Robot Positioning Estimation

2

In this chapter, we'll talk about robot positioning estimation, how can robots locate itself in unknown map (Localization), find the position in known map (Mapping) and what's the Simultaneous Localization and Mapping (SLAM).

2.1 Localization

Localization is determined the robot position and orientation, in outdoor setting localization we can use satellite-based GPS, but indoor setting the localization become headache problem.

Therefore, Localization is the challenge of determining our robot's pose in a pre-mapped environment, by implementing a probabilistic algorithm to filter noisy sensor measurements and track the robot's position and orientation.

Now, there're four very popular localization algorithms to localize the robot:

- Extended Kalman Filter Localization
- Markov Localization
- Grid Localization
- Monte Carlo Localization

We'll focus on Extended Kalman Filter Localization and Monte Carlo Localization.

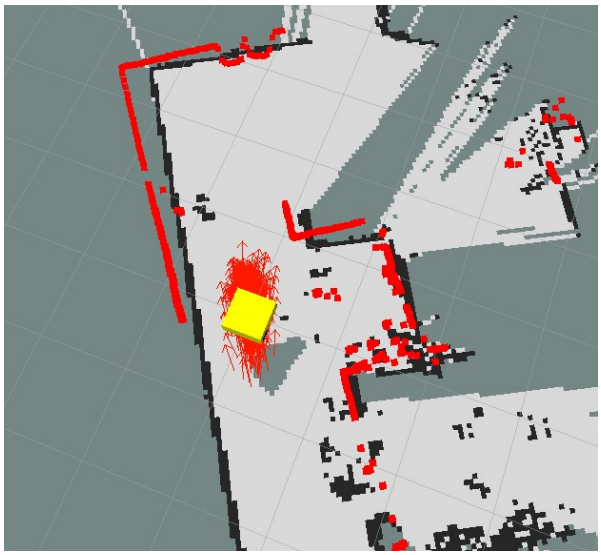
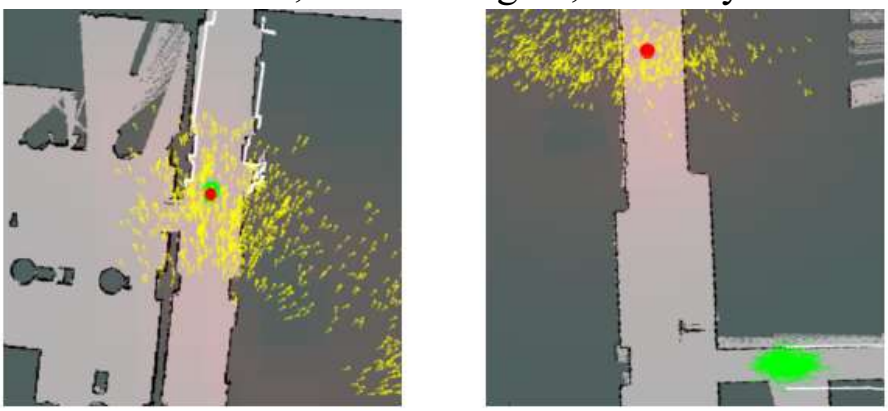


Figure 2.1 Robot Localization

2.1.a Localization Challenges

Before we dive into the specific localization algorithms, let's define the localization problem. Firstly, there are three different types of localization problems, these problems are not all equal. The amount of information present and the nature of the environment that a robot is operating in determine the difficulty of the localization task.

- Position Tracking Problem: The easiest localization problem, also known as Local Localization. In this problem, the robot knows its initial pose and the localization challenge entails estimating the robot's pose as it moves out on the environment. This problem is not as trivial as we might think since there is always some uncertainty in robot motion. However, the uncertainty is limited to regions surrounding the robot.
- Global Localization Problem: A more complicated localization challenge. In this case, the robot's initial pose is unknown, and the robot must determine its pose relative to the ground truth map. The amount of uncertainty in Global Localization is much greater than that in Position Tracking, making it a much more difficult problem.
- Kidnapped Robot problem: the most challenging localization problem. This problem is just like Global Localization except that the robot may be kidnapped at any time and moved to a new location on the map. Robot kidnapping is such a common occurrence, it's quite uncommon but think of it as the worst possible case. As robotics engineer, we should always design robots to deal with the worst circumstances. But keep in mind that localization algorithms are not free from error and there will be instances of a robot miscalculating where it is. The Kidnapped Robot problem teaches the robot to recover from such instances, and once again, correctly locate itself on a map.



(a) No KRP (b) KRP

Figure 2.2 Kidnapped Robot problem

2.1.b Extended Kalman Filter Localization

First let's talk about the Kalman filter, The Kalman filter is an estimation algorithm that is very prominent in controls. It's used to estimate the value of a variable in real time as the data is being collected. This variable can represent the position or velocity of a robot. The reason that the Kalman filter is so noteworthy is because it can take data with a lot of uncertainty or noise in the measurements and provide a very accurate estimate of the real value. Unlike other estimation algorithms, we don't need to wait for a lot of data to come in to calculate an accurate estimate, the Kalman filter also considers the uncertainty of the sensor readings, which are specific to the sensor being used and the environment that it is operating in.
Every time a measurement is recorded that's a two-step process, and the Kalman filter is a continuous iteration of these two steps.

- Measurement Update: Use the recorded measurement to update our state.
- State Prediction: Use the information that we have about the current state to predict what the future state will be.

At the start, we use an initial guess, then continue to iterate through these two steps and it doesn't take many iterations for our estimate to converge on the real value.

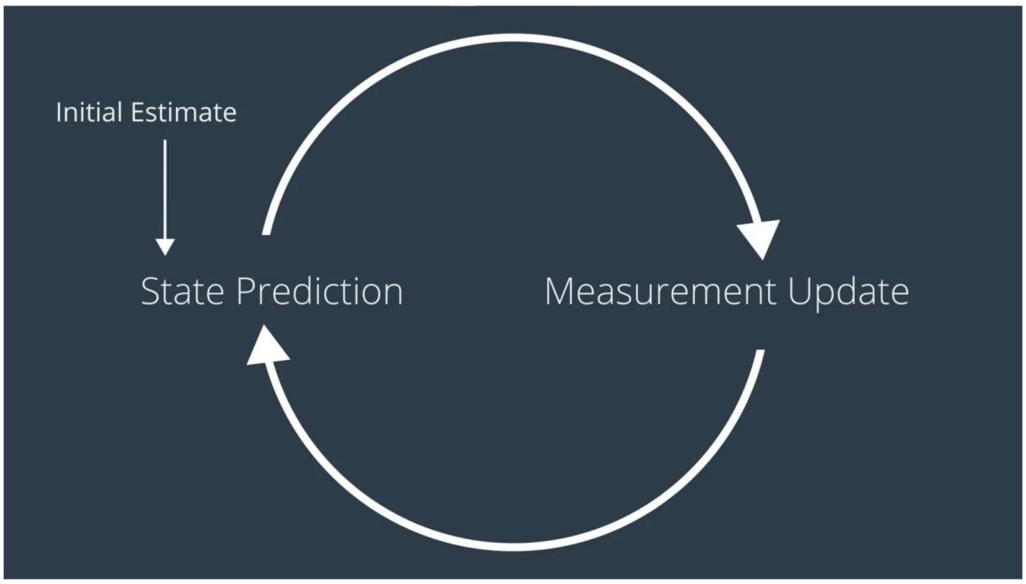


Figure 2.3 Kalman filter two-step process

The Kalman filter has become one of the most practical algorithms in the field of controls engineering. Today the Kalman filter is applied in many different disciplines within engineering the Kalman filter is often used to estimate the state of a system when the measurements are noisy. For example, position tracking of a mobile robot. Computer vision is another big user of the Kalman filter, for many different applications including feature tracking.

So far, we have been referring to the Kalman filter as one unique entity. However, they're three common types.

- Standard Kalman Filter: Can be applied to linear systems, ones where the output is proportional to the input.
- Extended Kalman Filter: Can be applied to non-linear systems which is more applicable in robotics, as real-world systems are more often non-linear than linear.
- Unscented Kalman Filter: Another type of non-linear estimator appropriate for highly non-linear systems where EKF may fail to converge.

