COM SCI 188

# Intro to Robotics

Lecture 8

Yuchen Cui

Winter 2026

## Agenda

- Announcements

- Particle Filter (PR4)
- SLAM (PR 10)

## Announcements

- Coding Assignment 2
    - Camera Calibration in Robosuite
    - <Demo>

## Announcements

- Problem Set 2 due Monday
- Coding Assignment 2 due next Friday

# State Estimation & Particle Filter

Slides adapted from USC CSCI 445: Introduction to robotics
Credit: Erdem Biyik, Heather Culbertson

## Probabilistic Robotics

- Robotics is by nature a very messy subject
  - Sensors are noisy
  - Actuators are imperfect

- We rarely ever know anything “for sure”
  - We can only collect evidence to try to make educated assumptions

## For Example…

- An IR rangefinder can tell us
    - if we are likely to be near a wall or not
    - if its likely that there is something close to us
    - if its likely that the area ahead of us is unobstructed

- A camera can tell us
    - if there is a good chance of a colored stuffed doll being in front of us
    - if its possible that there is a box on the table
    - If its likely the walls are blue

## However…

- We are making a big leap between sensing and perception
    - When a rangefinder gives us a certain voltage that indicates an obstacle on the path, *it doesn't guarantee that there is an actual obstacle in our path*
    - However, we can definitely say that if our rangefinder reports that voltage, there may be a ***better chance*** of an obstacle being in our way

Instead of considering a sensor
output as a *certainty*, we can think of the
*likelihood* that it is correct

## Probabilities

- We will use probabilistic representations for
  - World State
  - Sensor Models
  - Action Models

- Use the *calculus of probability theory* to combine these models

## Quick Probability Review

- "Probability of x" – $P(x)$
  - P(x) is a real number between 0 and 1 representing the "percent chance" of x occurring

- "Probability of x *and* y" – $P(x,y)$
  - If x and y are independent – $P(x)P(y)$
  - Otherwise - $P(x \mid y)P(y)$

- "Probability of x *or* y"
  - If x and y are mutually exclusive – $P(x) + P(y)$
  - Otherwise – $P(x) + P(y) - P(x,y)$

## Quick Probability Review

- For example:
  - P(rain tomorrow) = 0.05
  - P(I wear green next week) = 0.93
    - P(Rain tomorrow, and I wear green next week) = 0.05 * 0.93 = 0.046
    - Note that we can argue *independence* for these two events

## Quick Probability Review

- “Probability of x *given* y” = P(x | y)
  - What is the probability of x, given that we have the prior knowledge of y being true
  - E.g. P(wearing raincoat | rainy outside)

P( outerwear | weather conditions )

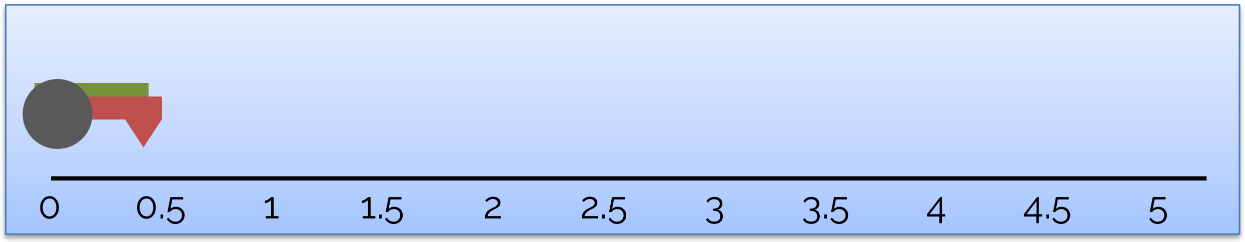
| | Raining Outside | Cloudy Outside | Sunny Outside |
|---|---|---|---|
| Wearing raincoat | 1.0 | 0.3 | 0.1 |
| Not wearing raincoat | 0.0 | 0.7 | 0.9 |

## Probabilistic Localization

- **Goal**: use probabilistic methods to represent both the motion and the perception of your robots.
- We will use a “Particle Filter” to represent these probability distributions

## Probability Filter Motion Models

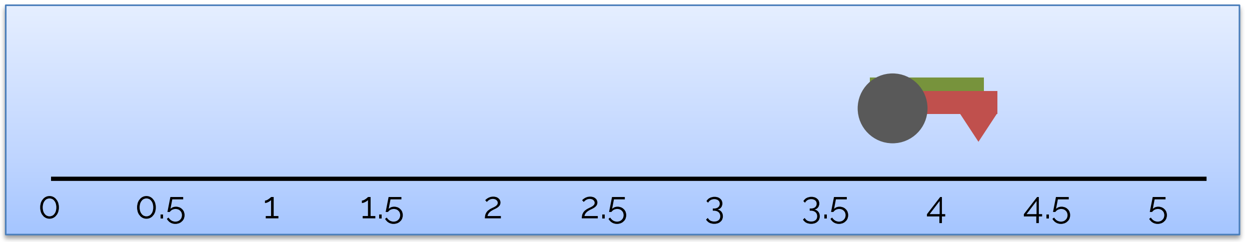
- We can describe every movement of your robots as a probability distribution
    - For example, let's say we want our robot to just move forward for 3.5 feet



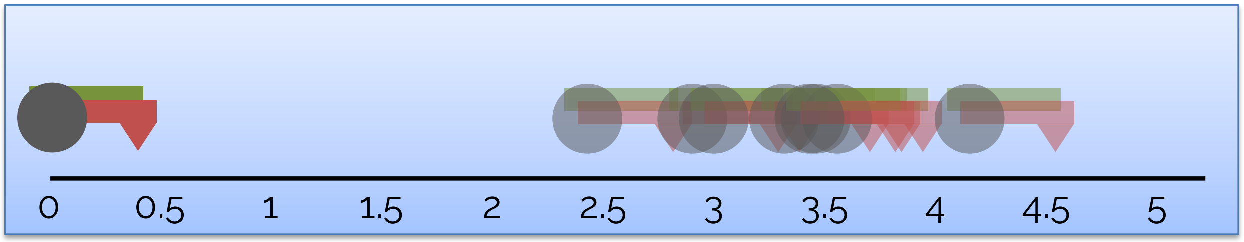
## Probability Filter Motion Models

- We can describe every movement of your robots as a probability distribution
    - For example, let's say we want our robot to just move forward for 3.5 feet

*The motors are subject to noise!*

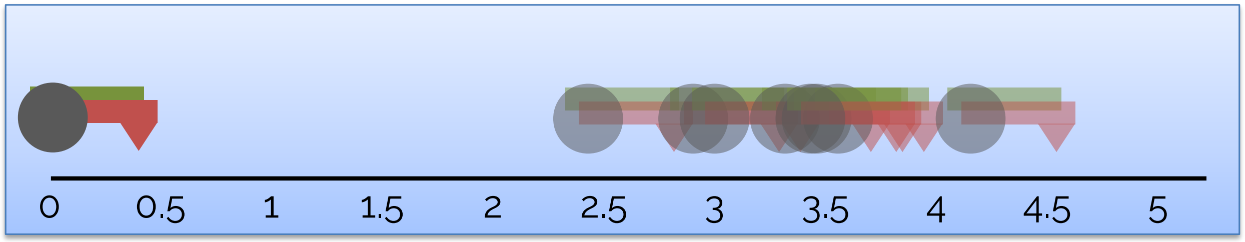


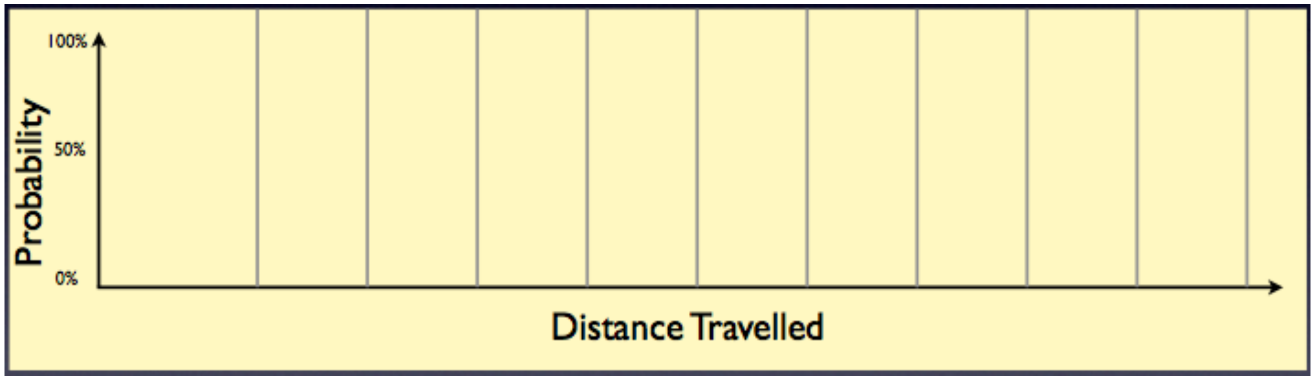
## Probability Filter Motion Models



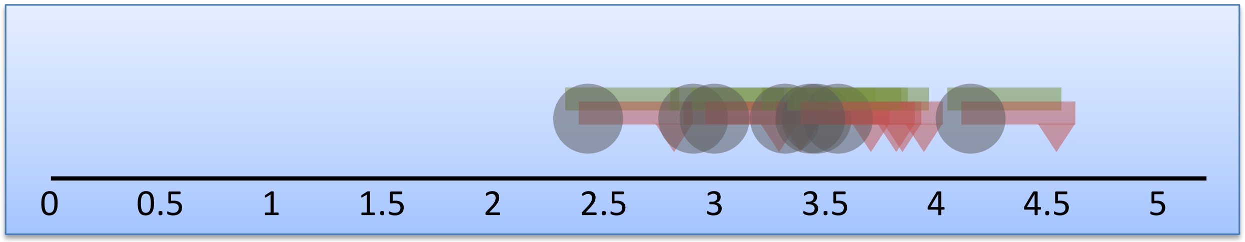
*What if we were to tell our robot to move forward 3.5 feet 100 different times?*

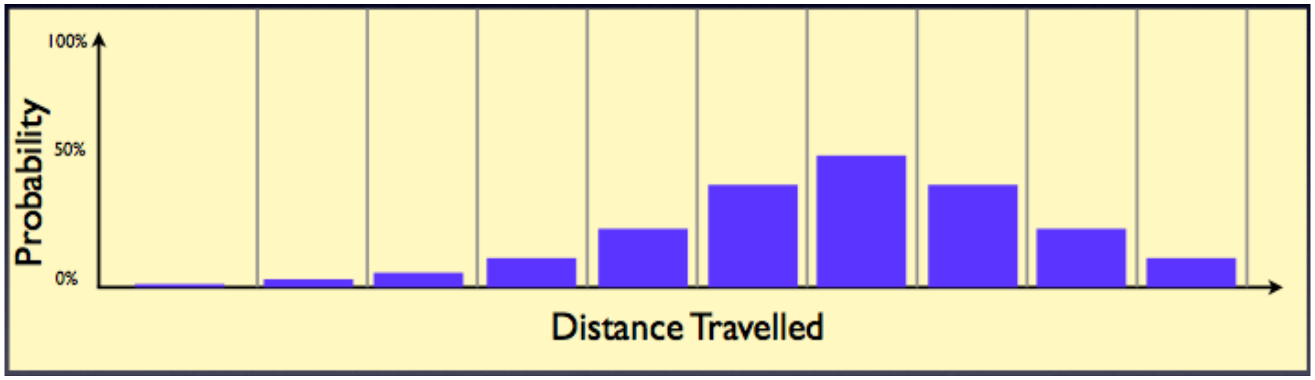
It would likely land in
100 different locations



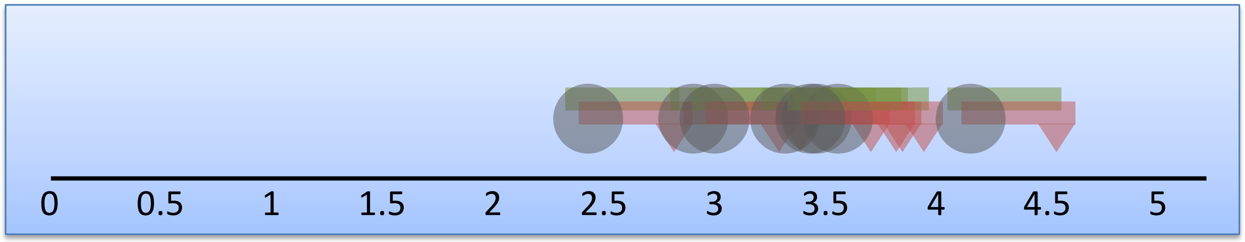


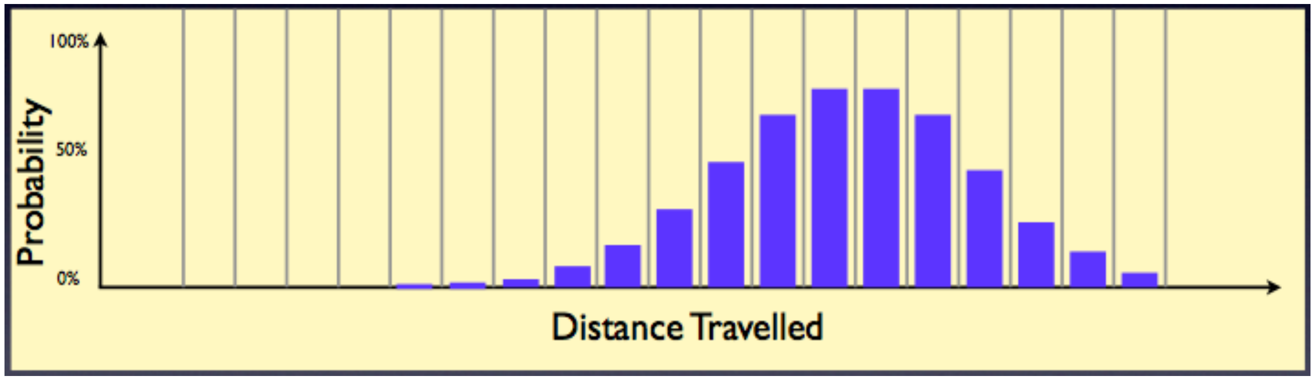
Let's break the track into 0.5 foot increments, and calculate the percentage of times our robot lands in each increment.





