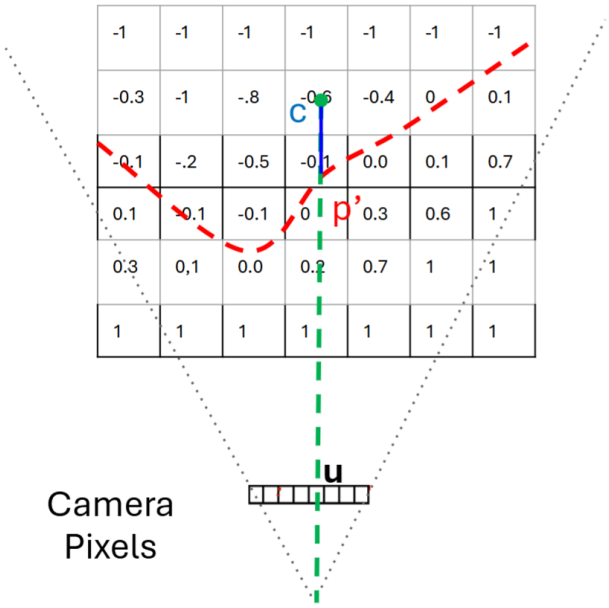
CS 188 Robotics Week 4

Aidan Jan

April 24, 2025

Projective Sign Distance Function



Sign indicated occluded / free space with respect to the camera

In this example:

- Red: surface observed in camera
- Green: camera projection ray
- We take p' as the nearest point.
- Sign: c is behind p' $\rightarrow$ occluded $\rightarrow$ negative
- Projective distance: blue line
- 3D point location $c\langle x, y, z\rangle$
- Project $c\langle x, y, z\rangle$ to the 2D image, gives us 2D coordinate u, v.
- Depth reading on pixel is $d(u, v)$

Signed projective distance:

$$d_{\text{proj}} = d(u, v) - z$$

State Estimation and Particle Filter

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

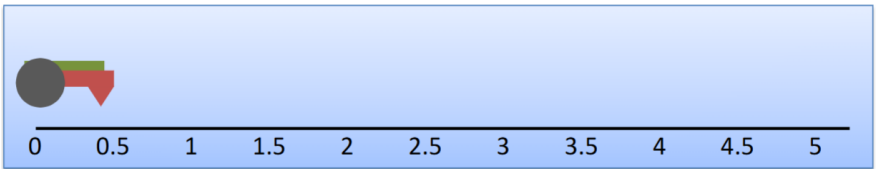
- We will use probabilistic representations for
  - The world state
  - sensor models
  - action models
- Use the calculus of probability theory to combine these models

Probabilistic Localization

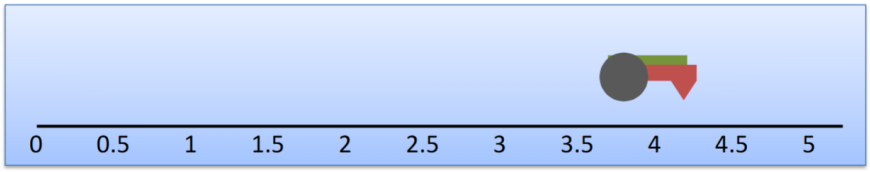
- Goal: use probabilistic methods to represent both the motion and the perception of your robots.
- We will use a "particle filter" to represent these probability distributions

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



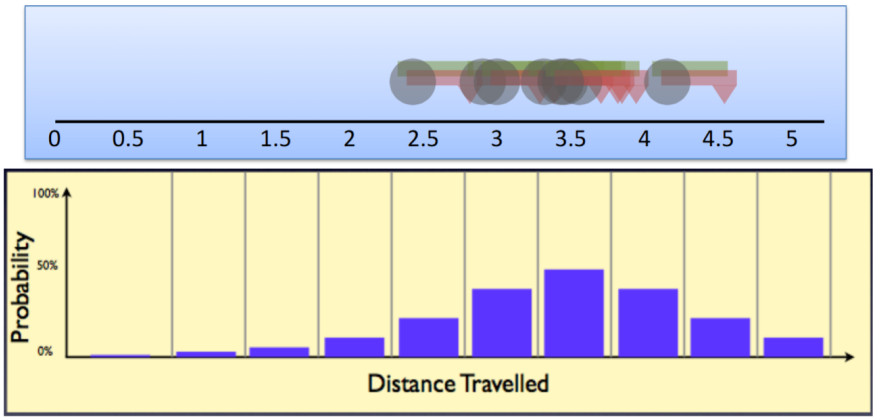
- The motors are subject to noise!



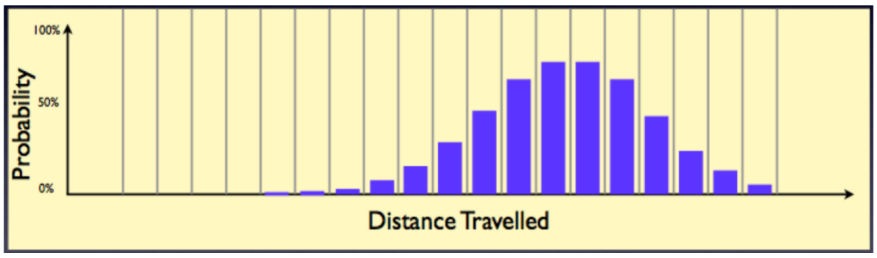
What if we were to tell our robot to move forward 3.5 feet 100 different times?

- 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.