The Kalman filter can very quickly develop a surprisingly accurate estimate of the true value of the variable being measured. For example, our robot's location in our one-dimensional real world. Unlike other algorithms that require a lot of data to make an estimate, the Kalman filter can do so after just a few sensor measurements. It does so by using an initial guess and by considering the expected uncertainty of a sensor or movement. If we use additional sensors on board the robot, we may be able to combine measurements from all of them to obtain a more accurate estimate. This is called sensor fusion; Sensor fusion uses the Kalman filter to calculate a more accurate estimate using data from multiple sensors. Once again, the Kalman filter considers the uncertainty of each sensor's measurements. So, whether it's making sense of noisy data from one sensor or from multiple, the Kalman filter is a very useful algorithm to use.

At the basis of the Kalman Filter is the Gaussian distribution, sometimes referred to as a bell curve or normal distribution. This is the role of a Kalman Filter - after a movement or a measurement update, it outputs a unimodal Gaussian distribution. This is its best guess at the true value of a parameter. A Gaussian distribution is a probability distribution, which is a continuous function. The probability that a random variable x will take a value between x_1 and x_2 is given by the integral of the function from x_1 to x_2

$$p(x_1 < x < x_2) = \int_{x_1}^{x_2} f_x(x)dx$$

A Gaussian is characterized by two parameters, its mean (μ) and its variance (σ^2). The mean is the most probable occurrence and lies at the center of the function, and the variance relates to the width of the curve.

There're two types of gaussian distribution:

- 1D Gaussian: The formula for the 1D Gaussian distribution is:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

Notice that the formula contains an exponential of a quadratic function. The quadratic compares the value of x to μ, and in the case that x=μ, the exponential is equal to 1 ($e^0 = 1$).
Just like with discrete probability, like a coin toss, the probabilities of all the options must sum to one. Therefore, the area underneath the function always sums to one.

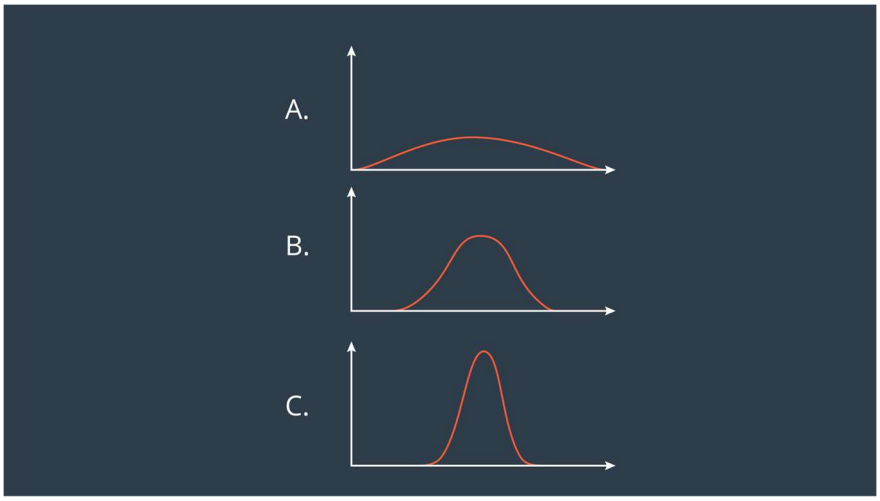


Figure 2.4 1D Gaussian

- Multivariate Gaussians: Most robots that we would be interested in modeling are moving in more than one dimension. For instance, a robot on a plane would have an x & y position. The simple approach to take, would be to have a 1D Gaussian represent each dimension, one for the x-axis and one for the y-axis.
The mean is now a vector:

$$\mu = \begin{bmatrix} \mu_x \\ \mu_y \end{bmatrix}$$

The variance is now a covariance matrix:

$$\Sigma = \begin{bmatrix} {\sigma_x}^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & {\sigma_y}^2 \end{bmatrix}$$

The multivariate Gaussian:

$$p(x) = \frac{1}{(2\pi)^{\frac{D}{2}}|\sum|^{\frac{1}{2}}} e^{-\frac{1}{2}(x-\mu)^T|\sum|^{-1}(x-\mu)}$$

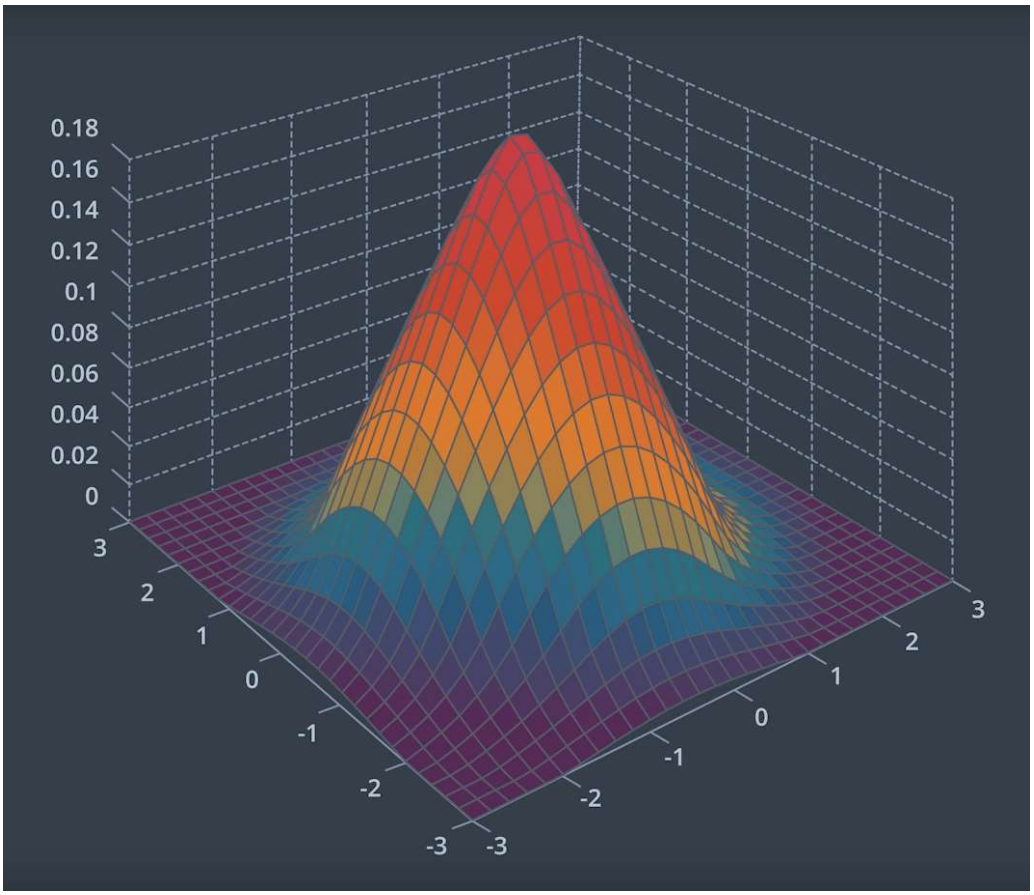


Figure 2.5 Multivariate Gaussian

For designing 1D Kalman filter, there're two parameters must consider we mentioned before:

- Measurement Update: Let's look at an example in a one-dimensional robot world. I have an inkling that the robot's current position is near the 20-meter mark, but I'm not incredibly certain. So, the prior belief Gaussian has a rather wide probability distribution. Next, a robot takes its first sensory measurement providing us with data to work with. The measurement data, Z, is more certain, so it appears as a narrower Gaussian with a mean of 30. The new Belief would have a mean somewhere in between the two Gaussians since it is combining information from both. The new mean is a weighted sum of the prior belief and measurement means. With uncertainty, a larger number represents a more uncertain probability distribution. However, the new mean should be biased towards the measurement update, which has a smaller standard deviation than the prior.

$$\mu' = \frac{r^2\mu + \sigma^2 v}{r^2 + \sigma^2}, \qquad \sigma^{2'} = \frac{1}{\frac{1}{r^2} + \frac{1}{\sigma^2}}$$