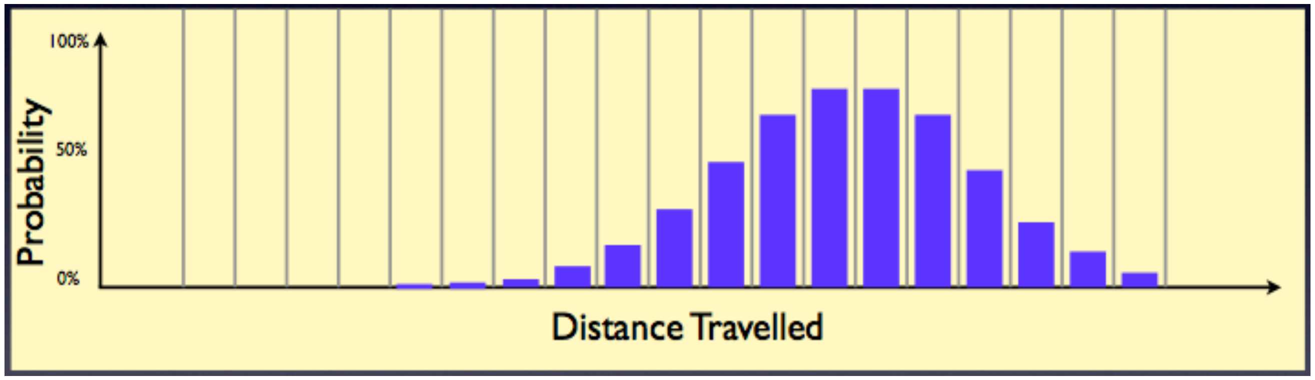
Let's break the track into 0.5 foot increments, and calculate the percentage of times our robot lands in each increment.





We can do the same thing even with smaller increments

This “bell curve” shape is very common when describing noisy processes

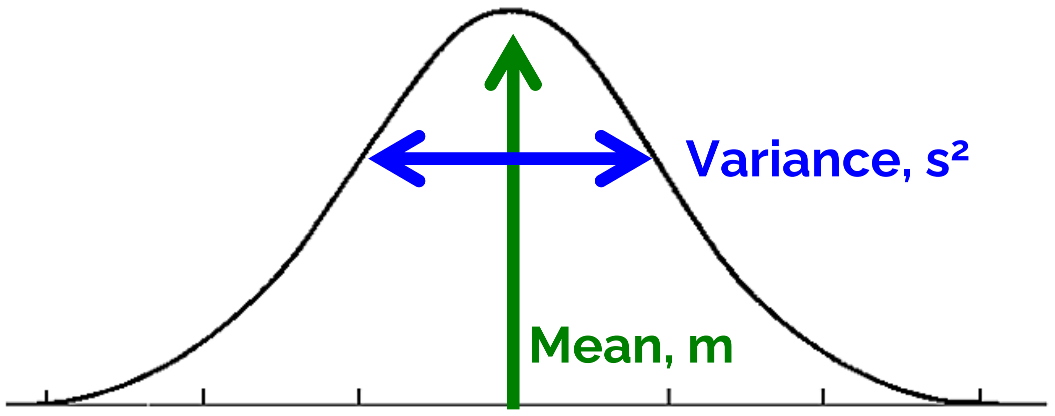


It is called the “Gaussian” or “Normal” distribution

## Gaussian Distribution

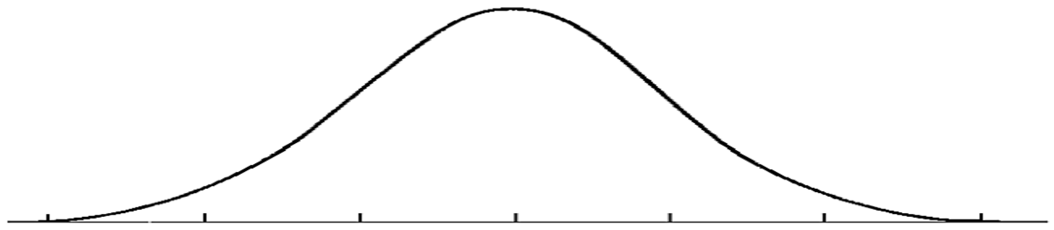
- Gaussian Distribution isn't the only way to describe a random process, but it is one of the easiest and most flexible.

- It often does a pretty good job of describing our physical systems



## Some Details…

This function is the *probability density function (PDF)* of the Normal Distribution



$$\mathcal{N}(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

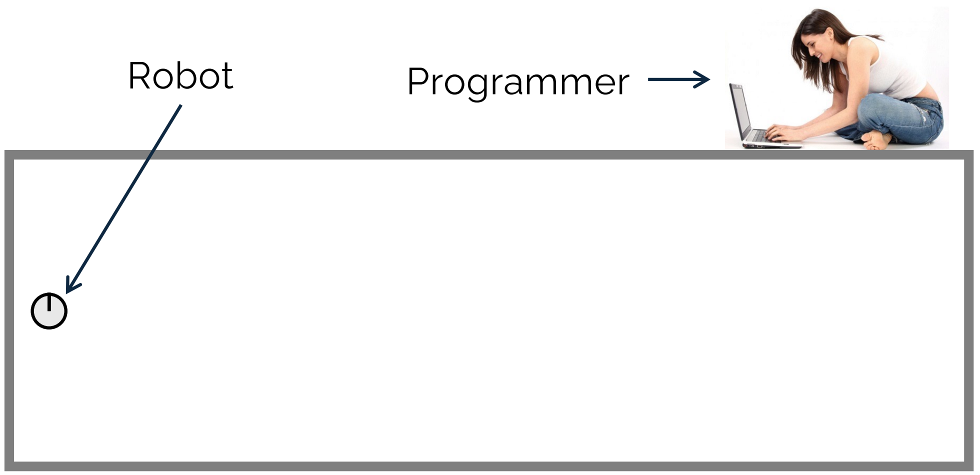
It describes the probability of getting a value of x if x is sampled from a Normal Distribution with mean m and variance s²

## An Example

Let's say we have a robot with a compass.

This robot accepts commands to turn to particular angles (which it does very well)

The robot also accepts commands to move forward (which it does with some rotational and translational noise)

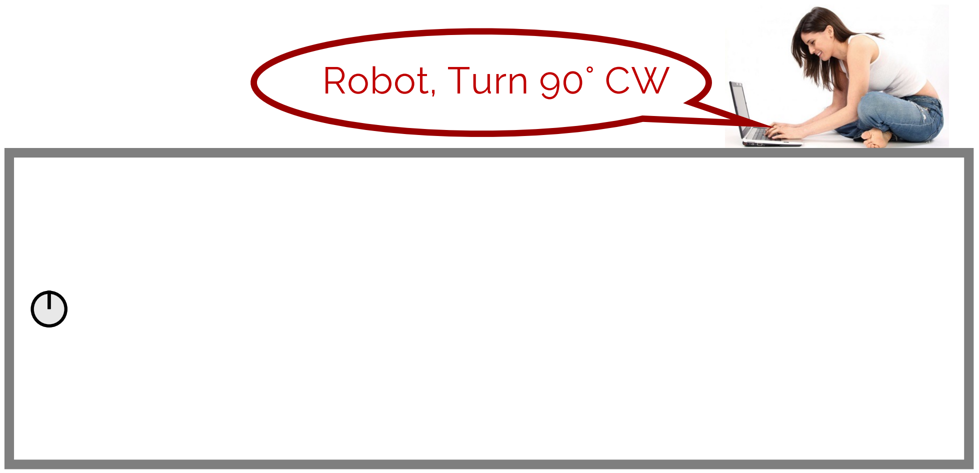


## An Example

Let's say we have a robot with a compass.

This robot accepts commands to turn to particular angles (which it does very well)

The robot also accepts commands to move forward (which it does with some rotational and translational noise)

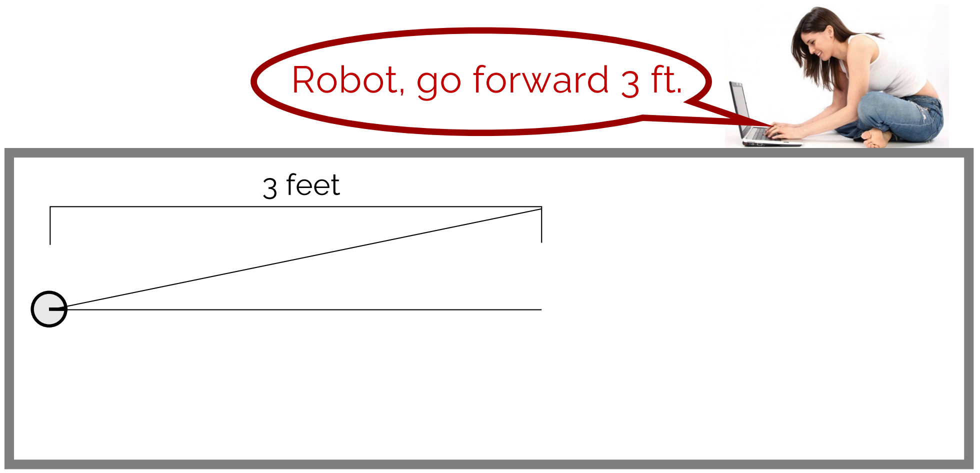


## An Example

Our Robot's motion is noisy!

*What's a simple way we can model noise?*

Let's say that each time we try to move in a straight line, our robot goes almost the right direction and distance, but with some Gaussian noise on both the direction and distance.



## Simple Model

$$\Theta = \text{Desired Movement Direction}$$
$$d = \text{Desired Movement Distance}$$
$$(x_t, y_t) = \text{Current Robot Position}$$

$$\Theta' = \text{Actual Movement Direction}$$
$$d' = \text{Actual Movement Distance}$$
$$(x_{t+1}, y_{t+1}) = \text{Simulated Noisy Robot Position}$$

$$\mathcal{N}(\mu, \sigma) = \text{Normal Random Variable Sample}$$
$$\sigma_\Theta^2 = \text{Direction Variance}$$
$$\sigma_d^2 = \text{Distance Variance}$$

$$\Theta' = \Theta + \mathcal{N}(0, \sigma_\Theta)$$
$$d' = d + \mathcal{N}(0, \sigma_d)$$

model

$$x_{t+1} = x_t + d' \cos \Theta'$$
$$y_{t+1} = y_t + d' \sin \Theta'$$

propagate

This is the simplest way to predict our robot's motion

There are much more complex and accurate models available, but we will use this one for simplicity.

## An Example

*How can we estimate the 2D position of our robot?*

Simulate the movement of a whole bunch of robots, each with its own random Gaussian noise.

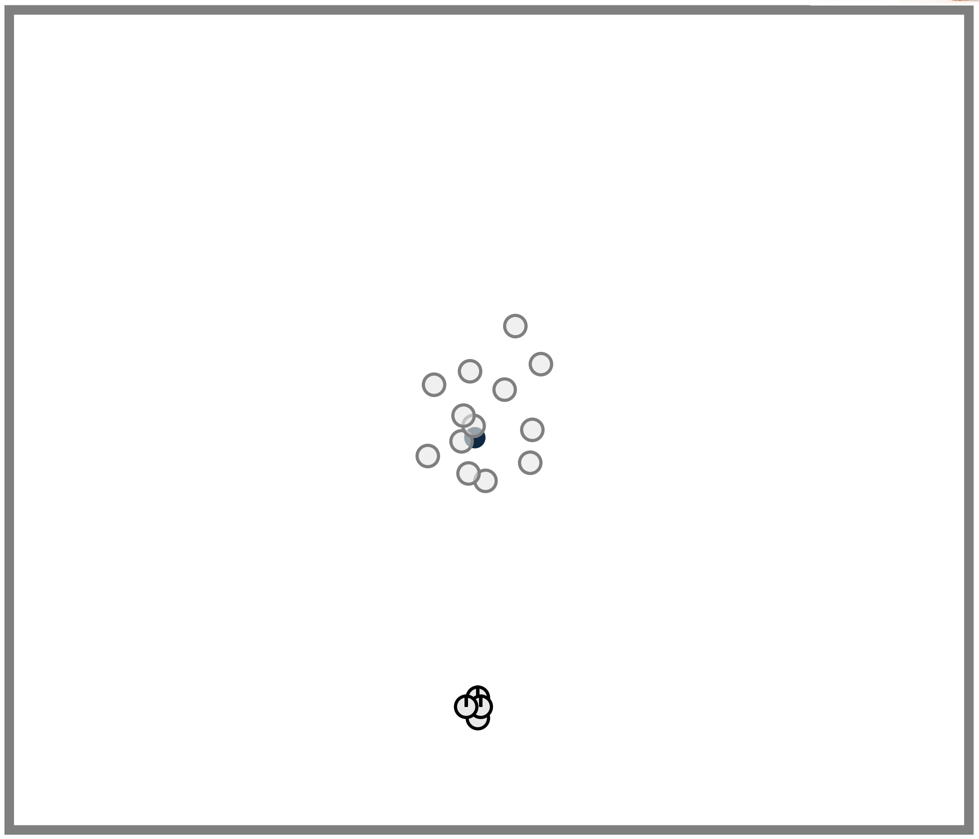
*What will happen to these "virtual" robots?*

They will scatter according to the amount of noise we add.