We can do the same thing even with smaller increments

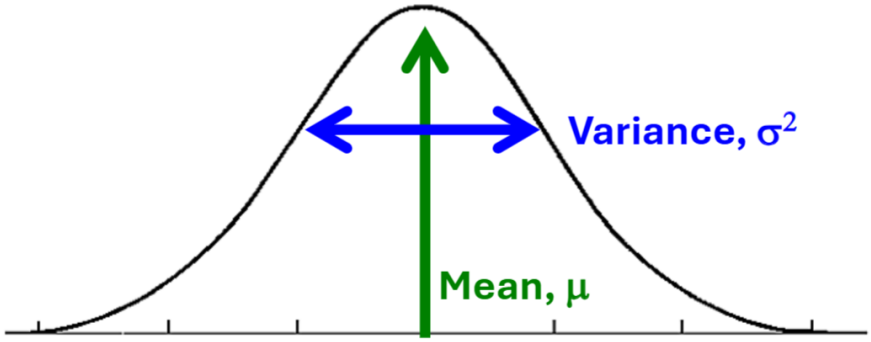


This "bell curve" shape is very common when describing noisy processes

- It is called the "Gaussian" or "Normal" distribution

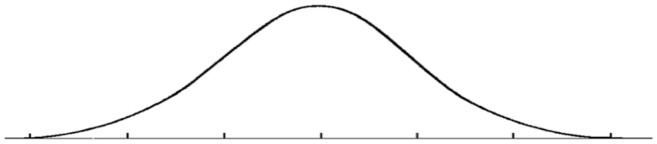
Gaussian Distribution

- 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 μ and variance σ^2

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

- 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

Θ = Desired Movement Direction
d = Desired Movement Distance
(x_t, y_t) = Current Robot Position

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

$\mathcal{N}(\mu, \sigma)$ = Normal Random Variable Sample
σ_Θ^2 = Direction Variance
σ_d^2 = Distance Variance

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

model

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

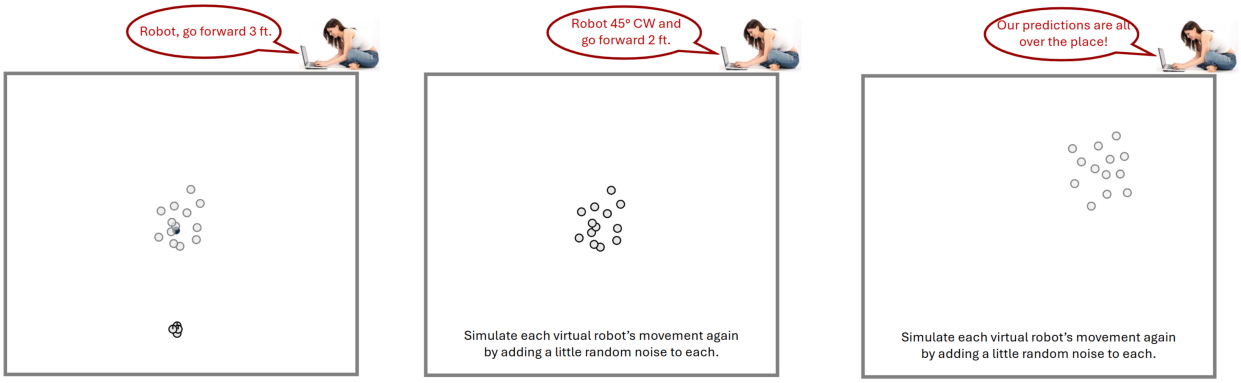
propagate

- This is the simplesst way to predict our robot's motion
- There are much more complex and accurate models available, but we will use this one for simplicity.

Example 2:

- 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.

Example 3:



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 assess the likelihood of each "virtual robot?"
- By taking measurements!

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

$$P(\text{virtual robot}) = P(\text{robot at location}|\text{sensor reading})$$

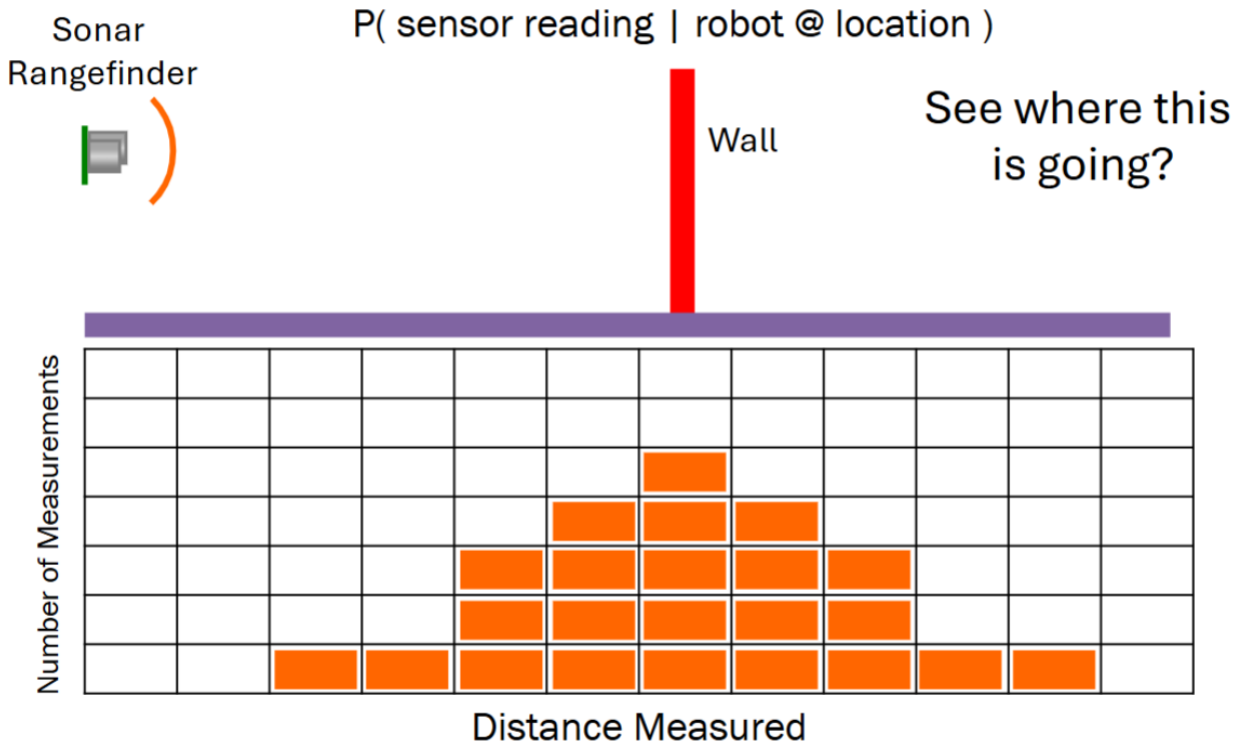
How do we calculate this?

