Overcomes lack-of-smoothness and computational limitations of Sensor Beam Model
- Idea: Instead of ray casting (computationally expensive) just check the end-point.
- The likelihood $p(z|x_t, m)$ is given by a Gaussian distribution

$$\frac{1}{\sqrt{2\pi\sigma}} \exp(-\frac{d^2}{2\sigma^2})$$

Where $d$ is the distance to the nearest obstacle and $\sigma$ is the std dev.



# Sensor Model: Likelihood Field

**Algorithm likelihood_field_range_finder_model**($z_t, x_t, m$)**:**

$q = 1$

*for all* $k$ *do*

*if* $z_t^k \neq z_{\max}$

$x_{z_t^k} = x + x_{k,\text{sens}} \cos\theta - y_{k,\text{sens}} \sin\theta + z_t^k \cos(\theta + \theta_{k,\text{sens}})$

$y_{z_t^k} = y + y_{k,\text{sens}} \cos\theta + x_{k,\text{sens}} \sin\theta + z_t^k \sin(\theta + \theta_{k,\text{sens}})$

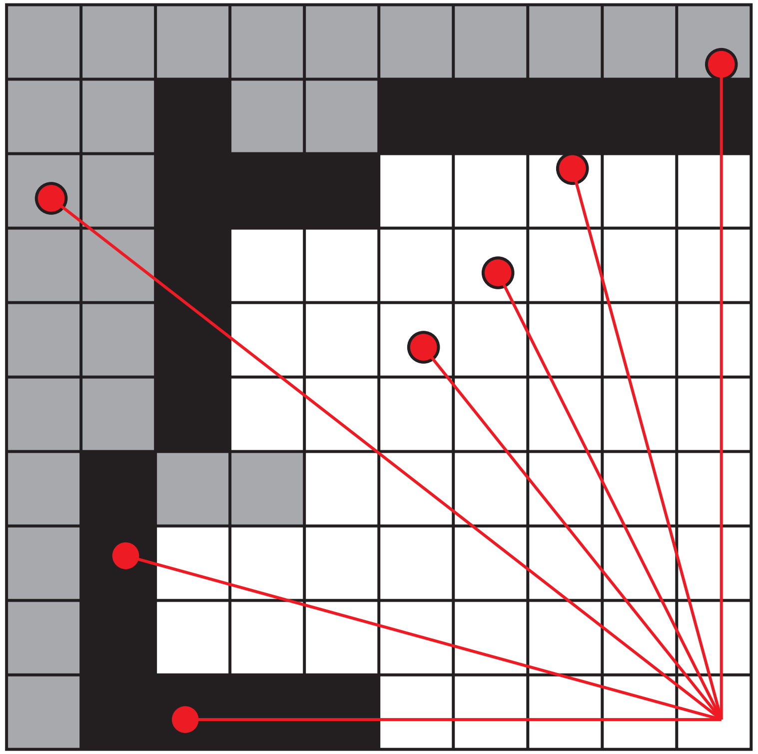
$dist = \min_{x',y'} \left\{ \sqrt{(x_{z_t^k} - x')^2 + (y_{z_t^k} - y')^2} \,\middle|\, \langle x', y' \rangle \text{ occupied in } m \right\}$

$q = q \cdot \left( z_{\text{hit}} \cdot \mathbf{prob}(dist, \sigma_{\text{hit}}) + \frac{z_{\text{random}}}{z_{\max}} \right)$

*return* $q$

# Simplified Likelihood Field Model

- Look at endpoints (like likelihood field)
- Take log odds of cell into scan score if positive
- If it is not a hit, check cell before and after along ray and take fraction of log odds into scan score



# SLAM Implementation

- Initialize Particle Filter
- Loop:

```
updateLocalization();
    updateFilter(odometry, lidar, map);
        resamplePosteriorDistribution(); // resampling using weights
        computeProposalDistribution();   // apply action model
        computeNormalizedPosterior();    // apply sensor model
        estimatePosteriorPose();         // i.e. weighted average
updateMap();     //update with posterior pose
```

# Moving Laser Scan in Sensor Model

```
double SensorModel::likelihood(...){

    double scanScore = 0.0;
    MovingLaserScan movingScan(scan, sample.parent_pose, sample.pose);

    for(const auto& ray : movingScan)
    {
      rayScore=scoreRay(ray);
      scanScore += rayScore;
    }
    return scanScore;
}
```