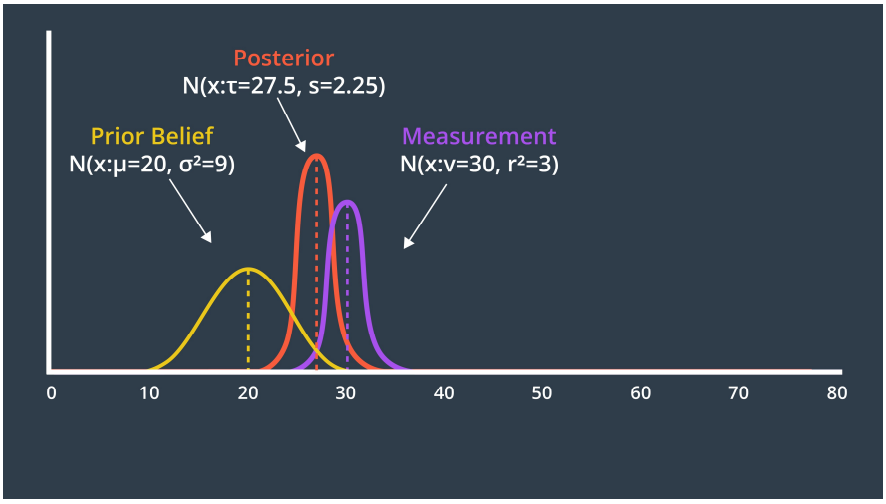


Figure 2.6 Measurement Update for 1D Kalman filter

- State Prediction: By implementing the measurement update, we've now completed half of the Kalman filter's iterative cycle. Recall that state prediction is the estimation that takes place after an inevitably uncertain motion. Since the measurement update and state prediction are an iterative cycle, it makes sense for us to continue where we left off. After considering the measurement, the posterior distribution was Gaussian with a mean of 27.5 and a variance of 2.25. However, since we've moved onto the state predictions step in the common filter cycle, this Gaussian is now referred to as the prior belief. This is the robot's best estimate of its current location. Next, the robot executes a command. Move forward 7.5 meters. The result of this motion is a Gaussian distribution centered around 7.5 meters with a variance of five meters. Calculating the new estimate is as easy as adding the mean of the motion to the mean of the prior, and similarly, adding the two variances together, to produce the posterior Gaussian.

$$\mu' = \mu_1 + \mu_2, \quad \sigma^{2'} = \sigma^2{}_1 + \sigma^2{}_2$$

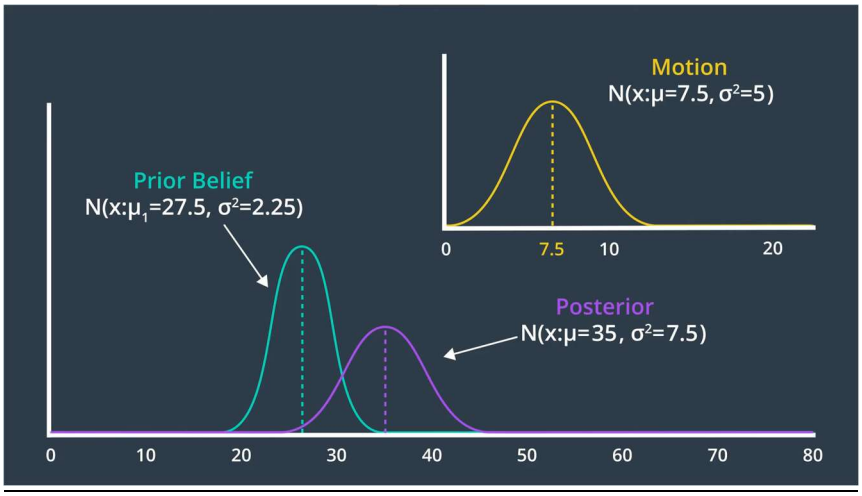


Figure 2.7 State Prediction for 1D Kalman filter

Design of Multi-Dimensional Kalman Filters, from this point forward we will transition to using linear algebra, as it allows us to easily work with multi-dimensional problems.

- State Transition: The formula is the state transition function that advances the state from time (t) to time (t + 1). It is just the relationship between the robot's position x and velocity $\dot{x}$, we will assume that the robot's velocity is not changing.

$$x' = x + \Delta t\dot{x} \;\; \& \;\; \dot{x}' = \dot{x}$$

We can express the same relationship in matrix form:

$$\begin{bmatrix} x \\ \dot{x} \end{bmatrix}' = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

The State Transition Function is denoted F, and the formula can be written as:

$$x' = Fx$$

The equation should also account for process noise, as its own term in the equation. However, process noise is a Gaussian with a mean of 0, so the update equation for the mean need not include it.

$$x' = Fx + noise \;\; \& \;\; noise \sim N(0, Q)$$

If we multiply the state x by F, then the covariance will be affected by the square of F. In matrix form, this will look like so:

$$P' = FPF^T$$

To calculate the posterior covariance, the prior covariance is multiplied by the state transition function squared, and Q added as an increase of uncertainty due to process noise. Q can account for a robot slowing down unexpectedly or being drawn off course by an external influence.

$$P' = FPF^T + Q$$

Now we've updated the mean and the covariance in the state prediction.

- Measurement Update: If we return to our original example, where we were tracking the position and velocity of a robot in the x-dimension, the robot was taking measurements of the location only (the velocity is a hidden state variable). Therefore, the measurement function is very simple - a matrix containing a one and a zero. This matrix demonstrates how to map the state to the observation z.

$$z = \begin{bmatrix} 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ \dot{x} \end{bmatrix}$$

This matrix, called the Measurement Function, is denoted H.

The measurement residual y, the measurement residual is the difference between the measurement and the expected measurement based on the prediction The measurement residual will be used later in a formula.

$$y = z - Hx'$$

It's time to consider the measurement noise denoted R. This formula maps the state prediction covariance into the measurement space and adds the measurement noise. The result S will be used in a subsequent equation to calculate the Kalman Gain.

$$S = HP'H^T + R$$

The Kalman Gain determines how much weight should be placed on the state prediction, and how much on the measurement update. It is an averaging factor that changes depending on the uncertainty of the state prediction and measurement update.

$$K = P'H^T S^{-1}$$

$$x = x' + Ky$$

The last step in the Kalman Filter is to update the new state's covariance using the Kalman Gain.

$$P = (I - KH)P'$$

Now, it's time to talk about Extended Kalman Filter (EKF). The Kalman Filter is applicable to problems with linear motion and measurement functions. This is limiting, as much of the real world is nonlinear. A nonlinear function can be used to update the mean of a function,

$$\mu \xrightarrow{f(x)} \mu'$$

but not the variance, as this would result in a non-Gaussian distribution which is much more computationally expensive to work with. To update the variance, the Extended Kalman Filter linearizes the nonlinear function f(x) over a small section and calls it F. This linearization, F, is then used to update the state's variance.

$$P \xrightarrow{F} P'$$

The linear approximation can be obtained by using the first two terms of the Taylor Series of the function centered around the mean.

$$F = f(x) + \frac{\delta f(\mu)}{\delta x}(x - \mu)$$

Design of Multi-Dimensional Extended Kalman Filters. Now we've seen the fundamentals behind the Extended Kalman Filter. The mechanics are not too different from the Kalman Filter, except for needing to linearize a nonlinear motion or measurement function to be able to update the variance. We've seen how this can be done for a state prediction or measurement function that is of one dimension, but now it's time to explore how to linearize functions with multiple dimensions. To do this, we will be using the multi-dimensional Taylor series.