- Bayes Law!

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

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

- Hence, this is what the $P(\text{sensor reading}|\text{robot at location})$ means.



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

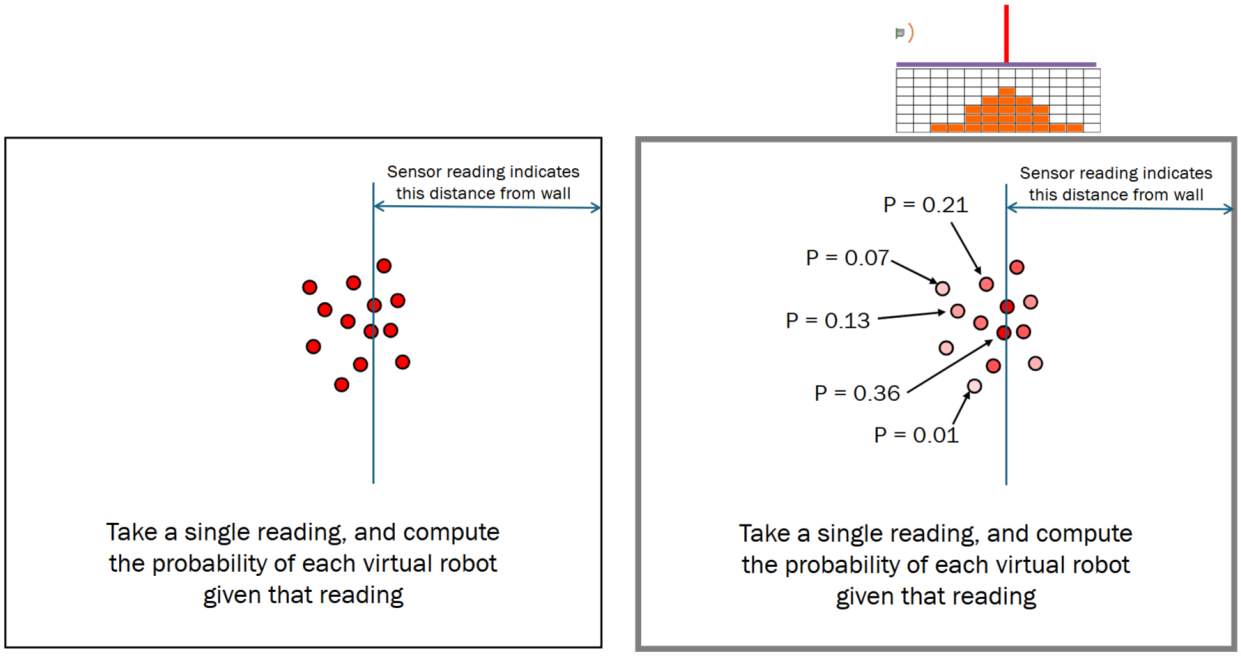
- If at each timestep, we calculate the probability of each virtual robot, then we can use those probabilities from the last timestep for the $P(\text{robot at location})$

Now all we have left is $P(\text{sensor reading})$. 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.

$$N = \frac{1}{\sum_x P(\text{sensor reading}|\text{robot at location})P(\text{robot at location})}$$

We now know everything we need to calculate $P(\text{robot at location}x|\text{sensor reading})$, the "posterior probability"

Example:



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

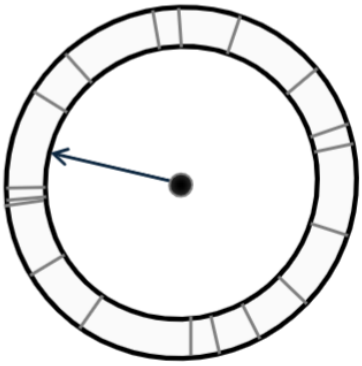
$$x_{est} = \sum_n P_n x_n$$
$$y_{est} = \sum_n P_n y_n$$

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

How does it work?

- 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 $\frac{1}{\text{numParticles}}$, 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 $P(\text{virtual robot})$

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

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:

$$N_{eff} = \frac{1}{\sum_n (\text{particle}_i \cdot \text{prob})^2}$$

- 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*

Putting it all together

Particle Filtering Algorithm:

- Create N particles at some starting location (or distributed randomly around the map), each with equal probability. Call this data structure "Particles"
  - When a new movement command (d, Q) is issued:
    - For each particle "p" in "Particles":
      - Generate a randomized movement command consisting of d' and Q'