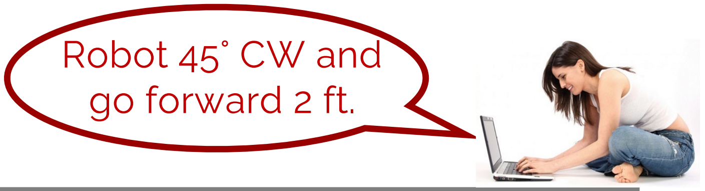
*If our noise model is a good approximation of real life…*

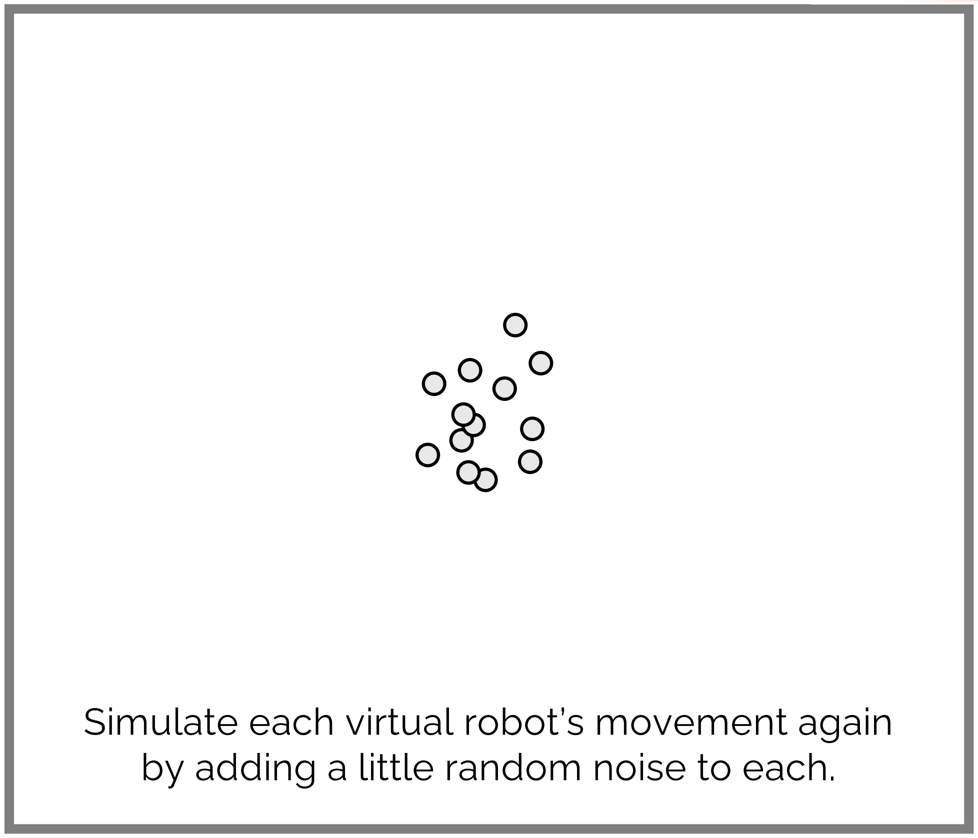
then the distribution of our virtual robots will describe the probability distribution of our real robot's location.

## An Example

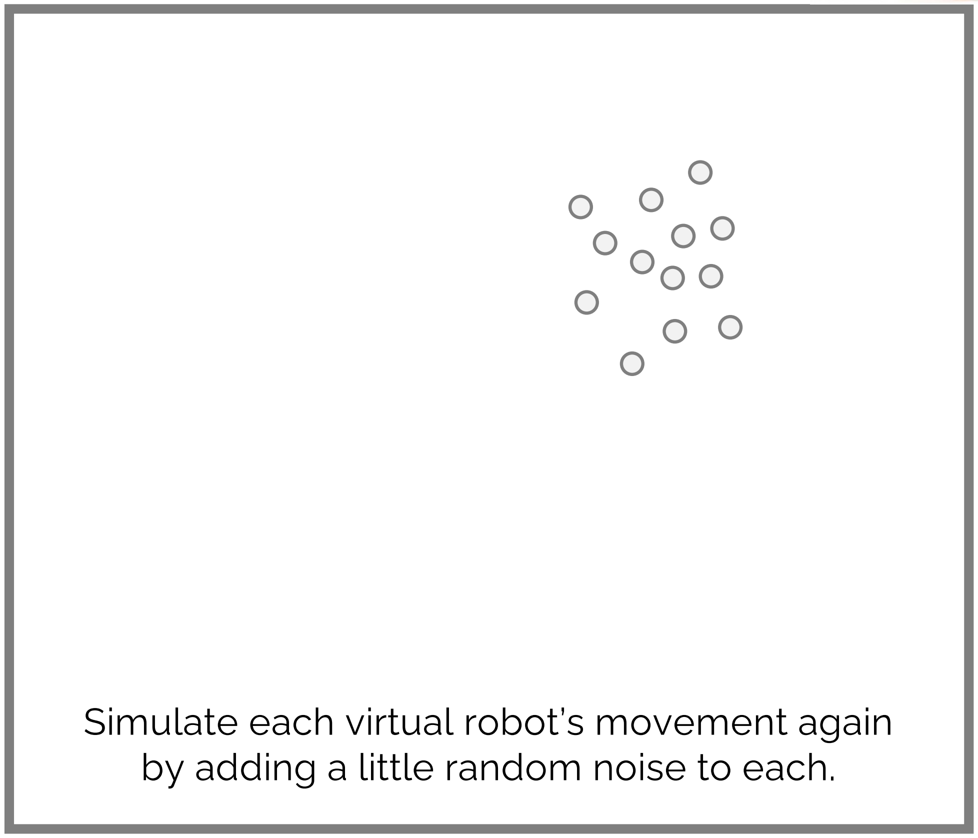


## An Example




Simulate each virtual robot's movement again by adding a little random noise to each.

## An Example



Simulate each virtual robot's movement again by adding a little random noise to each.

## Measurement Models

Obviously, if we keep moving around and adding noise with each step, all of our virtual robots will eventually be completely scattered.

*But how can we can assess the likelihood of each "virtual robot?"*

By taking measurements!

## Measurement Models

So, we would like to assess the probability of our robot actually being at one of our simulated robots' positions given some new sensor reading.

P(robot @ location | sensor reading)

*How do we calculate this?*

## Measurement Models

So, we would like to assess the probability of our robot actually being at one of our simulated robots' positions given some new sensor reading.

P(virtual robot) = P(robot @ location | sensor reading)

*How do we calculate this?*

## Measurement Models

P(robot @ location | sensor reading)

**Bayes Theorem!**

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

$$\text{P(robot @ location | sensor reading)} = \frac{\text{P(sensor reading | robot @ location) P(robot @ location)}}{\text{P(sensor reading)}}$$

## Bayes' Theorem Example

Let's say there's a rare disease that affects 1 in 1,000 people.

There's a test for it that's 99% accurate — meaning:

- If you have the disease, it gives a positive result 99% of the time (true positive).
- If you don't have the disease, it gives a negative result 99% of the time (true negative).

You take the test and it comes back positive. Should you be worried?

## Bayes' Theorem Example

- $A$: You have the disease
- $B$: You test positive

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}$$

- $P(A) = 0.001$ (1 in 1,000)
- $P(B|A) = 0.99$ (test is 99% accurate if you're sick)
- $P(B|\neg A) = 0.01$ (1% false positive rate)
- $P(\neg A) = 0.999$

$$P(B) = P(B|A) \cdot P(A) + P(B|\neg A) \cdot P(\neg A)$$
$$P(B) = 0.99 \cdot 0.001 + 0.01 \cdot 0.999 = 0.01098$$

$$P(A|B) = \frac{0.99 \cdot 0.001}{0.01098} \approx 0.09$$

Even with a positive test, there's only about a 9% chance you actually have the disease.

## Measurement Models

What does this mean,
and how do we calculate it?

$$\text{P(robot @ location | sensor reading)} = \frac{\boxed{\text{P(sensor reading | robot @ location)}}\ \text{P(robot @ location)}}{\text{P(sensor reading)}}$$

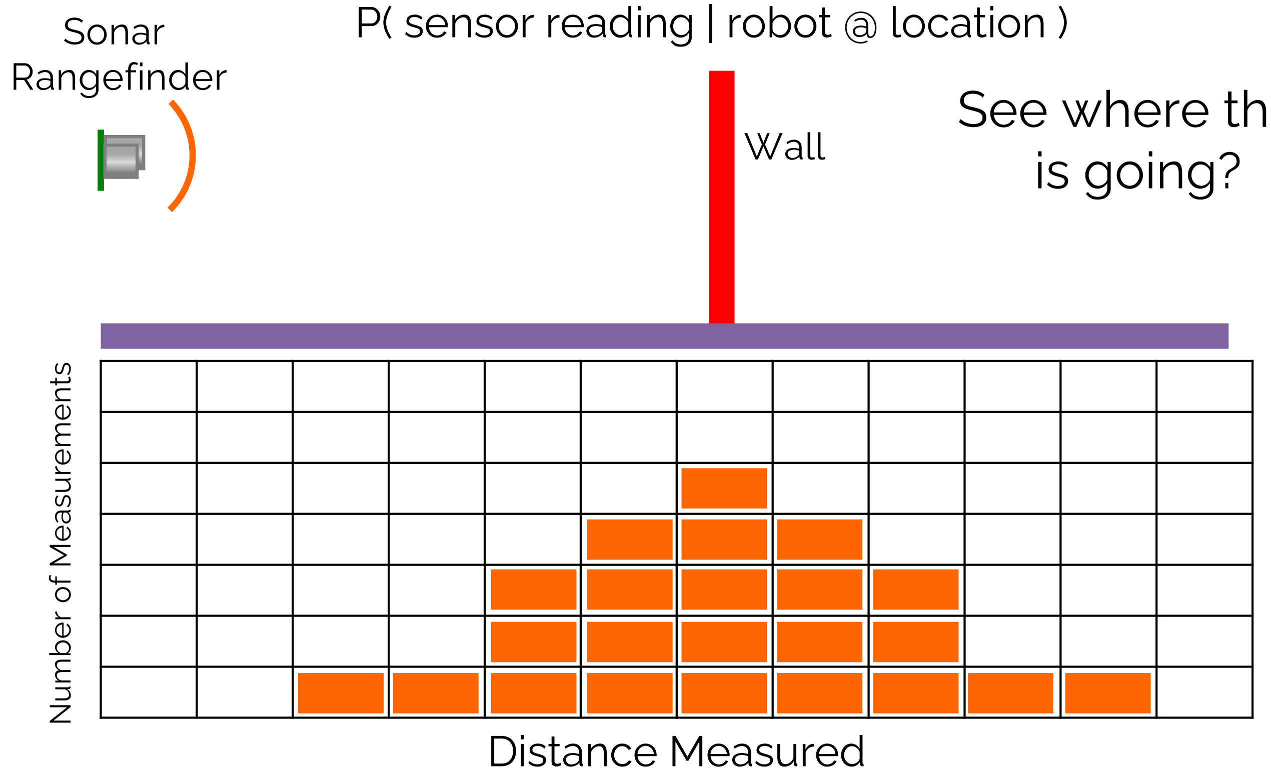
## Measurement Models

P( sensor reading | robot @ location )

Just like our motion model, our sensors are subject to noise, and can be modeled as a probability distribution.

## Measurement Models

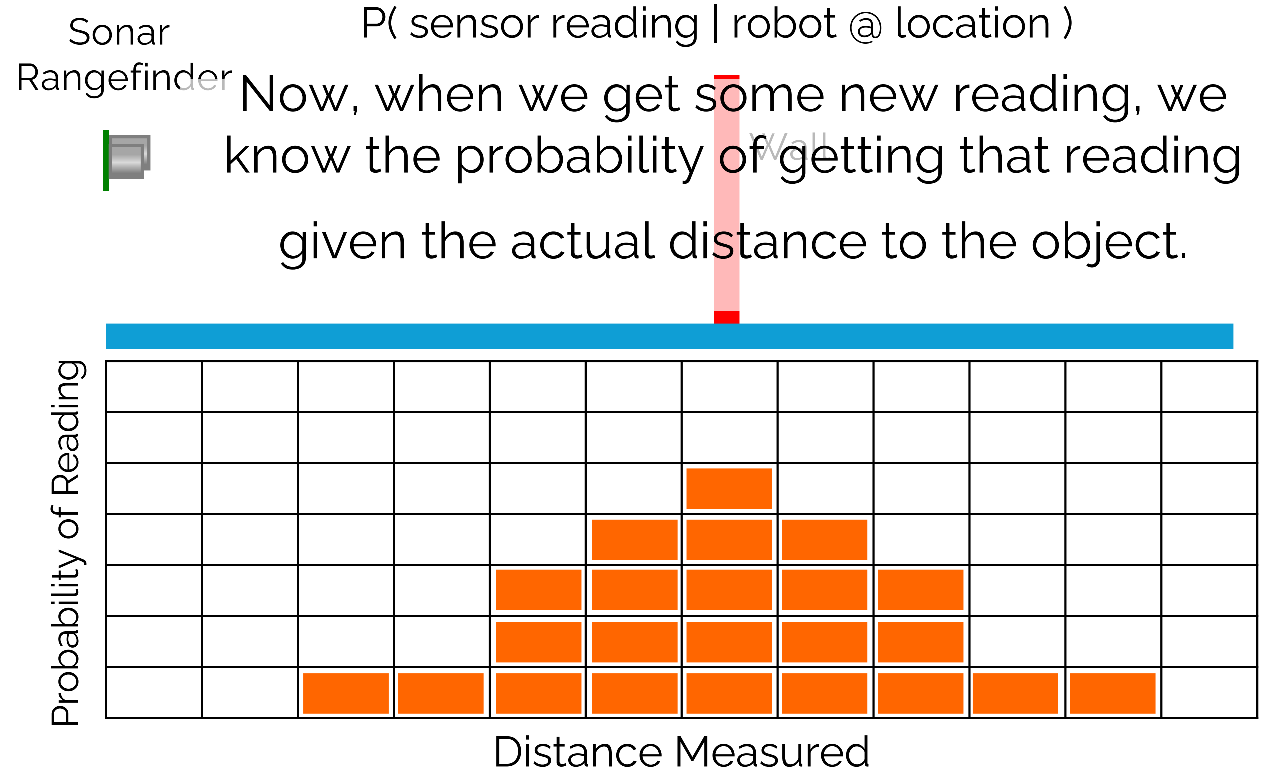
P( sensor reading | robot @ location )



See where this is going?

## Measurement Models

P( sensor reading | robot @ location )



Now, when we get some new reading, we know the probability of getting that reading given the actual distance to the object.