- Linearization in Multiple Dimensions: The equation for a multidimensional Taylor Series is presented below.

$$T(x) = f(a) + \frac{1}{n!}(x-a)^T D^n f(a)$$

We will see that it is very similar to the 1-dimensional Taylor Series. As before, to calculate a linear approximation, we only need the first two terms.

$$T(x) = f(a) + (x-a)^T Df(a)$$

We may notice a new term $Df(a)$. This is the Jacobian matrix, and it holds the partial derivative terms for the multi-dimensional equation.

$$Df(a) = \frac{\delta f(a)}{\delta x}$$

In its expanded form, the Jacobian is a matrix of partial derivatives. It tells us how each of the components of f changes as we change each of the components of the state vector.

$$Df(a) = \begin{bmatrix} \frac{\delta f_1}{\delta x_1} & \frac{\delta f_1}{\delta x_2} & \cdots & \frac{\delta f_1}{\delta x_n} \\ \frac{\delta f_2}{\delta x_1} & \frac{\delta f_2}{\delta x_2} & \cdots & \frac{\delta f_2}{\delta x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\delta f_m}{\delta x_1} & \frac{\delta f_m}{\delta x_2} & \cdots & \frac{\delta f_m}{\delta x_n} \end{bmatrix}$$

The rows correspond to the dimensions of the function, f, and the columns relate to the dimensions (state variables) of x. The first element of the matrix is the first dimension of the function derived with respect to the first dimension of x. The Jacobian is a generalization of the 1-dimensional case. In a 1-dimensional case, the Jacobian would have df/dx as its only term.

- Extended Kalman Filter Equations: These are the equations that implement the Extended Kalman Filter - we'll notice that most of them remain the same, with a few changes:
  - State Prediction:

$$x' = f(x)$$
$$P' = FPF^T + Q$$

  - Measurement Update:

$$y = z - h(x')$$
$$S = HP'H^T + R$$

  - Calculation of Kalman Gain:

$$K = P'H^T S^{-1}$$

  - Calculation of Posterior State and Covariance:

$$x = x' + Ky$$
$$P = (I - KH)P'$$

  Highlighted in blue are the Jacobians that replaced the measurement and state transition functions. The Extended Kalman Filter requires us to calculate the Jacobian of a nonlinear function as part of every single iteration since the mean (which is the point that we linearize about) is updated.

Here are the key take-aways about Extended Kalman Filters:

- The Kalman Filter cannot be used when the measurement and/or state transition functions are nonlinear since this would result in a non-Gaussian distribution.
- Instead, we take a local linear approximation and use this approximation to update the covariance of the estimate. The linear approximation is made using the first terms of the Taylor Series, which includes the first derivative of the function.
- In the multi-dimensional case, taking the first derivative isn't as easy as there are multiple state variables and multiple dimensions. Here we employ a Jacobian, which is a matrix of partial derivatives, containing the partial derivative of each dimension with respect to each state variable.

While it's important to understand the underlying math to employ the Kalman Filter. As a software package or programming language, we're working with will have libraries that allow we to apply the Kalman Filter.

2.1.e Monte Carlo Localization

The Monte Carlo Localization algorithm (MCL) is the most popular localization algorithm in robotics. So, we might want to choose MCL and deploy it to our robot to keep track of its pose. Now, our robot will be navigating inside its known map and collecting sensory information using range finder sensors. MCL will use these sensor measurements to keep track of our robot pose. Many scientists refer to MCL as a particle filter localization algorithm since it uses particles to localize our robot. In robotics, we can think of a particle as a virtual element that resembles a robot. Each particle has a position and orientation and represents a guess of where our robot might be located. These particles are re-sampled each time our robot moves and sense its environment. Keep in mind that MCL is limited to local and global localization problems only. So, we lose sight of our robot if someone hacks into it. We can estimate the pose of almost any robot with accurate onboard sensors. MCL presents many advantages over EKF:

- First, MCL is easy to program compared to EKF.
- Second, MCL represents non-Gaussian distributions and can approximate any other practical important distribution. This means that MCL is unrestricted by a linear Gaussian states-based assumption as is the case for EKF. This allows MCL to model a much greater variety of environments especially since we can't always model the real world with Gaussian distributions.
- Third, in MCL, we can control the computational memory and resolution of our solution by changing the number of particles distributed uniformly and randomly throughout the map.

Compare	MCL	EKF
Measurement	Raw Measurements	Landmarks
Measurement Noise	Any	Gaussian
Posterior	Particles	Gaussian
Memory Efficiency	Acceptable	Very Good
Time Efficiency	Acceptable	Very Good
Ease of Implementation	Very Good	Acceptable
Resolution	Acceptable	Very Good
Robustness	Very Good	Unacceptable
Memory and Resolution Control	Yes	No
Global Localization	Yes	No
State Space	Multi-model Discrete	Uni-model Continuous

Table 2.1 MCL vs. EKF

With the Monte Carlo localization algorithm. Particles are initially spread randomly and uniformly throughout this entire map. These particles do not physically exist and are just shown in the simulation. Every red arrow represents a single particle. And just like the robot, each particle has an x coordinate, y coordinate, and orientation vector. So, each of these particles represents the hypothesis of where the robot might be. In addition to the three-dimensional vector, particles are each assigned a weight. The weight of a particle is the difference between the robot's actual pose and the particle's predicted pose. The importance of a particle depends on its weight, and the bigger the particle, the more accurate it is. Particles with large weights are more likely to survive during a resampling process. After the resampling process, particles with significant weight are more likely to survive whereas the others are more likely to die. Finally, after several iterations of the Monte Carlo localization algorithm, and after different stages of resampling, particles will converge and estimate the robot's pose.

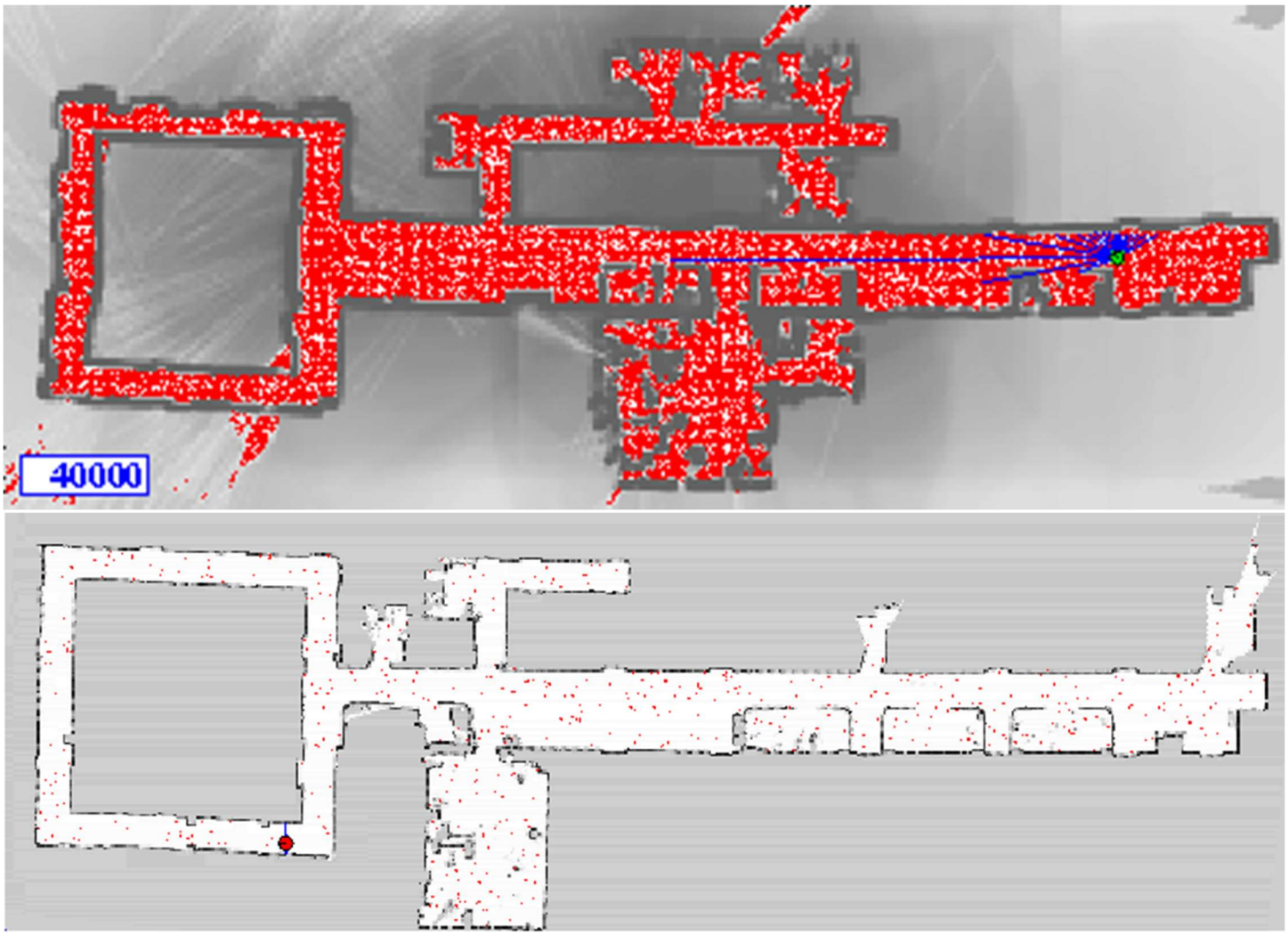


Figure 2.8 Particle Filters in MCL

The powerful Monte Carlo localization algorithm estimates the posterior distribution of a robot's position and orientation based on sensory information. This process is known as a recursive Bayes filter.
Using a Bayes filtering approach, roboticists can estimate the state of a dynamical system from sensor measurements. In mobile robot localization, it's important to be acquainted with the following definitions:

- Dynamical system: The mobile robot and its environment.
- State: The robot's pose, including its position and orientation.
- Measurements: Perception data like laser scanners and odometry data like rotary encoders.