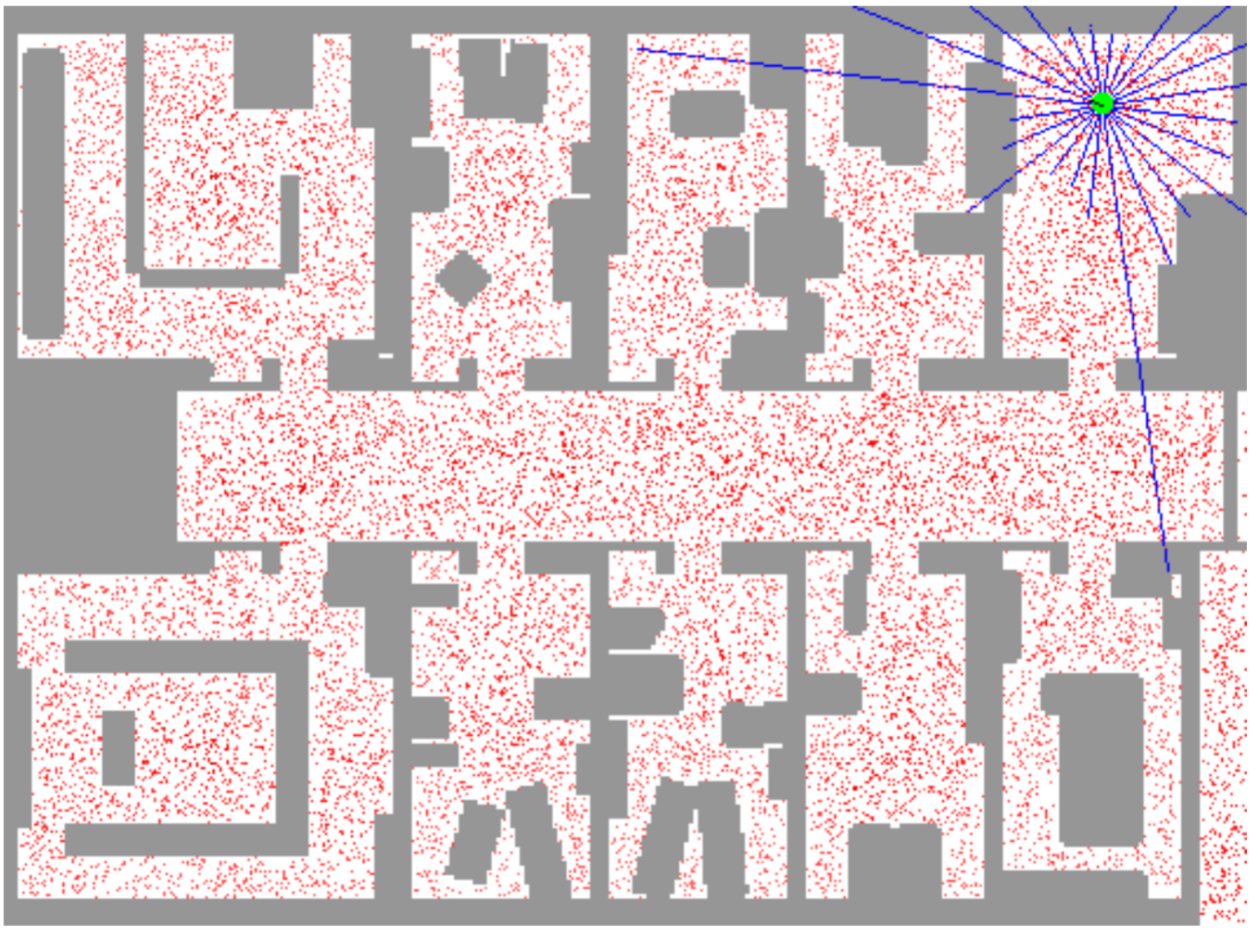
$$\theta' = \theta + \mathcal{N}(0, \sigma_\theta) \qquad d' = d + \mathcal{N}(0, \sigma_d)$$

      - Update the current position of 'p' according to the motion model applied to d' and Q'

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

      - Optionally do bounds checking to ehsure that our particles are not ghosting through walls
  - When a new sensor movement (z) is received:
    - For each particle "p" in "Particles":
      - Compute the posterior probability for each particle: P("p", location | "z")
    - Resample the particles: "Particles" = resampleParticles("Particles")

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 is hard, because

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

SLAM Applications

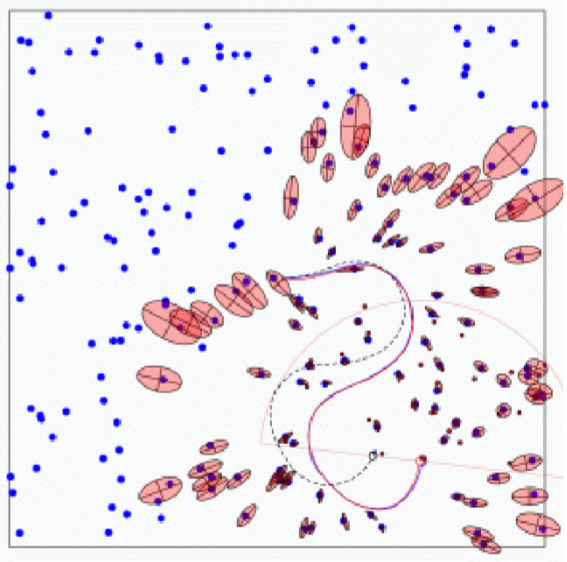
- SLAM is central to a range of indoor, outdoor, in-air, and underwater applications for both manned and autonomous vehicles.
- Examples:
  - At home: vacuum cleaner, lawn mower
  - Air: surveillance with unmanned air vehicles
  - Underwater: reef monitoring
  - Underground: exploration of mines
  - Space: terrain mapping for localization

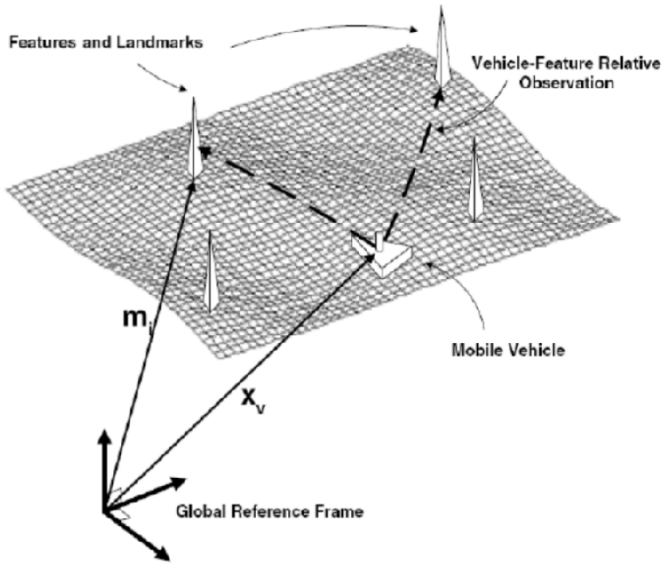
The SLAM Problem

- 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 $U_{1:k} = \{u_1, u_2, \ldots, u_k\}$
  - Relative observations $Z_{1:k} = \{z_1.z_2, \ldots, z_k\}$
- Wanted:
  - Map of features $m = \{m_1, m_2, \ldots, m_n\}$
  - Path of the robot $X_{1:k} = \{x_1, x_2, \ldots, x_k\}$