Sensor Noise

Location of virtual robot

Reading from sensor

P( sensor reading | robot @ location ) = $\mathcal{N}$( reading; location, s )

Remember (PDF): $\mathcal{N}(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$

Every time we take a reading, we can assess the probability of getting that reading from each of our virtual robot positions

$$\text{P(robot @ location | sensor reading)} = \frac{\text{P(sensor reading | robot @ location) P(robot @ location)}}{\text{P(sensor reading)}}$$

If at each timestep, we calculate the probability of each virtual robot, then we can use those probabilities from the last timestep here

$$P(\text{robot @ location} \mid \text{sensor reading}) = \frac{P(\text{sensor reading} \mid \text{robot @ location})\ P(\text{robot @ location})}{P(\text{sensor reading})}$$

Now all we have left is P(sensor reading)

This value is a bit less understandable, so for the sake of simplicity, lets make this easy…

$$P(\text{robot @ location} \mid \text{sensor reading}) = \frac{P(\text{sensor reading} \mid \text{robot @ location})\, P(\text{robot @ location})}{P(\text{sensor reading})}$$

P(robot @ location x | sensor reading) = N* P(sensor reading | robot @ location) P(robot @ location)

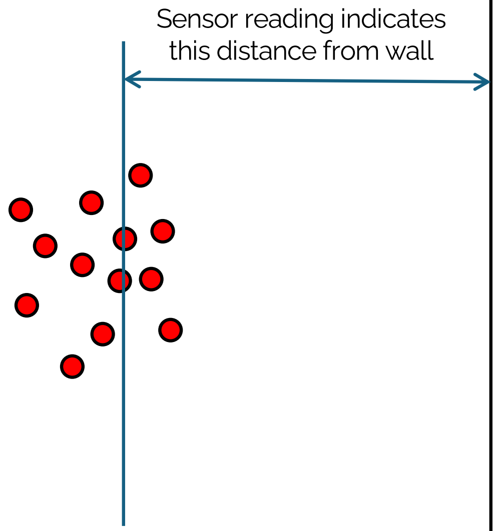
Each time we take a reading, and update the probability of each virtual robot, let's choose N such that all probabilities sum to 1.

$$N = \frac{1}{\sum_{x} \text{P(sensor reading | robot@location) P(robot@location)}}$$

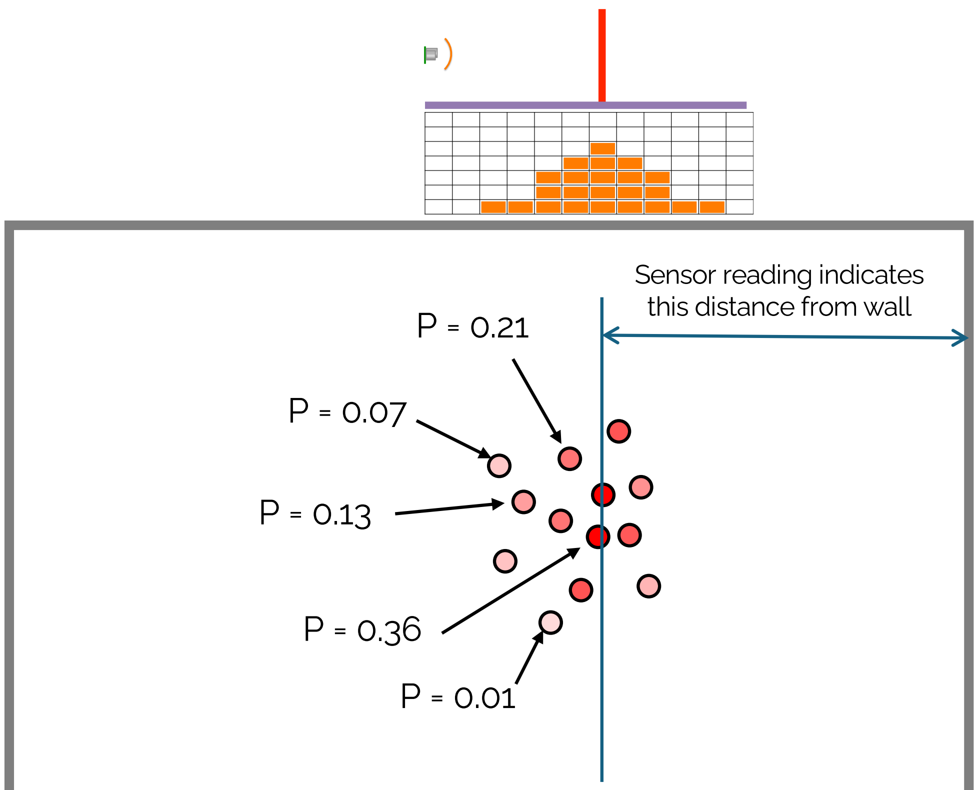
We know know everything we need to calculate

P(robot @ location x | sensor reading)

We call the result of this calculation the
"**Posterior Probability**"



Take a single reading, and compute the probability of each virtual robot given that reading



Take a single reading, and compute the probability of each virtual robot given that reading

- Calculate estimated robot's position
  - Weighted average of particles' positions using their probability

$$x_{est} = \sum_n P_n x_n$$

$$y_{est} = \sum_n P_n y_n$$

## Resampling

- Now that we can assess the probability of each of our virtual robots' positions given a new sensor measurement, let's kill off some of the virtual robots with lower probabilities

- There are quite a few ways to do this, but *resampling* is one efficient method

## Resampling

- Use a roulette wheel to probabilistically duplicate particles with high weights, and discard those with low weights.
- A 'Particle' is some structure that has a weight element w. The sum of all weights in old Particles should equal 1.
- Calculate a Cumulative Distribution Function (CDF) for our particle weights
- Loop through our particles as if spinning a roulette wheel.

» Each particle will have a probability of getting landed on and saved proportional to its posterior probability.

» If a particle has a very large posterior probability, then it may get landed on many times.

» If a particle has a very low posterior probability, then it may not get landed on at all.

» By incrementing by 1/(numParticles) we ensure that we don't change the number of particles in our returned set.

## Resampling: practical considerations

- Resampling just chooses particles (with repetition) with the computed probabilities: *it does not change the number of particles*
  - it "kills" some and "copies" others
- One extension is to have the particle number dynamic, based on the "confidence" of the current estimate
- If probabilities get too low, there could be numerical issues
  - Use logarithm when calculating P(virtual robot):

$$p = p_1 p_2 \rightarrow \log(p) = \log(p_1) + \log(p_2)$$

## Resampling: practical considerations

- *Do we really need to resample our particles every time we take a measurement?*

*Resampling is used to avoid the problem of degeneracy of the algorithm, that is, avoiding the situation that all but one of the importance weights are close to zero.*

No, we can calculate the number of effective particles:

$$N_{eff} = \frac{1}{\sum_i (particle_i.prob)^2}$$

And only resample when

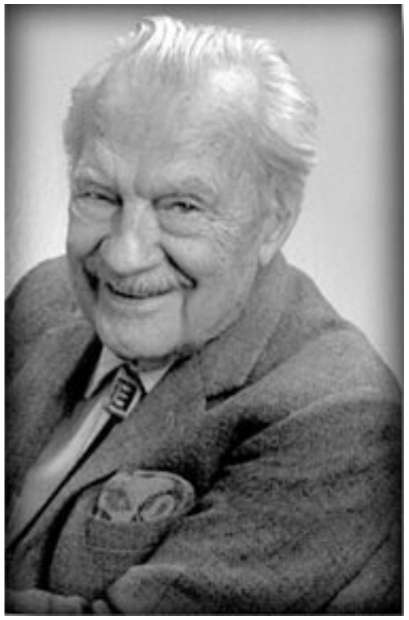
$$N_{eff} < N_{thresh}$$

## Particle Filter

- Notice we refer to our “virtual robots” as “particles”
- The algorithm we’ve put together in this lecture is called a “Particle Filter”
- Algorithm for robots to localize using a particle filter is called

  ***Monte Carlo Localization*** or *particle filter localization*

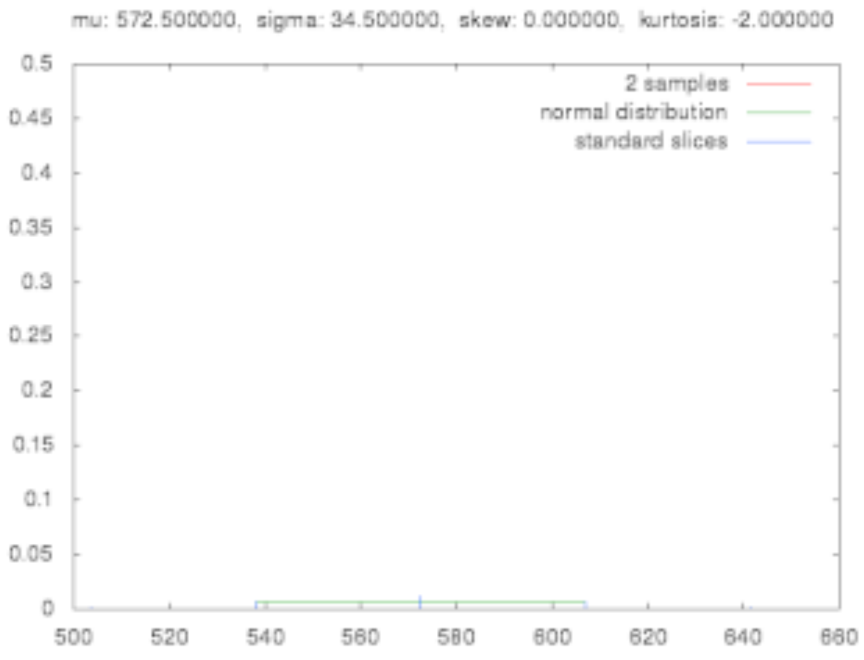
# Monte Carlo Methods

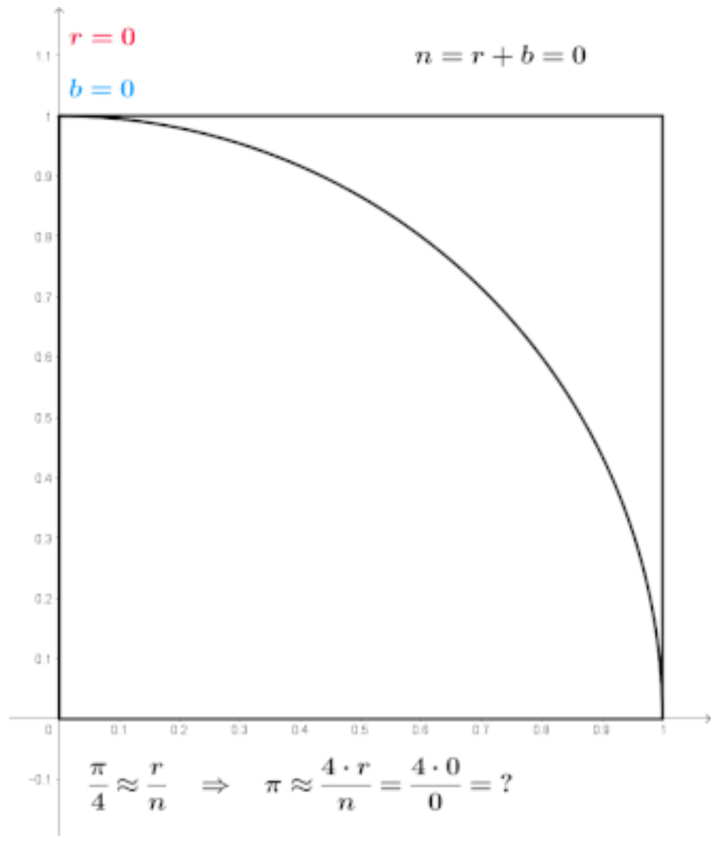

Nicholas Metropolis

John von Neumann

Stanisław Ulam

## random sampling + statistical analysis





$$\frac{\pi}{4} \approx \frac{r}{n} \quad \Rightarrow \quad \pi \approx \frac{4 \cdot r}{n} = \frac{4 \cdot 0}{0} = ?$$

## Particle Filtering for Localization

**Initialize:** Create N particles at some starting location (or distributed randomly around the map), each with equal probability. Call this data structure "Particles"

**Move:** When a new **movement command (d, $\Theta$)** is issued:

For each particle "p" in "Particles":
Generate a randomized movement command consisting of d' and $\Theta$'

$$\Theta' = \Theta + \mathcal{N}(0, \sigma_\Theta) \qquad d' = d + \mathcal{N}(0, \sigma_d)$$

Update the current position of "p" according to the motion model applied to d' and $\Theta$'

$$x_{t+1} = x_t + d' \cos \Theta' \qquad y_{t+1} = y_t + d' \sin \Theta'$$

Optionally do bounds checking to ensure that our particles are not ghosting through walls

**Update:** When a new **sensor measurement ("z")** is received
For each particle "p" in "Particles":
- Compute the posterior probability of each particle: P("p",location | "z" )

**Resample** the particles: "Particles" = resampleParticles("Particles")

## Posterior probabilities

When a new sensor measurement ("z") is received

For each particle "p" in "Particles":