The goal of Bayes filtering is to estimate a probability density over the state space conditioned on the measurements. The probability density, also known as posterior, is called the belief and is denoted as:

$$Bel(X_t) = P(X_t|Z_{1\dots t})$$

Where, X_t: state at time t and $Z_{1\dots t}$: Measurement from time 1 up to time t.
Given a set of probabilities, P(A|B) is calculated as follows:

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

The Monte Carlo Localization Algorithm is composed of two main sections represented by two for loops. The first section is the motion and sensor update, and the second one is the resampling process. Given a map of the environment, the goal of the MCL is to determine the robot's pose represented by the belief.

MCL Algorithm (X_{t-1}, u_t, z_t):
    $\bar{X}_t = X_t = \emptyset$
    for m = 1 to M:
        $x_t^{[m]} = motion\_update(u_t, x_{t-1}^{[m]})$
        $\omega_t^{[m]} = sensor\_update(z_t, x_t^{[m]})$
        $\bar{X}_t = \bar{X}_t + \langle x_t^{[m]}, \omega_t^{[m]} \rangle$
    endfor
    for m = 1 to M:
        draw $x_t^{[m]}$ from $\bar{X}_t$ with probability $\alpha\, \omega_t^{[m]}$
        $X_t = X_t + \omega_t^{[m]}$
    endfor
    return X_t

- At each iteration, the algorithm takes the previous belief, the actuation command, and the sensor measurements as input.
- Initially, the belief is obtained by randomly generating m particles.
- Then, in the first for loop, the hypothetical state is computed whenever the robot moves. Following, the particles weight is computed using the latest sensor measurements. Now, motion and measurement are both added to the previous state.
- The second section of the MCL is where a sampling process happens. Here, the particles with high probability survive and are re-drawn in the next iteration, while the others die.
- Finally, the algorithm outputs the new belief
- Another cycle of iteration starts implementing the next motion by reading the new sensor measurements.

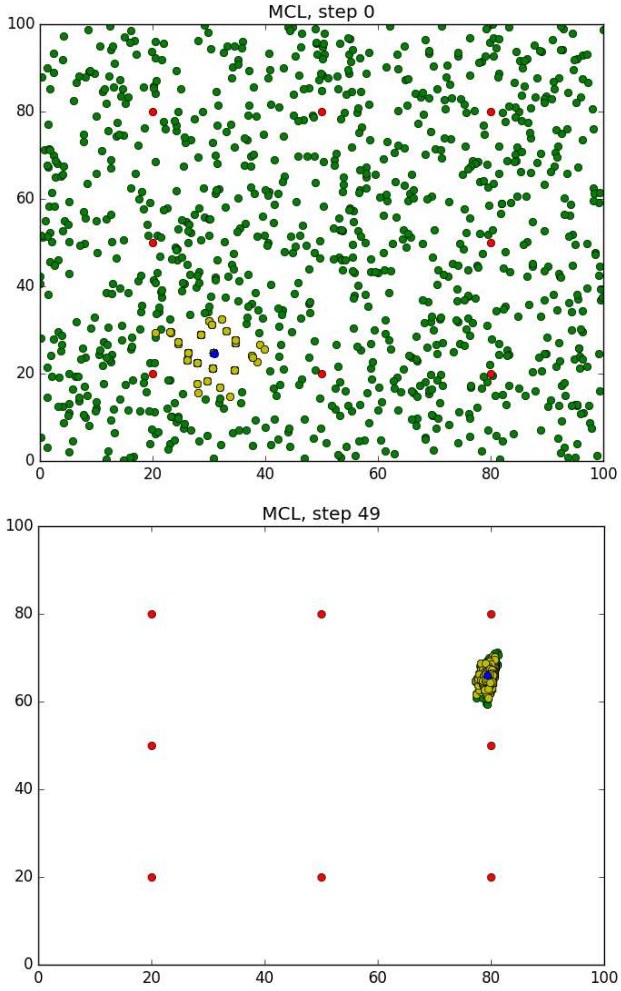


Figure 2.9 MCL algorithm result

2.2 Mapping

As we recall, in localization, the robot is provided a map of its environment. The robot also has access to its movement and sensor data and using the three of these together, it must estimate its pose. But what if a map of the environment doesn't exist? There are many applications where there isn't a known map, either because the area is unexplored or because the surroundings change often, and the map may not be up to date. In such a case, the robot will have to construct a map.
This leads us to robotic mapping. Mapping assumes that a robot knows its pose, and as usual, has access to its movement and sensor data. The robot must produce a map of the environment using the known trajectory and measurement data.

Robot mapping sounds quite like localization. Instead of assuming an old map and estimating our pose, we're assuming an old path and estimating our environment. Well, our poses can be defined with some finite number of state variables such as the robot's x and y position. In most applications, there is only a handful. On the other hand, a map generally lies in a continuous space, and as a result, there are infinitely many variables used to describe it. Couple this with the uncertainties present in perception and the mapping task becomes even more challenging. Depending on the nature of the space that we are mapping. For instance, the geometry and whether the space is repetitive in the sense that many different places look alike. If we're driving through a forest that only contains trees of one type, the kind where they are neatly planted in rows. We're going to have a lot of trouble distinguishing between trees and establishing correspondences. Thus, making it challenging to map the environment.

There are several different mapping algorithms, and we will be focusing on the occupancy grid mapping algorithm. With this algorithm, we can map any arbitrary environment by dividing it into a finite number of grid cells. By estimating the sets of each individual cell, we will end up with an estimated map of our environment.

Compare	Localization	Mapping
Assumption	Known Map	Robot's Trajectory
Estimation	Robot's Trajectory	Map

Table 2.2 Localization vs. Mapping

2.2.a Mapping Challenges

In localization, our goal was to estimate the robot's pose in a known environment. We did this by assuming a known map that was previously generated by the robot, drawn by hand, or provided by an architect. But what if the map is unavailable and the robot must navigate in an undiscovered area like another planet. So, now our goal has changed from estimating a robot's pose in a known map to estimating the map itself. The problem of generating a map inside of an unknown environment is called mapping.

Mapping with mobile robots is a challenging problem for two main reasons:

- Unknown Map and Poses: In localization, we assumed the known map and aim to estimate the robot's pose. Now, the map is unknown to us, so we'll either must assume known poses and estimate the map or assume unknown poses and estimate the map and the poses relative to it. Estimating the map on unknown poses is a challenging problem because of the large number of variables. This can be solved using the occupancy grid mapping algorithm.
- Huge Hypothesis Space: The hypothesis space is the space of all possible maps that can be formed during mapping. This space is highly dimensional since maps are defined over a continuous space, especially, when the robots are deployed in open environments where they'll have to sense an infinite number of objects. The occupancy grid mapping algorithm presents a discrete approximation of the map. But even under this discrete approximation, the space of all possible maps would still be high. So, the challenge is to estimate the full posterior map, for maps with high dimensional spaces. The base filtering approach that we use in localization to estimate the posterior pose will diverge and an extension to it should be used in mapping to accommodate for the huge hypothesis space.