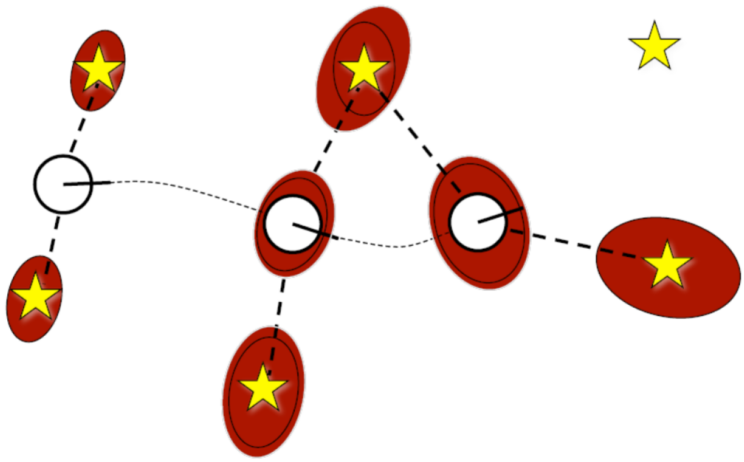




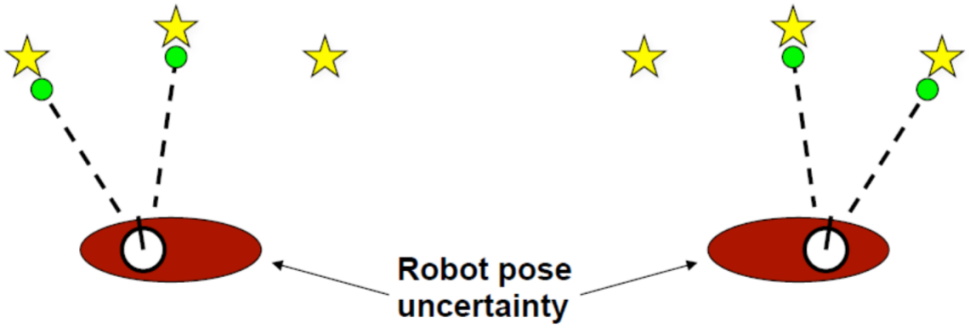
- **Absolute** robot poses
- **Absolute** landmark positions
- But only **relative** measurements of landmarks

Why is SLAM a hard problem?

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

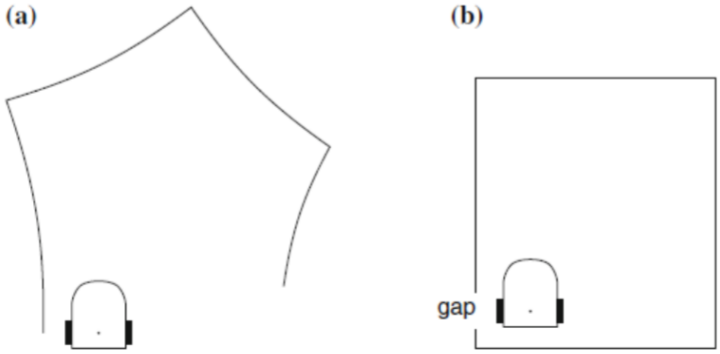


3. The mapping between observations and landmarks is unknown
4. Picking wrong data associations can have catastrophic consequences (divergence)