# Resampling Particles

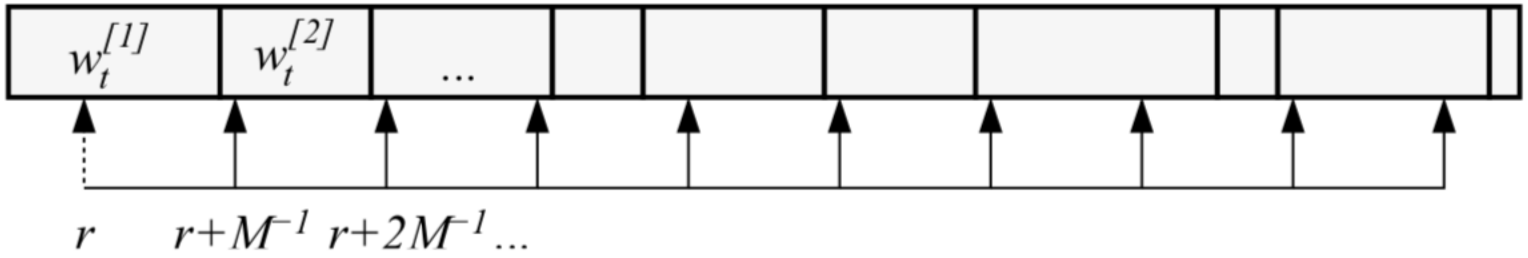
- We want to resample the particle distribution at each iteration
- Sample based on weight of each particle
- Sampling is done *with replacement* – we can potentially sample high weighted particles multiple times
- Repetitive resampling amplifies the variance (not good)
- Should not resample if the robot does not move
- If we randomly chose weighted particles to resample independent of each other we risk losing diversity of the particles

# Low Variance Resampling

**Algorithm Low_variance_sampler($\mathcal{X}_t, \mathcal{W}_t$):**

$\bar{\mathcal{X}}_t = \emptyset$
$r = \text{rand}(0; M^{-1})$
$c = w_t^{[1]}$
$i = 1$
*for* $m = 1$ *to* $M$ *do*
  $U = r + (m-1) \cdot M^{-1}$
  *while* $U > c$
    $i = i + 1$
    $c = c + w_t^{[i]}$
  *endwhile*
  *add* $x_t^{[i]}$ *to* $\bar{\mathcal{X}}_t$
*endfor*
*return* $\bar{\mathcal{X}}_t$

- For resampling $M$ particles, Choose a random number $r$
- Select particles by repeatedly adding $M^{-1}$ to $r$ and choosing the corresponding particle
- $c$ accumulates weights to handle resampling higher weighted particles more often



# Losing Diversity

- Loss of diversity can be caused by sampling a discrete distribution

- Solution: “regularization”
  - Consider particles represent a continuous distribution
  - Sample from the continuous density

  - Example: Given 1D particles: $\{x^{(1)}, x^{(2)}, \ldots, x^{(K)}\}$

  - Sample from the density: $p(x) = \sum_{k=1}^{K} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(x-x^{(k)})^2}{\sigma^2}}$

# Particle Deprivation

- Happens when there are no particles in the vicinity of the correct state
- Occurs as the result of the variance in random sampling. An unlucky series of random numbers can wipe out all particles near the true state.
  - This has non-zero probability to happen at each time so it will happen eventually
- Popular solution: add a small number of randomly generated particles when resampling
  - Advantage: reduces particle deprivation, is simple
  - Disadvantage: incorrect posterior estimate even in the limit of infinitely many particles

# Posterior Pose Estimate

- Particle representation of distribution may not contain true pose, especially with few particles.
- You will need to come up with a robust method to estimate the posterior pose to feed into the mapping step.
- Many options:
  - Weighted average of poses?
  - Average of “good” poses (i.e. top 10% of weights)?
  - K-means clustering?
  - Depends on the variance of distribution?

# Averaging Angles

- Arithmetic mean is not appropriate for mean of angles
- Find mean in Cartesian coordinates and convert back with atan2.

$$\bar{\theta} = \mathrm{atan2}(\sum_{i=0}^{N} w_i \sin(\theta_i), \sum_{i=0}^{N} w_i \cos(\theta_i))$$

# Putting it together

- Mapping
  - slam –mapping-only
  - botgui
  - log-player-gui <log-file> (turn off SLAM_MAP & SLAM_PARTICLES)
- Action Model
  - slam –action-only –localization-only <map_file>
  - botgui
  - log-player-gui <log-file> (turn off SLAM_MAP & SLAM_POSE & SLAM_PARTICLES)
- Sensor Model
  - slam –localization-only <map_file>
  - botgui
  - log-player-gui <log-file> (turn off SLAM_MAP & SLAM_POSE & SLAM_PARTICLES)
- Full Slam
  - slam
  - botgui
  - log-player-gui <log-file> (turn off SLAM_MAP & SLAM_POSE & SLAM_PARTICLES)

# A problem with the logs

- RPLidar driver changed from W22 to F22
- Before – LIDAR scan clockwise
- After – LIDAR scan counter clockwise
- Solution:
  - Add a command line arg for direction of LIDAR
  - If -CW use $2\pi - \theta_{ray}$
  - If -CCW use $\theta_{ray}$
  - Old logs will be CW, but new data will be CCW