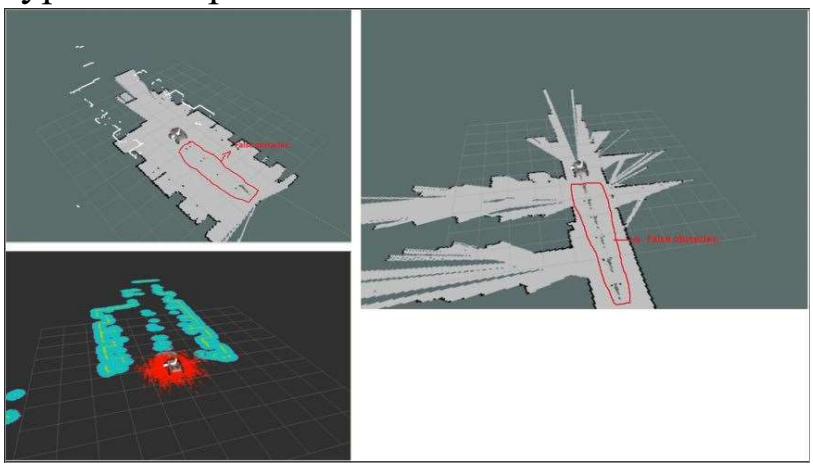


Figure 2.10 Huge Hypothesis Space

Mapping with mobile robots is facing difficulties, to map an environment, we need information about walls and objects. For example, we can deploy a mobile robot with a laser range finder sensor. The robot collects sensory information which permits it to detect obstacles in the environment. By using one of the mapping algorithms, we can group this data into a resulting map. In this example, we see that there are several possible data represented by instantaneous maps that the robot must combine to estimate the actual map. This problem is not as easy as we might think, as there are three difficulties while mapping an environment.

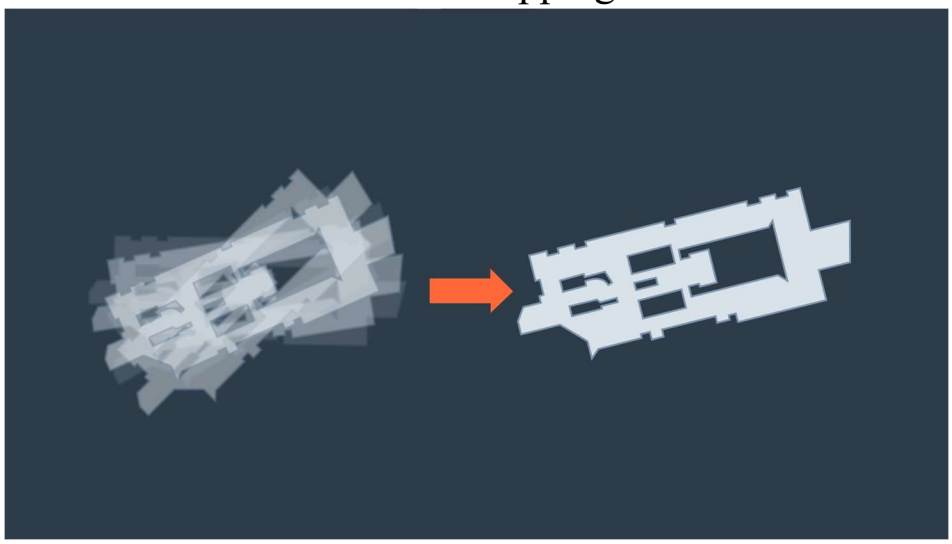


Figure 2.11 Mapping algorithm to combine estimation poses

- Size of the environment: Mapping large areas is hard because of the large amount of data to be processed. The robot's on-board micro-controller must collect all the instantaneous poses and obstacles, then form a resulting map and localize the robot with respect to this map. Also, the mapping problem becomes even more difficult when the size of the map is larger than the robot's perceptual range.

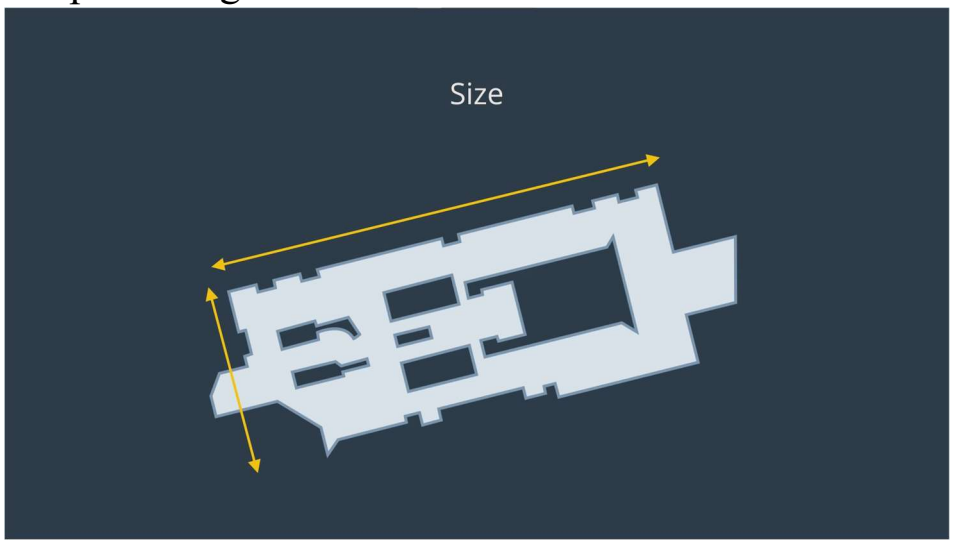


Figure 2.12 size of the environment in mapping

- Noise: always exists in perception sensors, audiometry sensors, and actuators. During mapping, we must always filter the noise from these sensors and actuators. With large noise, mappings become more difficult and challenging as this noise adds up.

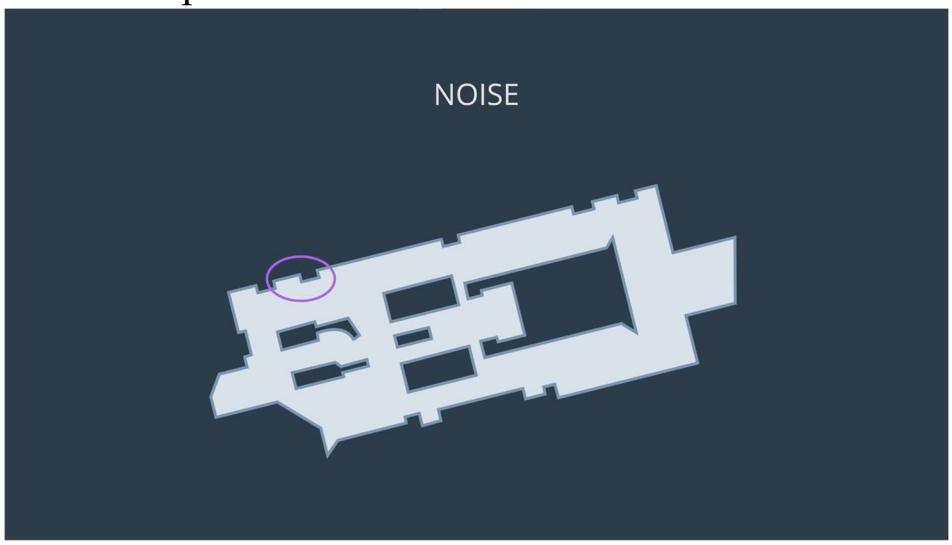


Figure 2.13 Noise in mapping

- Perceptual Ambiguity: The ambiguity occurs when two places look alike, and the robot must correlate between these two places which the robot travel through at different points in time. Another difficulty in mapping occurs when a robot travels in a cyclic manner, for example, going back and forth in a corridor. When traveling in cycles, robot audiometry incrementally accumulates error. And at the end of the cycle, the error is quite large.