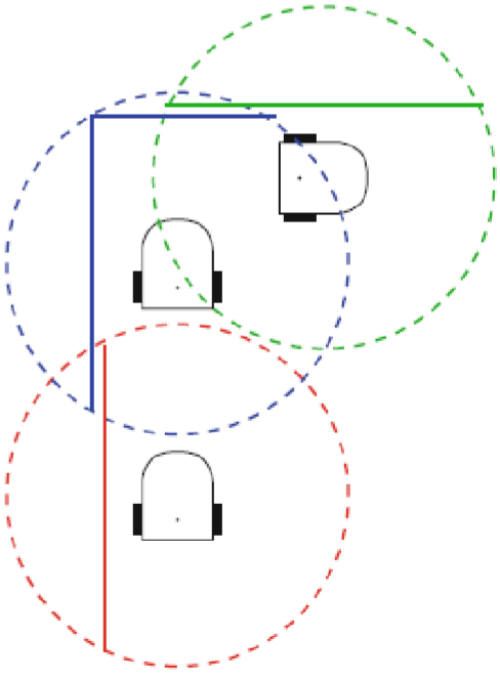
Knowledge about Environment

- 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



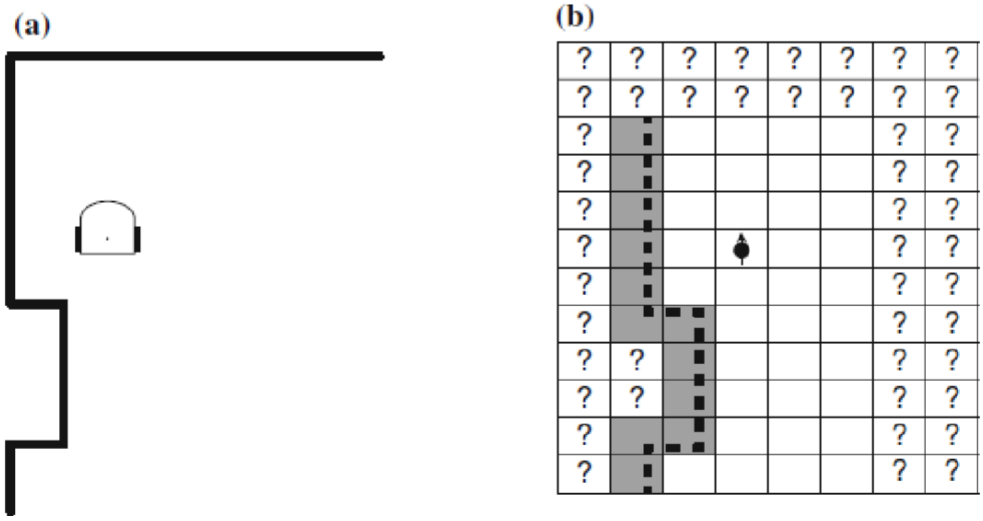
Overlapping Sensor Measurements

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

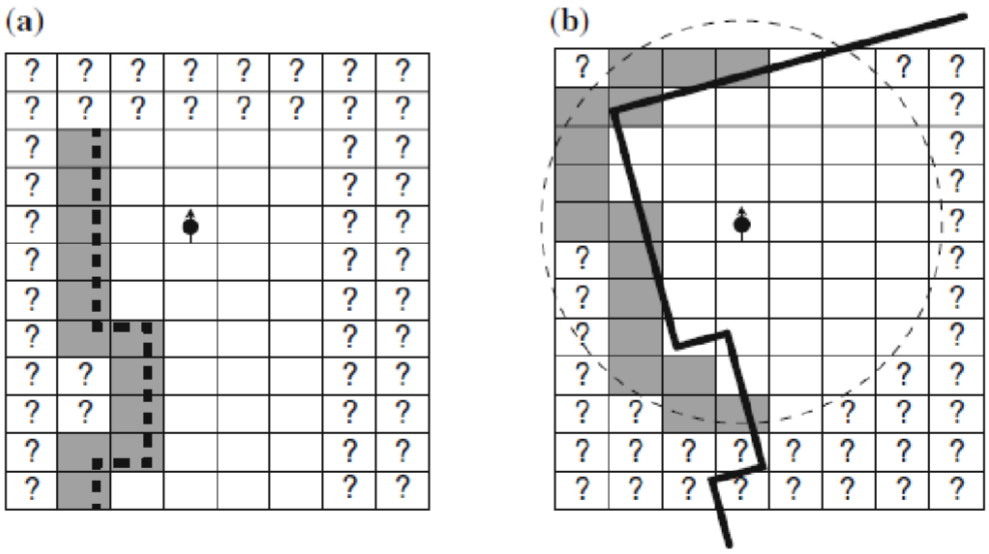


SLAM - Derivation

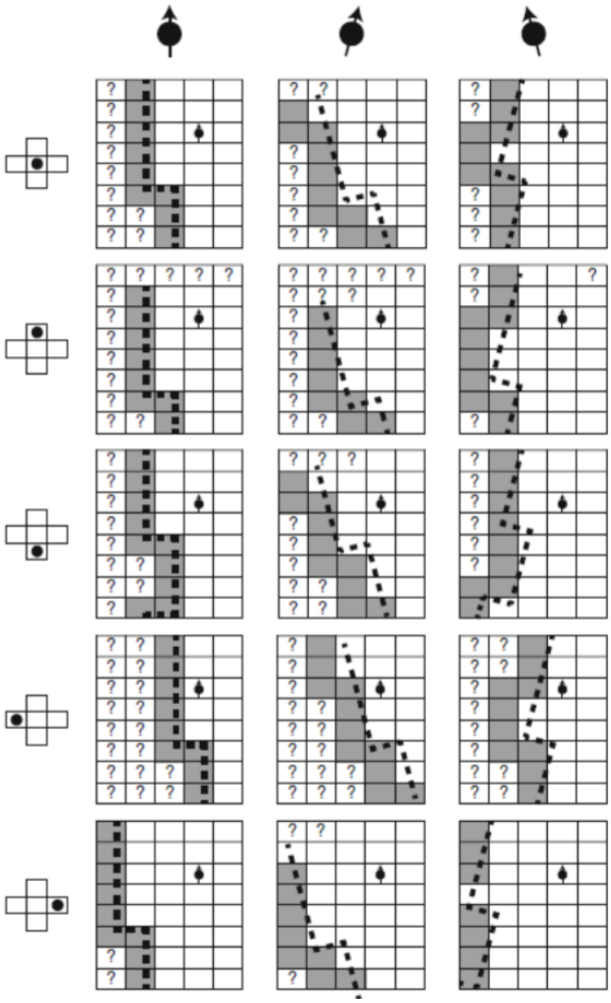
- Divide environment into a grid
  - Each cell is labeled as free or obstacle
  - Unknown / unexplored cells shown as ?



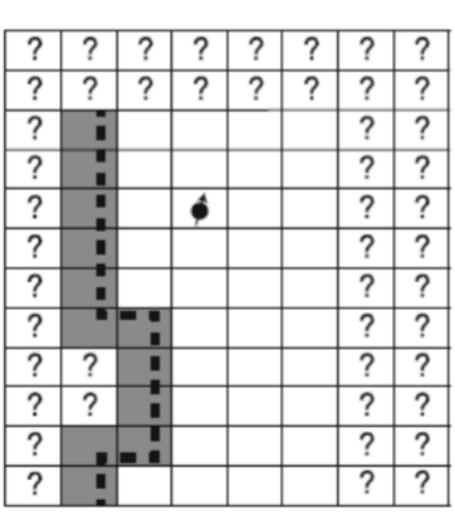
- Inconsistencies in actuators and effectors mean that the robot does not perfectly execute each motion command
- (a) is intended perception
- (b) is actual

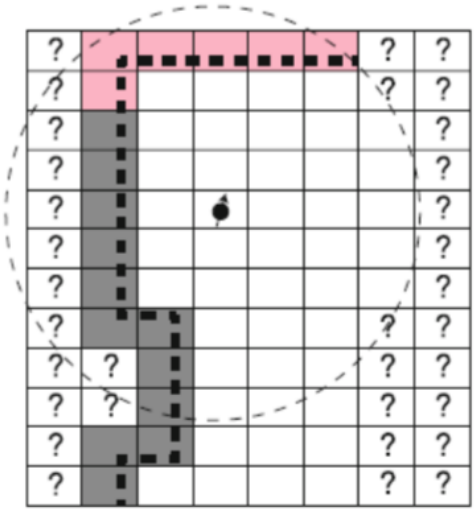


- Assume odometry gives a reasonable estimation of pose (position and heading) for short motions
- For each relatively small possible error in pose, compute what the perception of the current map would be and compare it with the current map would be and compare it with the actual perception computed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