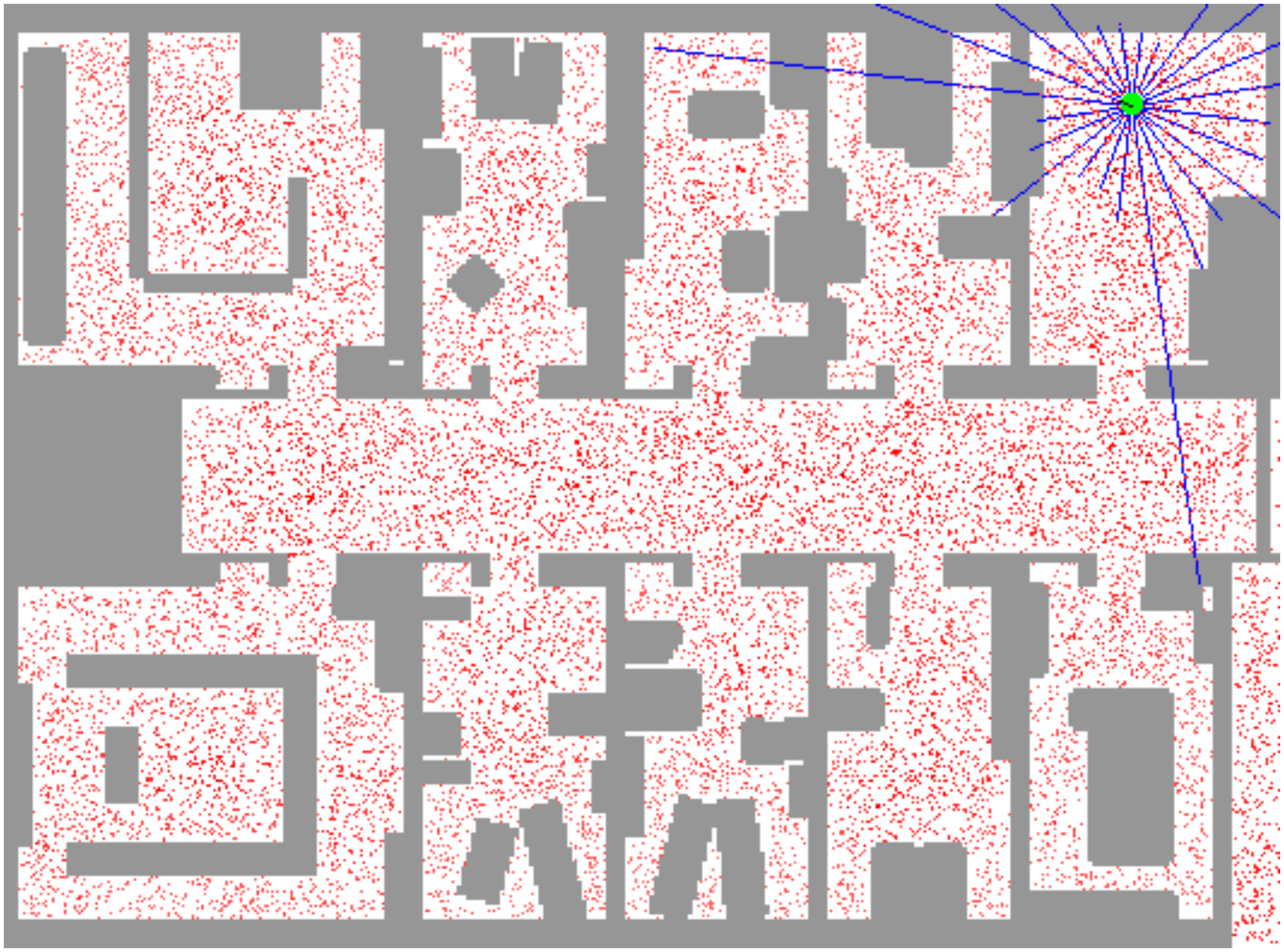
Compute the posterior probability of each particle: P("p",location | "z" )

$$P(\text{robot @ location} \mid \text{sensor reading}) = \frac{P(\text{sensor reading} \mid \text{robot @ location})\, P(\text{robot @ location})}{P(\text{sensor reading})}$$

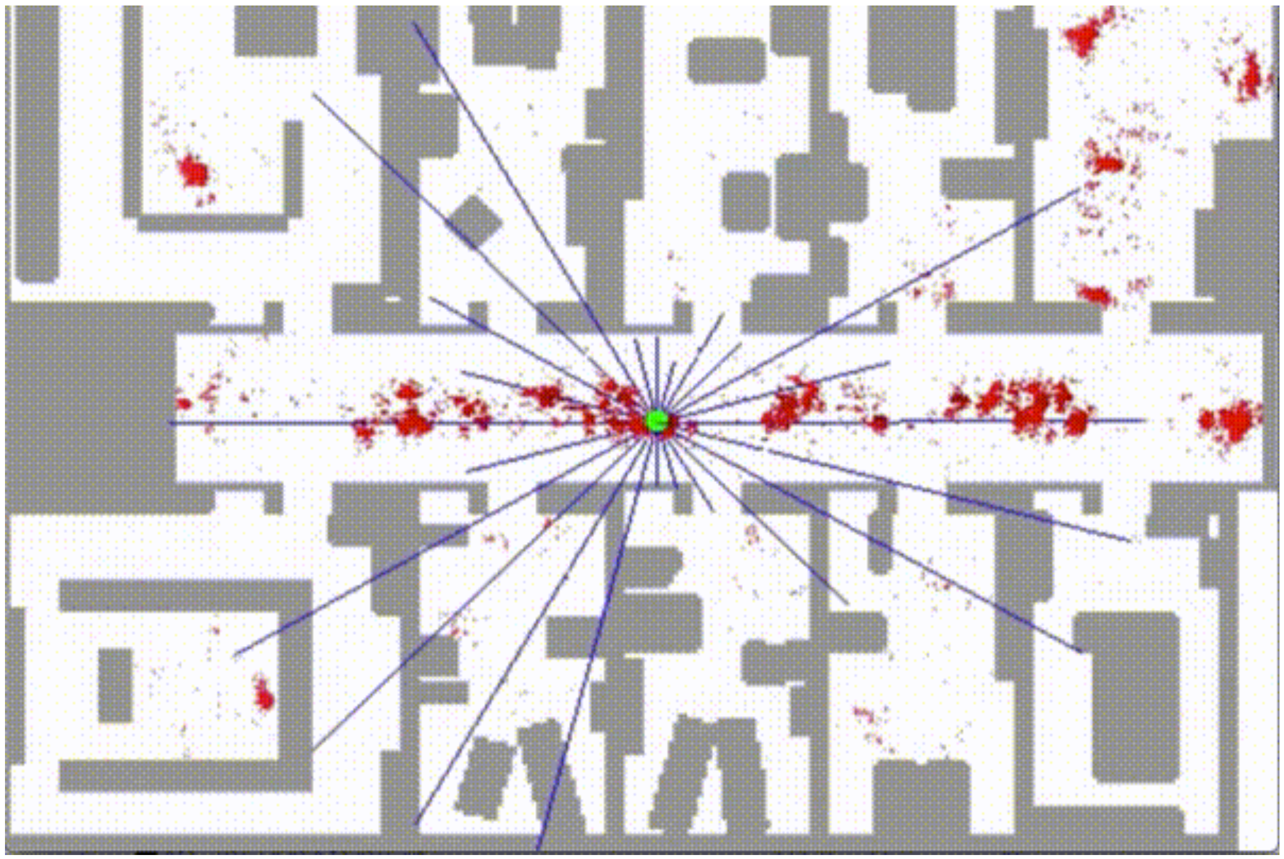
$$P(\text{ sensor reading} \mid \text{robot @ location }) = \mathcal{N}(\overset{\text{Reading from sensor}}{\downarrow\ \text{reading}};\ \overset{\text{Location of virtual robot}}{\downarrow\ \text{location}},\ \overset{\text{Sensor Noise}}{\downarrow\ s})$$

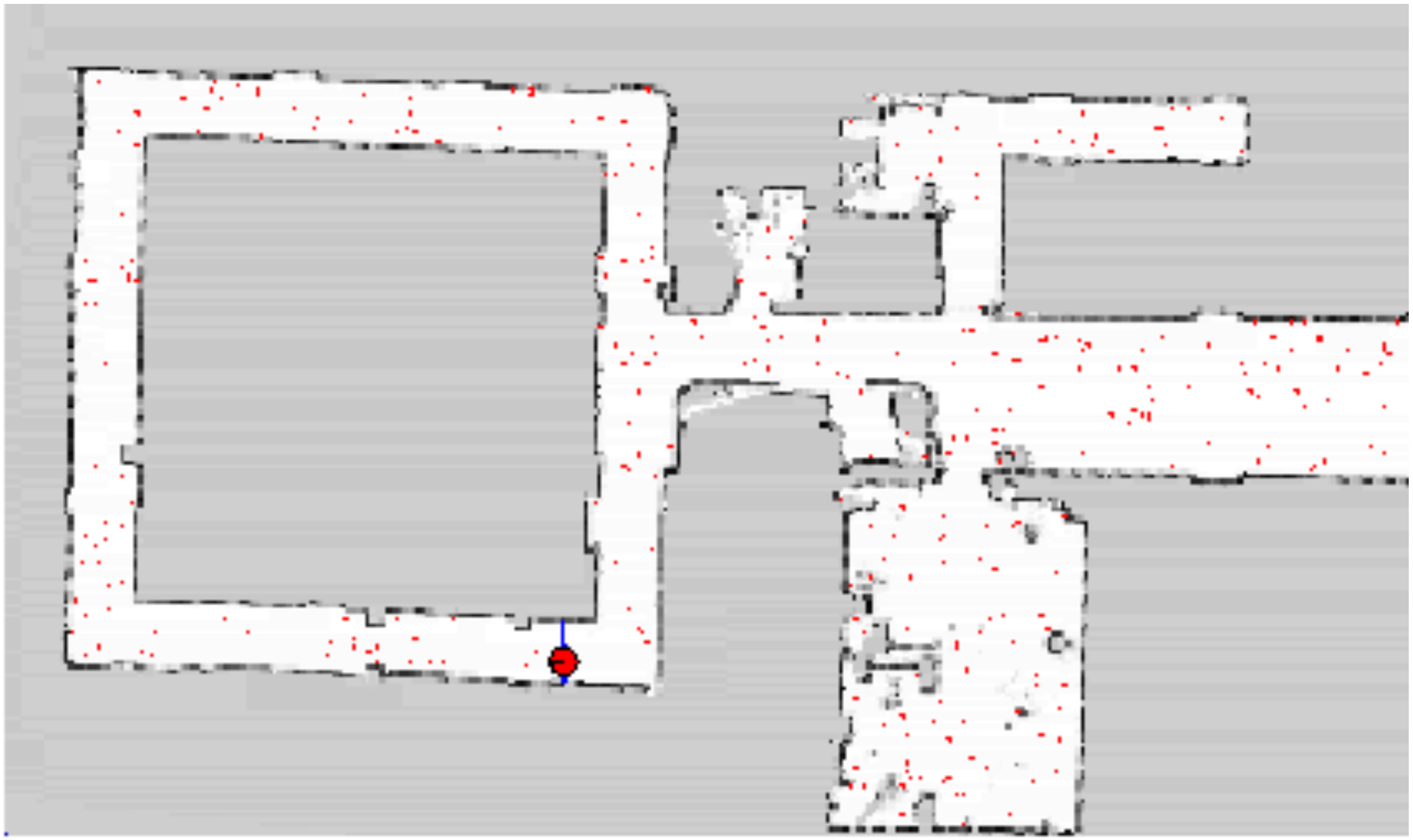
$$P(\text{robot @ location}) = \text{probability of virtual robot from last timestep}$$

$$P(\text{sensor reading}) = \sum_{x} P(\text{ robot @ location } x \mid \text{sensor reading })$$



## Particle Filter for Localization





<break>

## SLAM: Simultaneous localization and mapping

Slides adapted from USC CSCI 445: Introduction to robotics
Credit: Erdem Biyik, Heather Culbertson

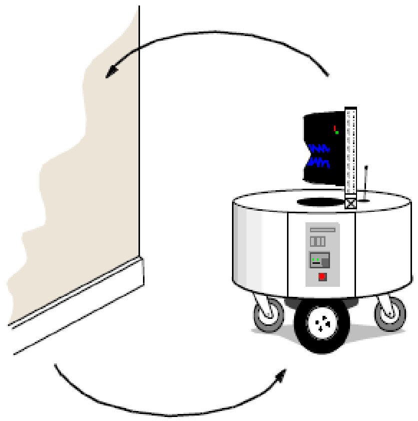
## SLAM: Simultaneous localization and mapping

- Estimate the pose of a robot and the map of the environment at the same time

- **Localization**: inferring location given a map
- **Mapping**: inferring a map given locations
- **SLAM**: learning a map and locating the robot simultaneously

## SLAM: Simultaneous localization and mapping

- SLAM is hard, because
  - a map is needed for localization and
  - a good pose estimate is needed for mapping
  - (a chicken-and-egg problem)



## SLAM applications

- SLAM is central to a range of indoor, outdoor, in-air and underwater applications for both manned and autonomous vehicles.