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





SLAM Algorithm

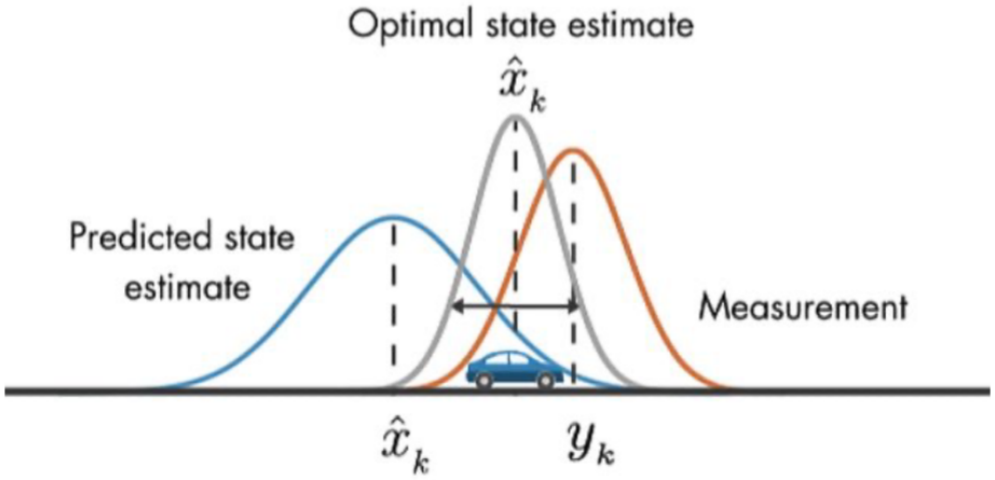
```
matrix m <- partial map           // current map
matrix p                          // perception map
matrix e                          // expected map
coordinate c <- initial position  // current position
coordinate n                      // new position
coordinate array T                // set of test positions
coordinate t                      // test position
coordinate b <- none              // best position

loop
    move a short distance
    n <- odometry(c)                    // New position based on odometry
    p <- analyze sensor data

    for every t in T                    // T is the positions around n
        e <- expected(m, t)             // Expected map at test position
        if compare(p, e) better than b
            b <- t                      // Best test position so far

    n <- b                              // Replace new position by best position
    m <- update(m, p, n)                // Update map based on new position
    c <- n                              // Current position is new position
```

How do we actually determine what is the "best position" in practice?

- Use a Kalman filter!

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

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

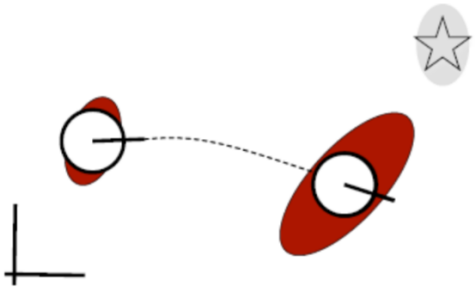
- SLAM
  - Landmarks simply extend the state. Growing both the state vector and covariance matrix.
  - Ex: Map with N landmarks

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

EKF SLAM: Building the Map

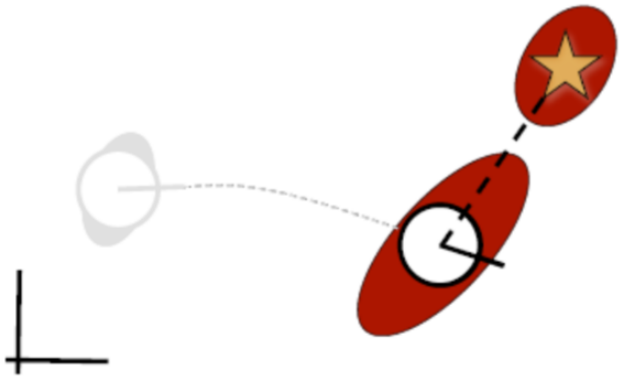
Filter cycle:

1. **Predict robot's state** (odometry)

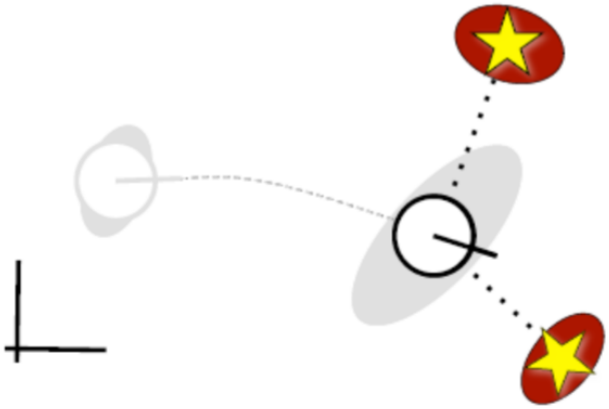


$$\mathbf{x}_k = \begin{bmatrix} \mathbf{x}_R \\ \mathbf{m}_1 \\ \mathbf{m}_2 \\ \vdots \\ \mathbf{m}_n \end{bmatrix}_k \qquad C_k = \begin{bmatrix} C_R & C_{RM_1} & C_{RM_2} & \cdots & C_{RM_n} \\ C_{M_1R} & C_{M_1} & C_{M_1M_2} & \cdots & C_{M_1M_n} \\ C_{M_2R} & C_{M_2M_1} & C_{M_2} & \cdots & C_{M_2M_n}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

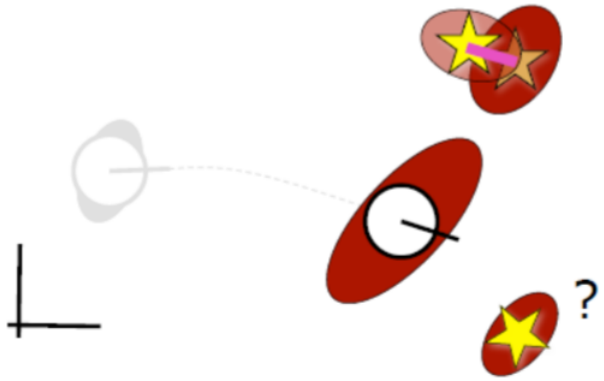
2. **Measurement prediction** (predict what the robot should see based on landmarks that currently exist in map)