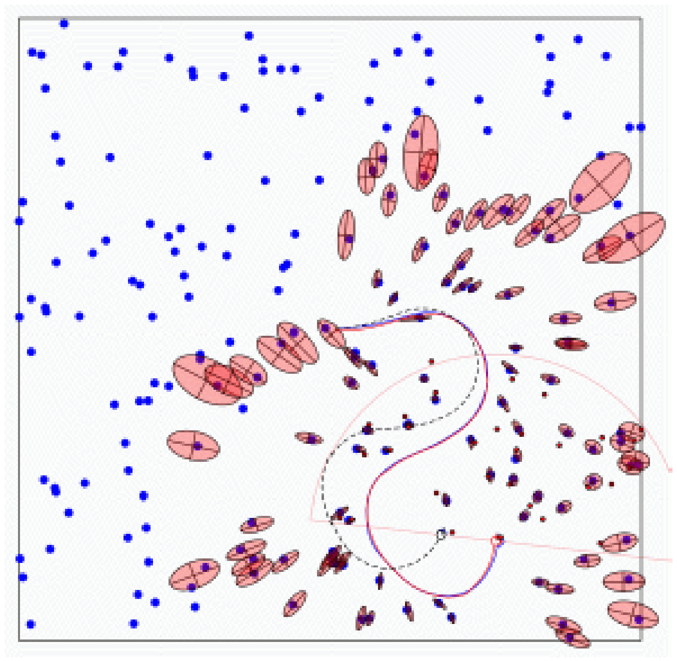
- Examples:
  - At home: vacuum cleaner, lawn mower
  - Air: surveillance with unmanned air vehicles
  - Underwater: reef monitoring
  - Underground: exploration of mines
  - Space: terrain mapping for localization

## The SLAM problem

- SLAM is considered a fundamental problem for robots to become truly autonomous
- Large variety of different SLAM approaches have been developed
- The majority uses probabilistic concepts
- History of SLAM dates to the mid-eighties

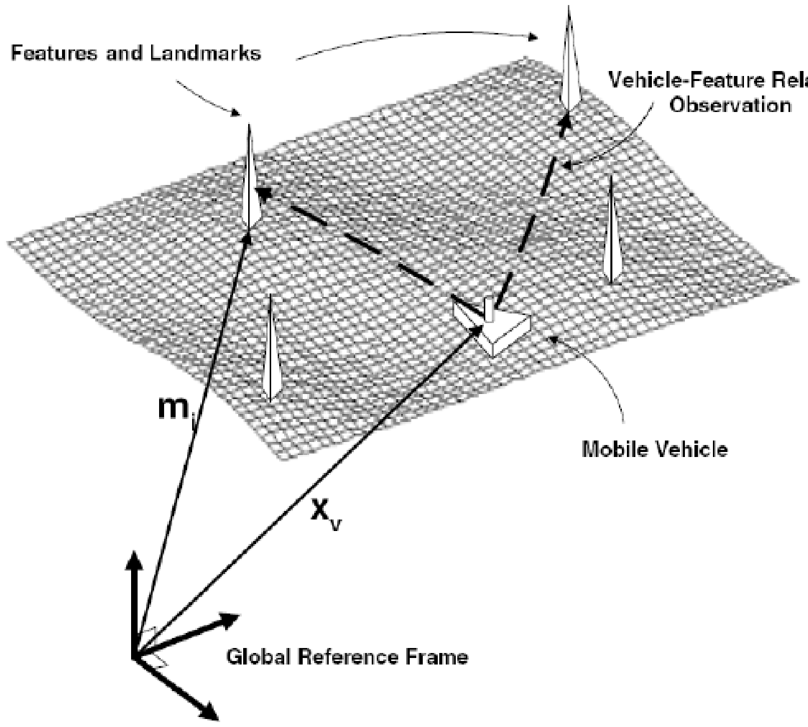
## Feature-based SLAM

- Given:
  - The robot's controls $\boldsymbol{U}_{1:k} = \{\boldsymbol{u}_1, \boldsymbol{u}_2, \ldots, \boldsymbol{u}_k\}$
  - Relative observations $\boldsymbol{Z}_{1:k} = \{\boldsymbol{z}_1, \boldsymbol{z}_2, \ldots, \boldsymbol{z}_k\}$
- Wanted
  - Map of features $\boldsymbol{m} = \{\boldsymbol{m}_1, \boldsymbol{m}_2, \ldots, \boldsymbol{m}_n\}$
  - Path of the robot $\boldsymbol{X}_{1:k} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_k\}$



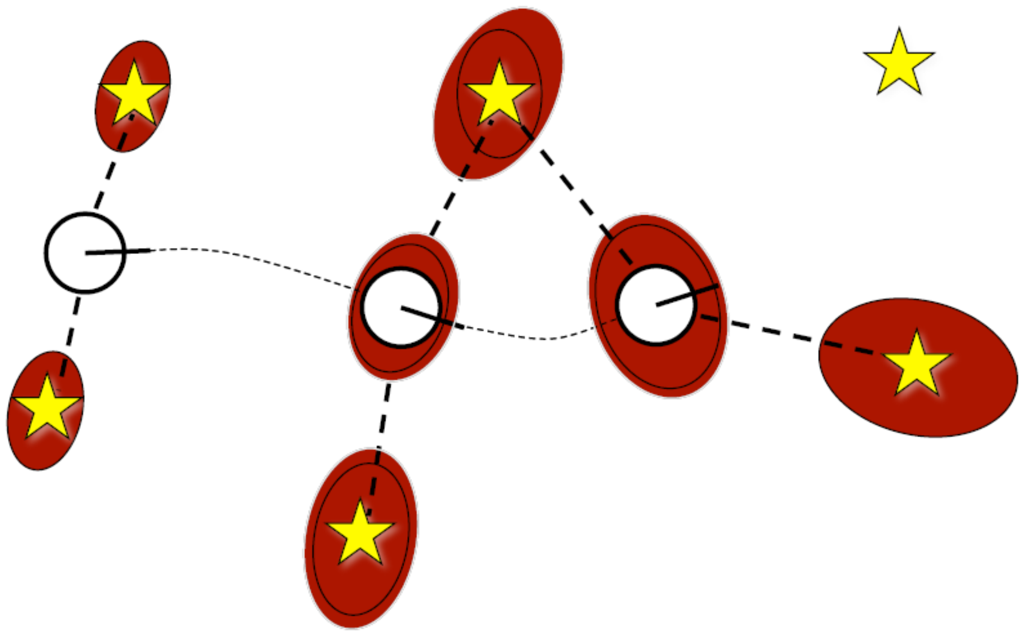
## Feature-based SLAM

- **Absolute** robot poses
- **Absolute** landmark positions
- But only **relative** measurements of landmarks



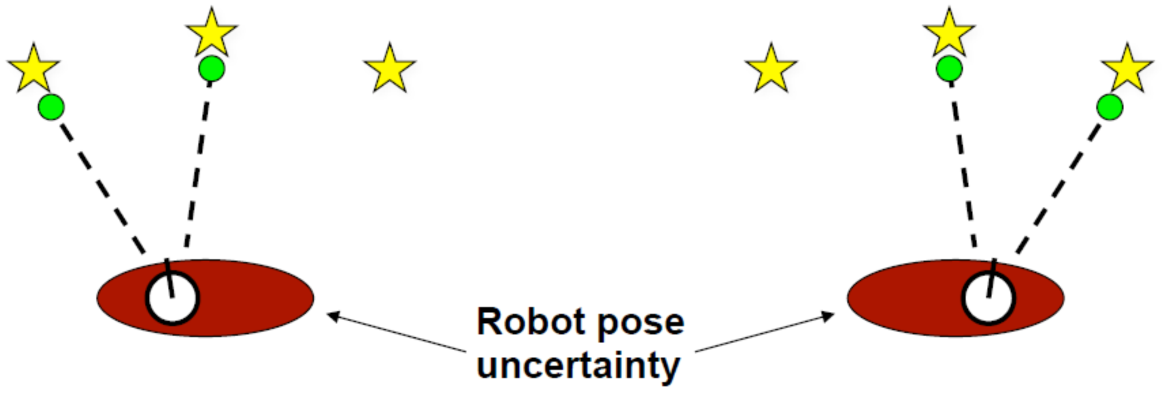
## Why is SLAM a hard problem?

- 1. Robot path and map are both unknown
- 2. Errors in map and pose estimates correlated



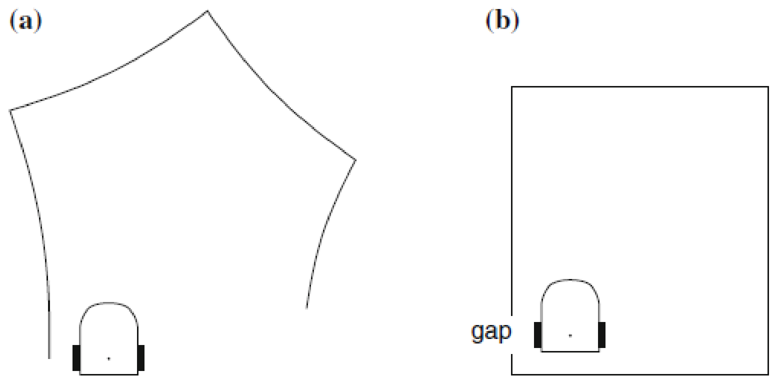
## Why is SLAM a hard problem?

- The mapping between observations and landmarks is unknown
- Picking wrong data associations can have catastrophic consequences (divergence)



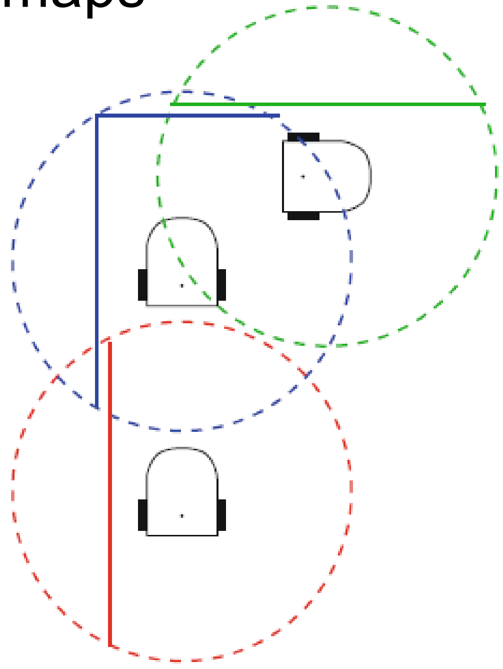
## Knowledge about environment

- Cannot rely on odometry alone
- Need knowledge about environment or its structure
  - i.e., walls are straight, corners are 90°
- Can be used to close the loop



## Overlapping sensor measurements

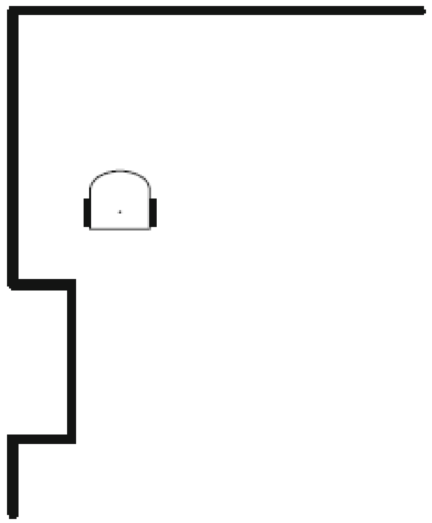
- Regular features, seen from multiple angles at different points in time
- Can be used to update localization and correct maps
- Basis of SLAM algorithm



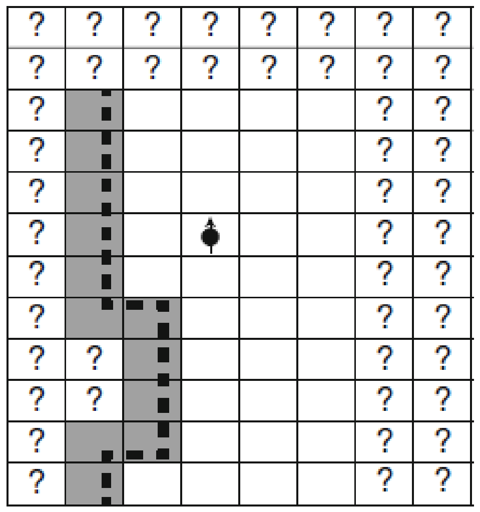
## SLAM - Derivation

- Divide environment into a grid
  - Each cell is labeled as free or obstacle
  - Unknown/unexplored cells shown as ?

(a)

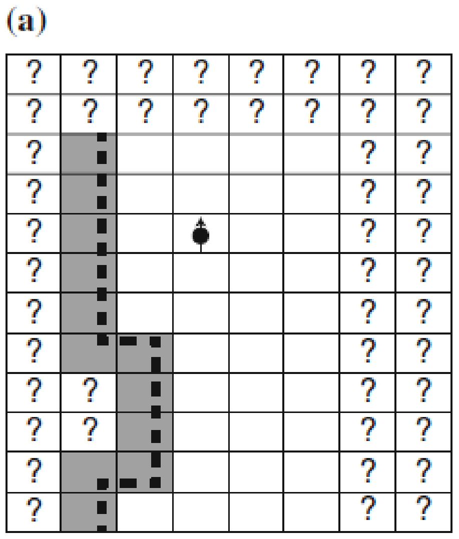


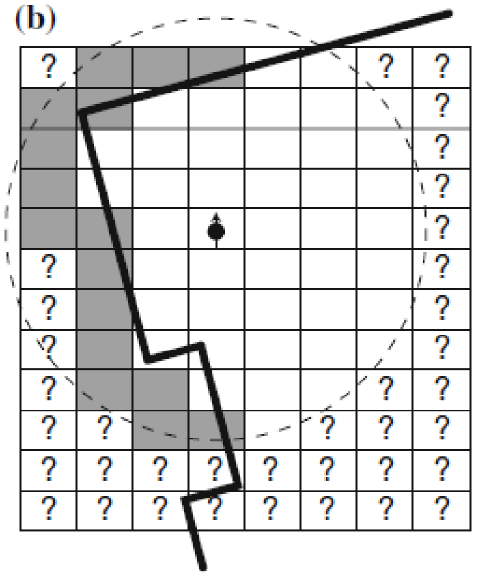
(b)



## SLAM - Derivation

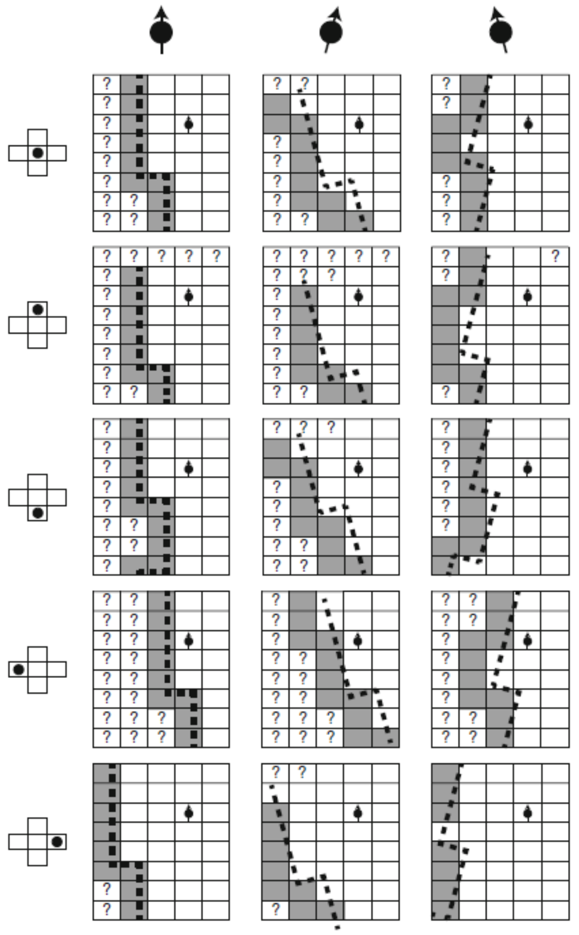
- Inconsistencies in actuators and effectors mean that the robot does not perfectly execute each motion command
- (a) is intended perception
- (b) is actual





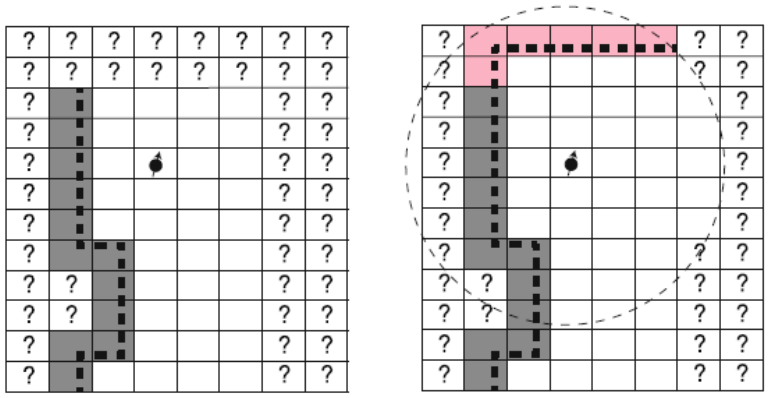
## SLAM - Derivation

- Assume odometry gives a reasonable estimation of pose (position and heading) for short motions
- For each relatively small possible error in pose, compute what the perception of the current map would be and compare it with the actual perception computed from sensor data
    - Choose pose that gives best match
    - Set this as actual pose of robot and update map



## SLAM - Derivation

- Correct the pose of the robot
- Use data from perception map to update current map stored in the robot's memory



## SLAM Algorithm

**Algorithm 9.2: SLAM**

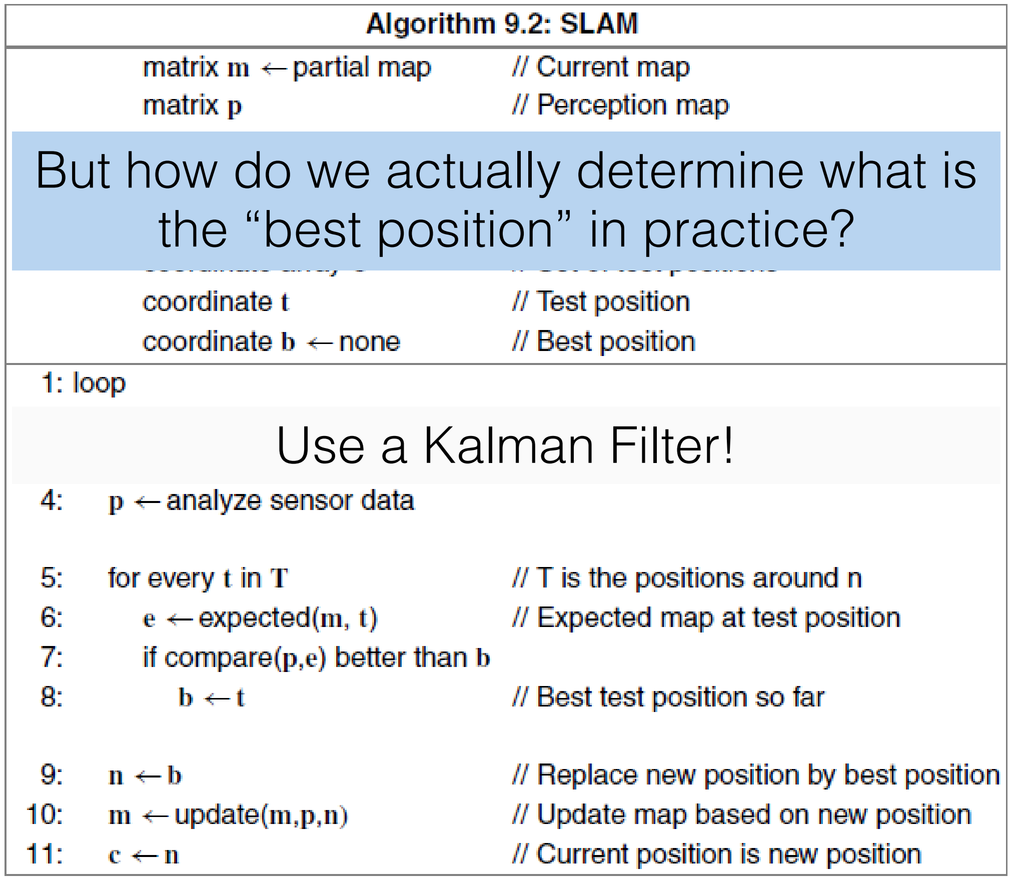
```
       matrix m ← partial map            // Current map
       matrix p                          // Perception map
       matrix e                          // Expected map
       coordinate c ← initial position   // Current position
       coordinate n                      // New position
       coordinate array T                // Set of test positions
       coordinate t                      // Test position
       coordinate b ← none               // Best position

 1: loop
 2:    move a short distance
 3:    n ← odometry(c)                   // New position based on odometry
 4:    p ← analyze sensor data

 5:    for every t in T                  // T is the positions around n
 6:       e ← expected(m, t)             // Expected map at test position
 7:       if compare(p,e) better than b
 8:          b ← t                       // Best test position so far

 9:    n ← b                             // Replace new position by best position
10:    m ← update(m,p,n)                 // Update map based on new position
11:    c ← n                             // Current position is new position
```

## SLAM Algorithm



**Algorithm 9.2: SLAM**

```
        matrix m ← partial map            // Current map
        matrix p                          // Perception map
```

But how do we actually determine what is the “best position” in practice?

```
        coordinate t                      // Test position
        coordinate b ← none               // Best position
1: loop
```

Use a Kalman Filter!

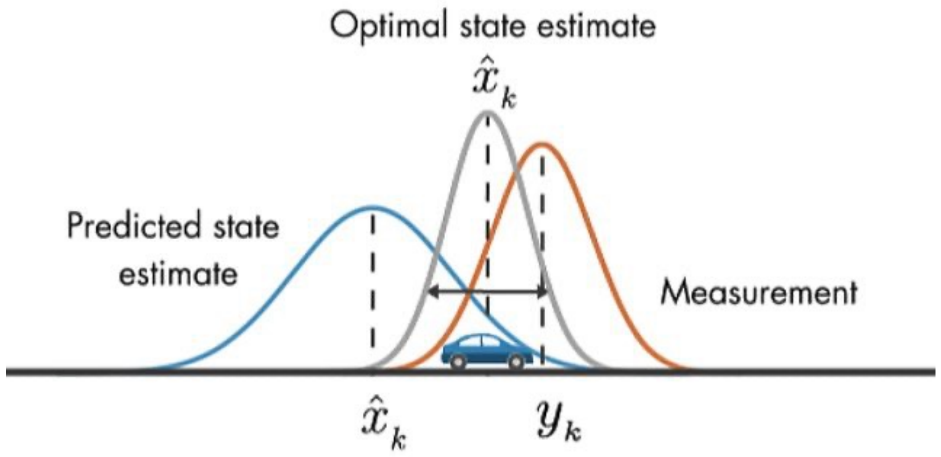
```
4:    p ← analyze sensor data

5:    for every t in T                    // T is the positions around n
6:        e ← expected(m, t)              // Expected map at test position
7:        if compare(p,e) better than b
8:            b ← t                       // Best test position so far

9:    n ← b                               // Replace new position by best position
10:   m ← update(m,p,n)                   // Update map based on new position
11:   c ← n                               // Current position is new position
```

## Kalman filter

- Also known as Linear Quadratic Estimation (LQE)
- assumes **Gaussian noise** in both motion and measurements and uses a **linearized approximation** of the system dynamics
- Uses a series of measurements observed over time
- (Follows similar process to particle filter but is not the same!)



| Feature | **Kalman Filter** | **Particle Filter** |
|---|---|---|
| Assumes Gaussian noise | ✅ Yes | ❌ Not required |
| Works with nonlinear systems | ❌ Only with extended/unscented variants (EKF/UKF) | ✅ Naturally handles nonlinear systems |
| State representation | A single mean and covariance matrix | A set of particles (samples of possible states) |
| Scalability | Very efficient for low-dimensional states | Can handle high-dimensional states (but needs many particles) |
| Accuracy | Accurate if assumptions hold (e.g., linearity, Gaussian) | More accurate in complex or non-Gaussian cases |
| Computational load | Low to moderate | Higher (especially with many particles) |

## Extended Kalman Filter (EKF) SLAM

- Localization
  - 3x1 pose vector
  - 3x3 covariance matrix

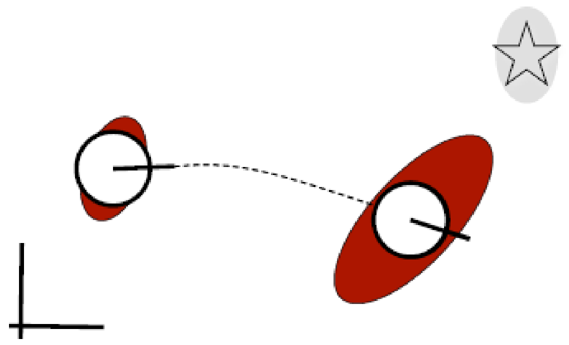
$$\mathbf{x}_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \end{bmatrix} \qquad C_k = \begin{bmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} \\ \sigma_{yx} & \sigma_y^2 & \sigma_{y\theta} \\ \sigma_{\theta x} & \sigma_{\theta y} & \sigma_\theta^2 \end{bmatrix}$$

- SLAM
  - Landmarks simply extend the state. Growing both the state vector and covariance matrix.
  - Ex: Map with *N* landmarks

$$Bel(x_t, m_t) = \left\langle \begin{pmatrix} x \\ y \\ \theta \\ l_1 \\ l_2 \\ \vdots \\ l_N \end{pmatrix}, \begin{pmatrix} \sigma_x^2 & \sigma_{xy} & \sigma_{x\theta} & \sigma_{xl_1} & \sigma_{xl_2} & \cdots & \sigma_{xl_N} \\ \sigma_{xy} & \sigma_y^2 & \sigma_{y\theta} & \sigma_{yl_1} & \sigma_{yl_2} & \cdots & \sigma_{yl_N} \\ \sigma_{x\theta} & \sigma_{y\theta} & \sigma_\theta^2 & \sigma_{\theta l_1} & \sigma_{\theta l_2} & \cdots & \sigma_{\theta l_N} \\ \sigma_{xl_1} & \sigma_{yl_1} & \sigma_{\theta l_1} & \sigma_{l_1}^2 & \sigma_{l_1 l_2} & \cdots & \sigma_{l_1 l_N} \\ \sigma_{xl_2} & \sigma_{yl_2} & \sigma_{\theta l_2} & \sigma_{l_1 l_2} & \sigma_{l_2}^2 & \cdots & \sigma_{l_2 l_N} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ \sigma_{xl_N} & \sigma_{yl_N} & \sigma_{\theta l_N} & \sigma_{l_1 l_N} & \sigma_{l_2 l_N} & \cdots & \sigma_{l_N}^2 \end{pmatrix} \right\rangle$$

## EKF SLAM: Building the Map

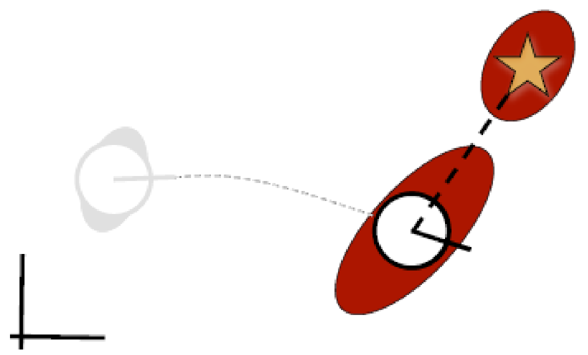
- Filter cycle:
  1. **Predict robot's state** (odometry)



$$\mathbf{x}_k = \begin{bmatrix} \mathbf{x}_R \\ \mathbf{m}_1 \\ \mathbf{m}_2 \\ \vdots \\ \mathbf{m}_n \end{bmatrix}_k \qquad C_k = \begin{bmatrix} C_R & C_{RM_1} & C_{RM_2} & \cdots & C_{RM_n} \\ C_{M_1R} & C_{M_1} & C_{M_1M_2} & \cdots & C_{M_1M_n} \\ C_{M_2R} & C_{M_2M_1} & C_{M_2} & \cdots & C_{M_2M_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{M_nR} & C_{M_nM_1} & C_{M_nM_2} & \cdots & C_{M_n} \end{bmatrix}_k$$

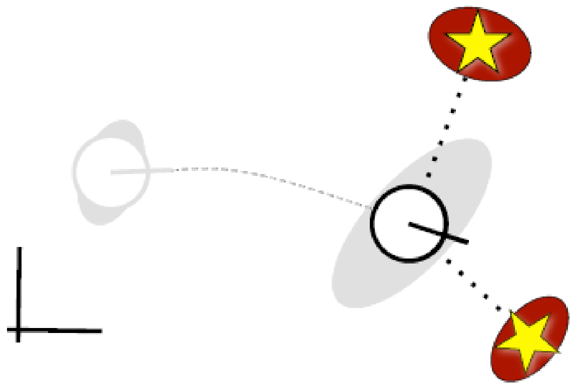
## EKF SLAM: Building the Map

- Filter cycle:
  1. **Predict robot's state** (odometry)
  2. **Measurement prediction** (predict what the robot should see based on landmarks that currently exist in map)



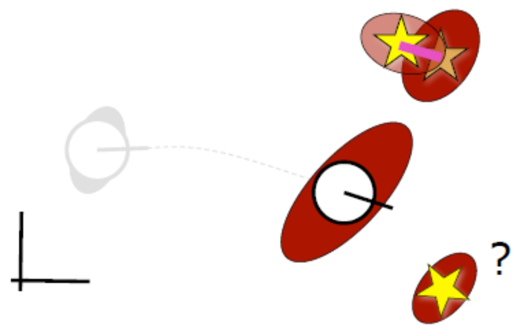
## EKF SLAM: Building the Map

- Filter cycle:
  1. **Predict robot's state** (odometry)
  2. **Measurement prediction** (predict what you the robot should see based on landmarks that currently exist in map)
  3. **Observation** (measure actual distance and orientation to landmarks)