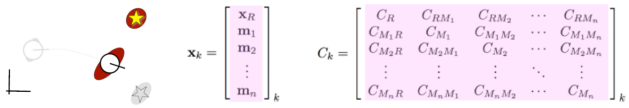
3. **Observation** (measure actual distance and orientation to landmarks)



4. **Data association** (associates predicted measurements to observations)

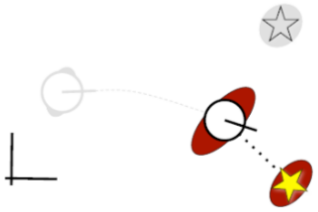


5. **Update** (update pose vector and covariance matrix using Kalman Filter)



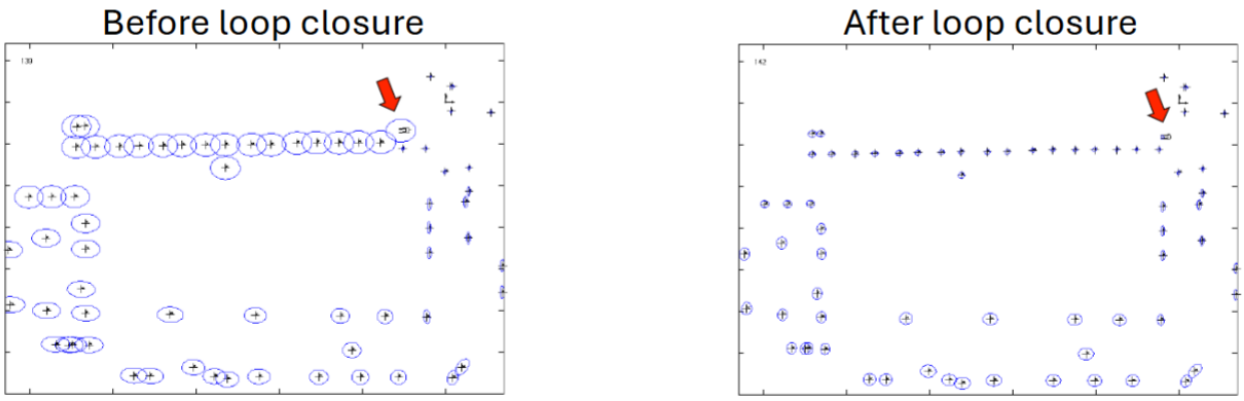
6. **Integration of new landmarks** (extend pose vector and covariance matrix with new landmarks)

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

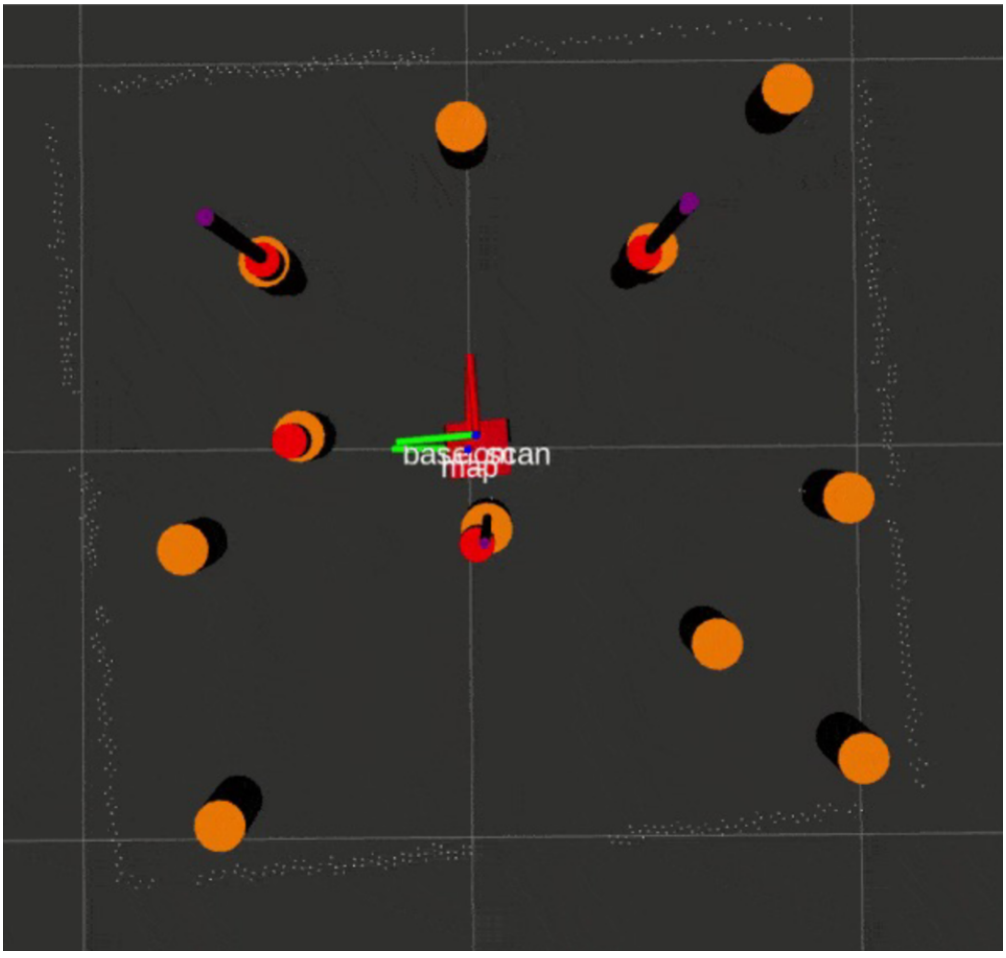


Loop Closure

- Recognizing an already mapped area, typially 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)



- By revisiting already mapped areas, uncertainties in robot 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



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.
- ...