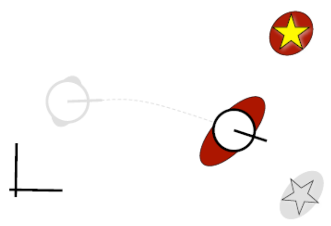
## EKF SLAM: Building the Map

- Filter cycle:
    1. **Predict robot's state** (odometry)
    2. **Measurement prediction** (predict what you the robot should see based on landmarks that currently exist in map)
    3. **Observation** (measure actual distance and orientation to landmarks)
    4. **Data association** (associates predicted measurements to observations)



## EKF SLAM: Building the Map

- Filter cycle:
  1. **Predict robot’s state** (odometry)
  2. **Measurement prediction** (predict what you the robot should see based on landmarks that currently exist in map)
  3. **Observation** (measure actual distance and orientation to landmarks)
  4. **Data association** (associates predicted measurements to observations)
  5. **Update** (update pose vector and covariance matrix using Kalman Filter)

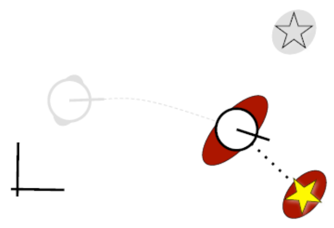


$$\mathbf{x}_k = \begin{bmatrix} \mathbf{x}_R \\ \mathbf{m}_1 \\ \mathbf{m}_2 \\ \vdots \\ \mathbf{m}_n \end{bmatrix}_k \qquad C_k = \begin{bmatrix} C_R & C_{RM_1} & C_{RM_2} & \cdots & C_{RM_n} \\ C_{M_1R} & C_{M_1} & C_{M_1M_2} & \cdots & C_{M_1M_n} \\ C_{M_2R} & C_{M_2M_1} & C_{M_2} & \cdots & C_{M_2M_n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{M_nR} & C_{M_nM_1} & C_{M_nM_2} & \cdots & C_{M_n} \end{bmatrix}_k$$

## EKF SLAM: Building the Map

- Filter cycle:
  1. **Predict robot's state** (odometry)
  2. **Measurement prediction** (predict what you the robot should see based on landmarks that currently exist in map)
  3. **Observation** (measure actual distance and orientation to landmarks)
  4. **Data association** (associates predicted measurements to observations)
  5. **Update** (update pose vector and covariance matrix using Kalman Filter)
  6. **Integration of new landmarks** (extend pose vector and covariance matrix with new landmarks)

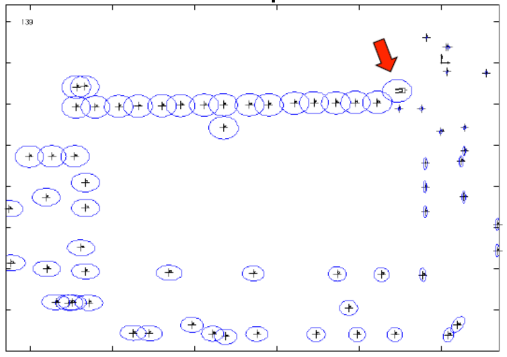
$$\mathbf{x}_k = \begin{bmatrix} \mathbf{x}_R \\ \mathbf{m}_1 \\ \mathbf{m}_2 \\ \vdots \\ \mathbf{m}_n \\ \mathbf{m}_{n+1} \end{bmatrix}_k \quad C_k = \begin{bmatrix} C_R & C_{RM_1} & C_{RM_2} & \cdots & C_{RM_n} & C_{RM_{n+1}} \\ C_{M_1R} & C_{M_1} & C_{M_1M_2} & \cdots & C_{M_1M_n} & C_{M_1M_{n+1}} \\ C_{M_2R} & C_{M_2M_1} & C_{M_2} & \cdots & C_{M_2M_n} & C_{M_2M_{n+1}} \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{M_nR} & C_{M_nM_1} & C_{M_nM_2} & \cdots & C_{M_n} & C_{M_nM_{n+1}} \\ C_{M_{n+1}R} & C_{M_{n+1}M_1} & C_{M_{n+1}M_2} & \cdots & C_{M_{n+1}M_n} & C_{M_{n+1}} \end{bmatrix}_k$$



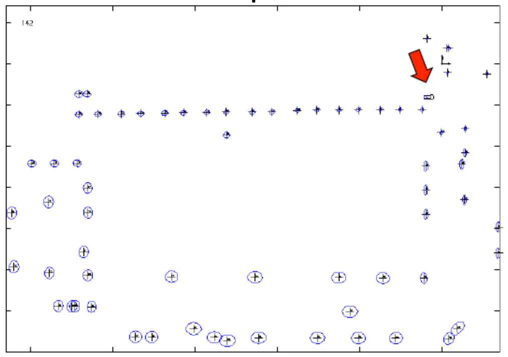
## Loop Closure

- Recognizing an already mapped area, typically after a long exploration path (the robot "closes a loop")
- Structurally identical to data association, but
  - high levels of ambiguity
  - possibly useless validation gates
  - environment symmetries
- Uncertainties collapse after a loop closure (whether the closure was correct or not)

Before loop closure

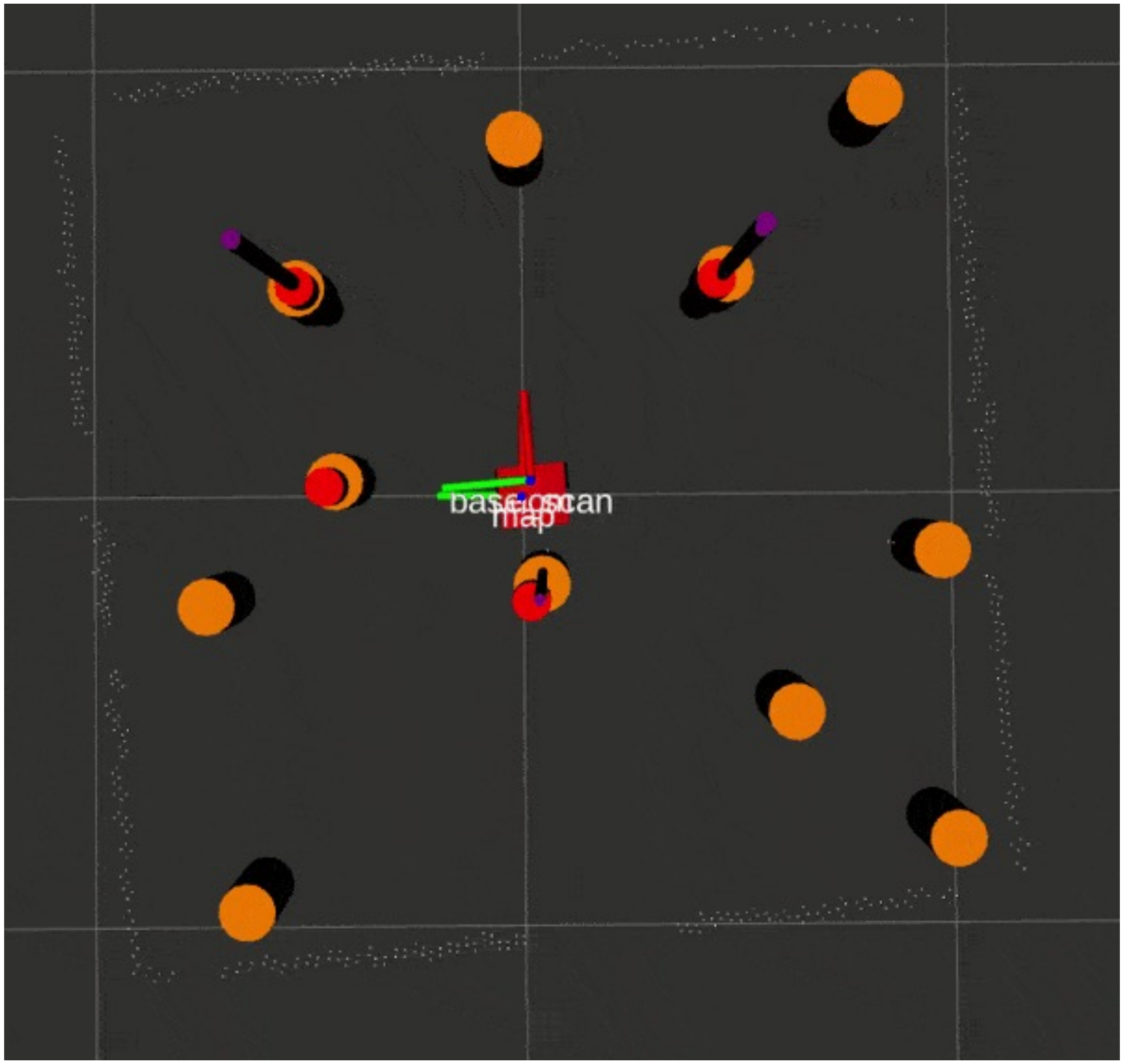


After loop closure



## Loop closure

- By revisiting already mapped areas, uncertainties in robot and landmark estimates can be reduced
- This can be exploited when exploring an environment for the sake of better (e.g. more accurate) maps
- Exploration: the problem of where to acquire new information


base_scan
map

## EKF-SLAM: Complexity

- Cost per step: quadratic in n, the number of landmarks: $O(n^2)$
- Total cost to build a map with n landmarks: $O(n^3)$
- Memory consumption: $O(n^2)$
- Problem: becomes computationally intractable for large maps!
- There exists variants to circumvent these problems

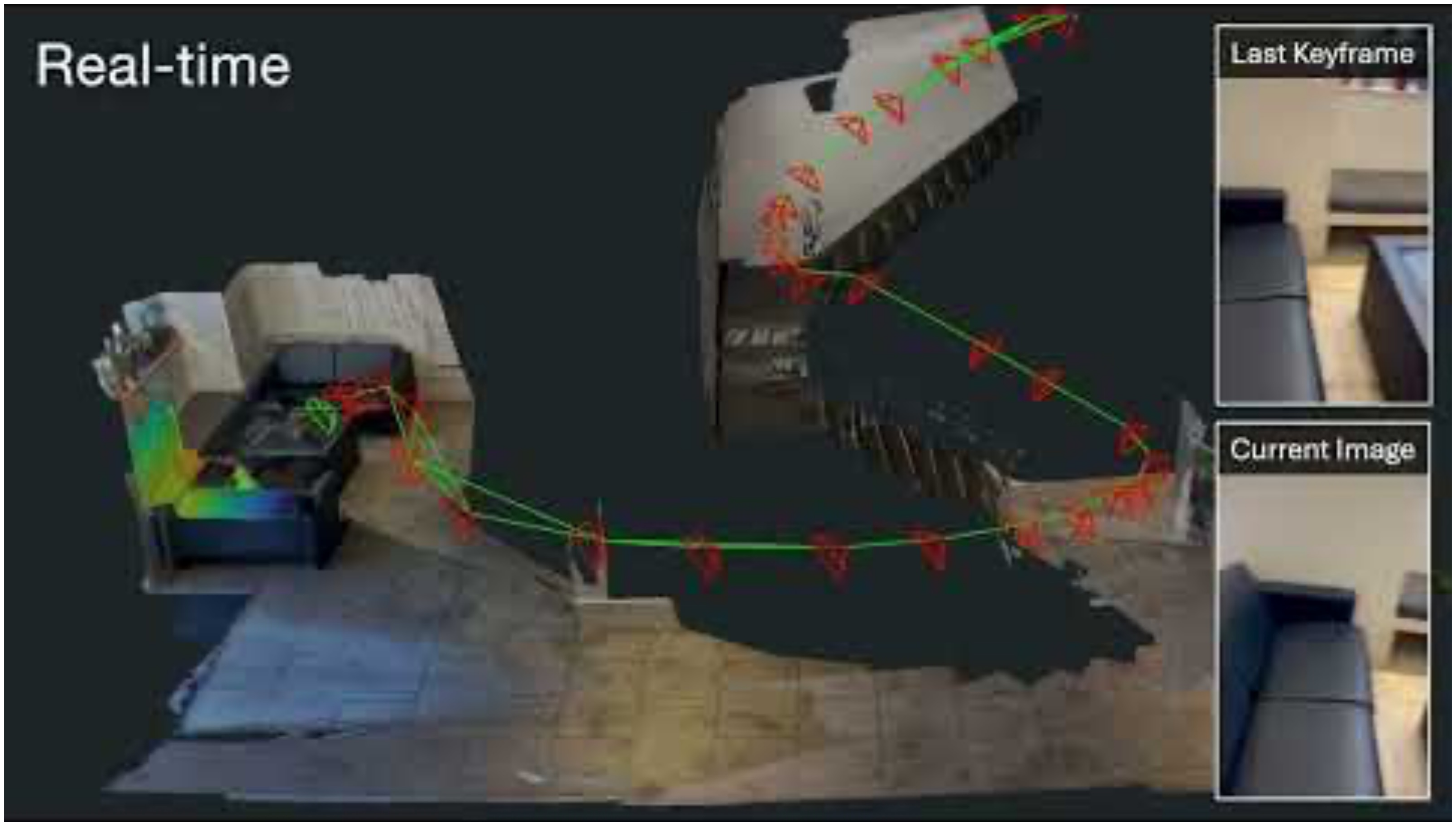
## SLAM Techniques

- EKF SLAM
- FastSLAM (particle filter version!)
- Graph-based SLAM
- Topological SLAM (mainly place recognition)
- Scan Matching / Visual Odometry (only locally consistent maps)
- Approximations for SLAM: Local submaps,
- Sparse extended information filters, Sparse links, Thin junction tree filters, etc.
- …

## MASt3R-SLAM: Real-Time Dense SLAM with 3D Reconstruction Priors

CVPR 2025

Riku Murai*Eric Dexheimer*Andrew J. Davison
Imperial College London



That’s it for today!

Questions?