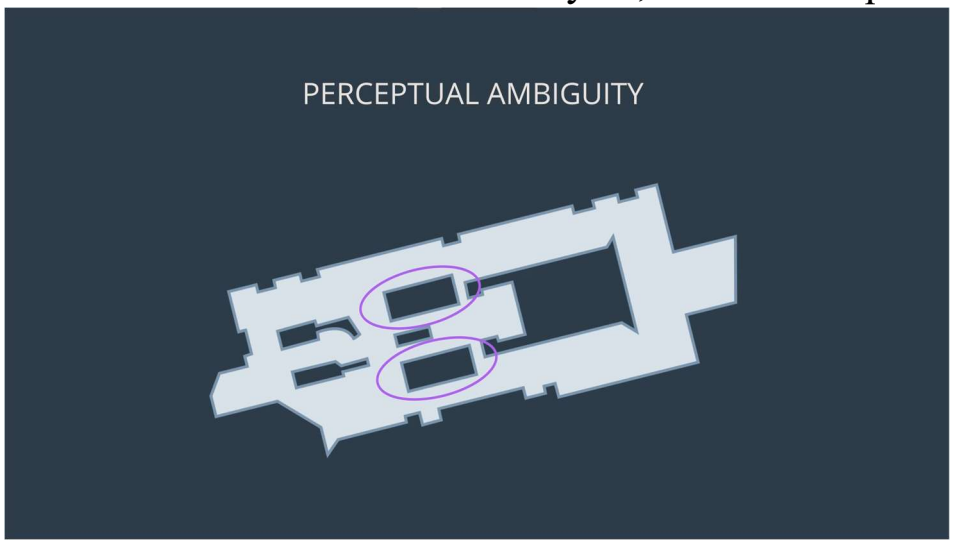


Figure 2.14 Perceptual Ambiguity in mapping

2.2.b Occupancy Grid Map

The problem of generating a map under the assumption that the robot poses are known and non-noisy is referred to as mapping with known poses. The mapping was known poses problem is represented in this graph where x represents the poses, z the measurements, and m the map.

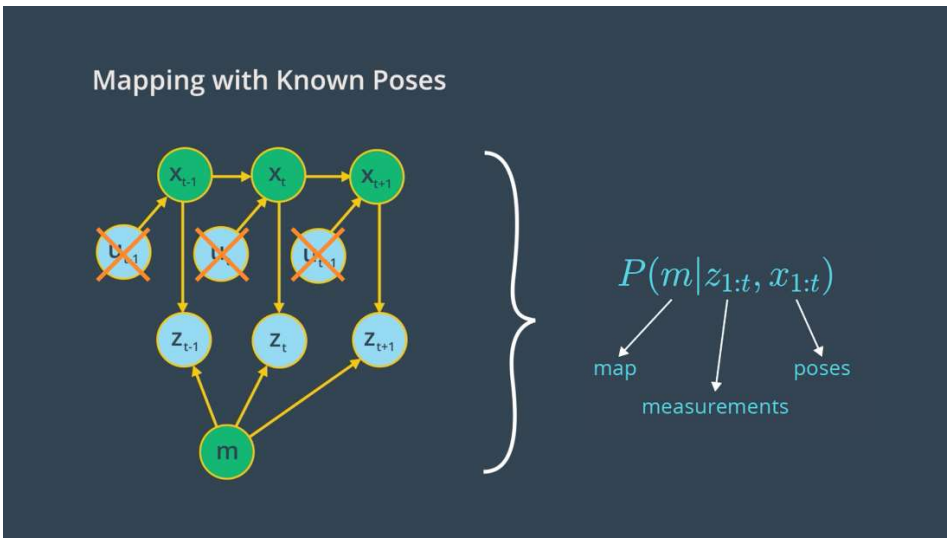


Figure 2.15 Mapping was known poses

- Posterior Probability: The Mapping with Known Poses problem can be represented with : $P(m|z_{1:t}, x_{1:t})$ function. With this function, we can compute the posterior over the map given all the measurements up to time t and all the poses up to time t represented by the robot trajectory.
- 2D Map: For now, we will only estimate the posterior for two-dimensional maps. In the real world, a mobile robot with a two-dimensional laser rangefinder sensor is generally deployed on a flat surface to capture a slice of the 3D world. Those two-dimensional slices will be merged at each instant and partitioned into grid cells to estimate the posterior through the occupancy grid mapping algorithm. Three-dimensional maps can also be estimated through the occupancy grid algorithm, but at much higher computational memory because of the large number of noisy three-dimensional measurements that need to be filtered out.

- Grid Cells: To estimate the posterior map, the occupancy grid will uniformly partition the two-dimensional space in a finite number of grid cells. Each of these grid cells will hold the miner random value that corresponds to the location it covers. Based on the measurements data, this grid space will be filled with zeros and ones. If the laser range finder sensor detects an obstacle, the cell will be considered as occupied and its value will be one. And for free spaces, the cell will be considered as unoccupied, and its value will be zero.

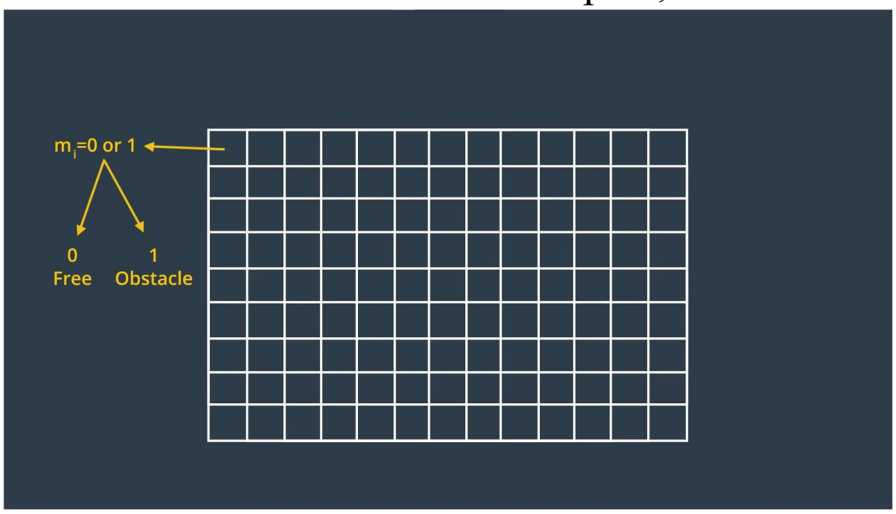


Figure 2.16 2D map grid cells

let's pick the first two cells and work out their combinations. The first combination is a map with two free spaces. Proceeding, the second combination is a map with one free space on the left side and one obstacle on the right side. Next, by flipping the second map, we get the third combination. And finally, the fourth combination is a map with two obstacles. Thus, the total number of maps that can be formed out of these two grid cells is equal to four. To generalize, the number of maps that can be formed out of grid cells are equal to two to the power of number of cells.

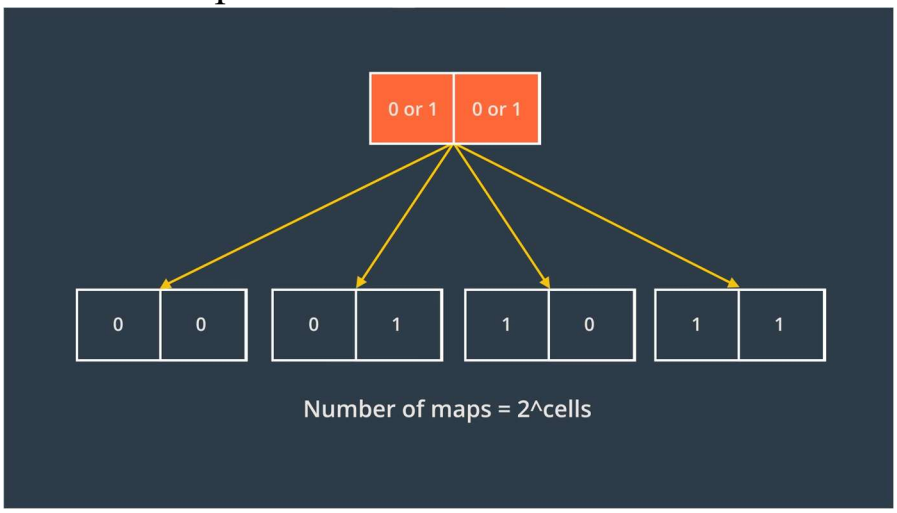


Figure 2.17 2D map cells representation

- Computing the Posterior:

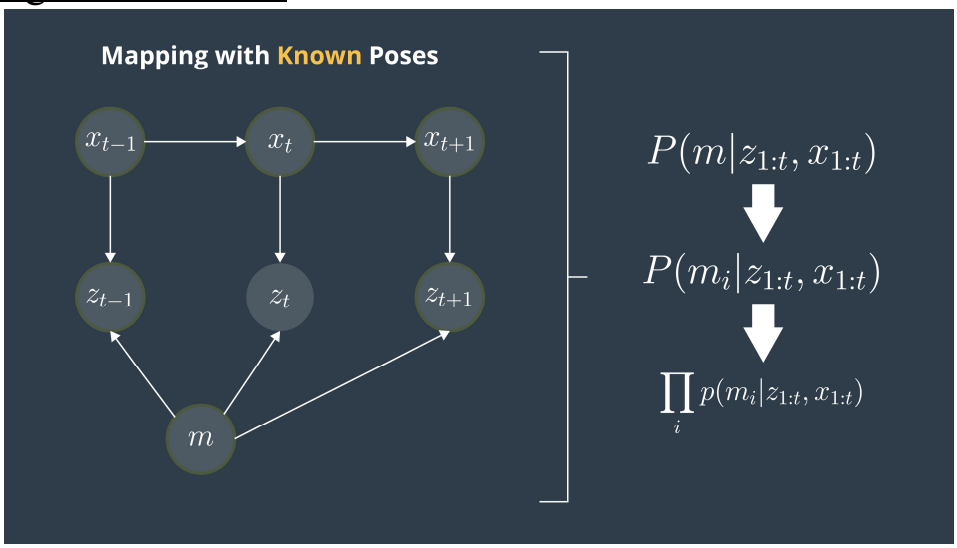


Figure 2.18 2D map Computing the Posterior

- First Approach:

$$P(m|z_{1:t}, x_{1:t})$$

We just saw that maps have high dimensionality so it will be too pricey in terms of computational memory to compute the posterior under this first approach.

- Second Approach:

$$P(m_i|z_{1:t}, x_{1:t})$$

A second or better approach to estimating the posterior map is to decompose this problem into many separate problems. In each of these problems, we will compute the posterior map m_i at each instant. However, this approach still presents some drawbacks because we are computing the probability of each cell independently. Thus, we still need to find a different approach that addresses the dependencies between neighboring cells.

- Third Approach:

$$\prod_i^n P(m_i|z_{1:t}, x_{1:t})$$

Now, the third approach is the best approach to computing the posterior map by relating cells and overcoming the huge computational memory, is to estimate the map with the product of marginals or factorization.

- Binary Bayes Filter Algorithm:

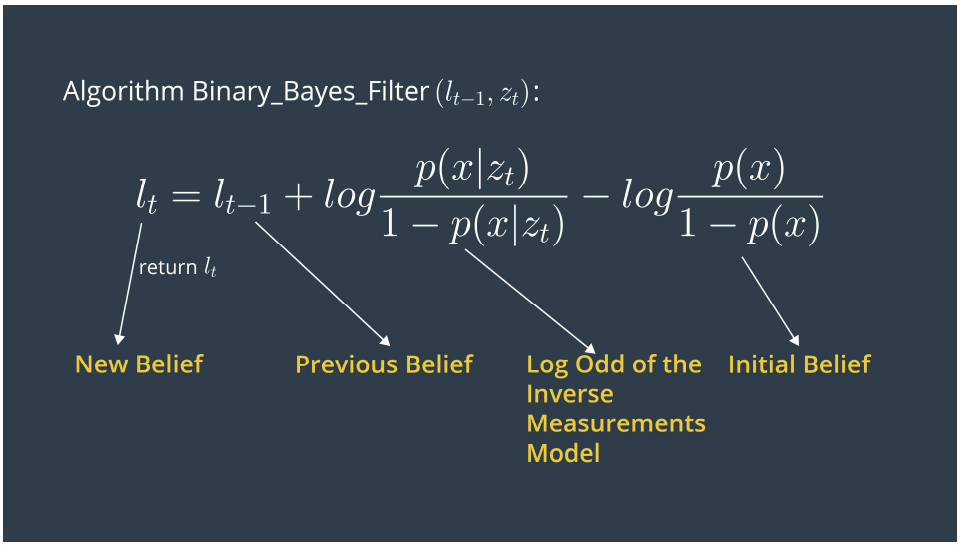


Figure 2.19 Binary Bayes Filter Algorithm

- Input:
  The binary Bayes filter algorithm computes the log odds of the posterior belief denoted by I_t. Initially, the filter takes the previous log odds ratio of the belief $._{t-1}$and the measurements z_t as parameters.
- Computation:
  Then, the filter computes the new posterior belief of the system lt by adding the previous belief lt-1 to the log odds ratio of the inverse measurement model:

$$log\frac{p(x|z_t)}{1-p(x|z\_t)}$$

  And subtracting the prior probability state also known by initial belief:

$$log\frac{p(x)}{1-p(x)}$$

  The initial belief represents the initial state of the system before taking any sensor measurements into consideration.
- Output:
  Finally, the algorithm returns the posterior belief of the system lt, and a new iteration cycle begins.

- The occupancy grid mapping algorithm: By implementing a binary base filter to estimate the occupancy value of each cell.

```
Occupancy_Grid_Mapping Algorithm ({l_{t-1,i}}, x_t, z_t):
    for all cell m_i do:
        if m_i in perceptual field of z_t then
            l_{t,i} = l_{t-1,i} + Inverse_Sensor_Model(m_i, x_t, z_t) - l_0
        else
            l_{t,i} = l_{t-1,i}
        endif
    endfor
    return l_{t,i}
```

  - Initially, the algorithm takes the previous occupancy values of the cells, the poses, and the measurements as parameters. The occupancy grid map now loops through all the grid cells. For each of the cells, the algorithm tests if the cells are currently being sensed by the range finder sensors.
  - The cells that fall under the measurement cones are highlighted in white and black color. While looping through all the cells, the algorithm will consider those white and black cells as sets falling under the perceptual field of the measurements.
  - The algorithm will update the cells that fall under the measurement cones by computing their new belief using the binary base filter algorithm. The new state of the cell is computed by adding the previous belief to the inverse sensor model.
  - For the cells that do not fall under the measurement cone, their occupancy value remains unchanged. The algorithm returns the updated occupancy values of the cells, and another cycle of iteration starts.

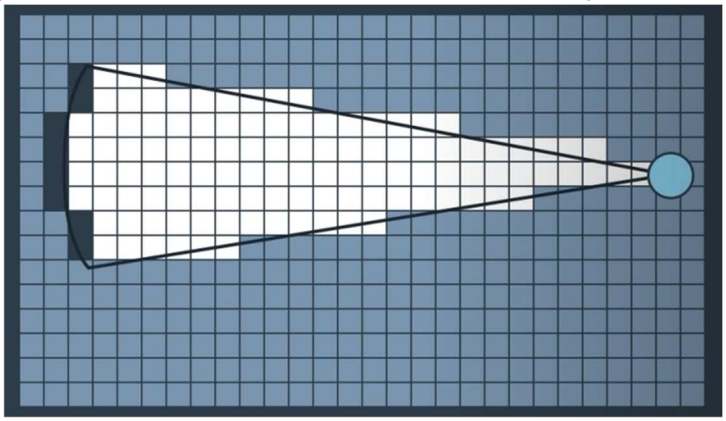


Figure 2.20 2D map measurement cone

- Inverse Sensor Model: In the Occupancy Grid Mapping Algorithm, we saw the inverse sensor model which the probability of the map is giving the measurements and the poses in its log odds representation. As a reminder, the advantage of using log odds representation is to avoid any numerical instabilities near zero or one. The inverse sensor model algorithm takes each cell, the poses, and measurements as parameters.

Inverse_Sensor_Model Algorithm (m_i, x_t, z_t):

Let x_i and y_i be the center of mass of m_i

$r = \sqrt{(x_i - x)^2 + (y_i - y)^2}$

$\phi = \text{atan2}(y_i - y, x_i - x) - \theta$

$k = argmin_j \left|\phi - \theta_{j,sens}\right|$

if $r > \min\left(z_{max}, {z_t}^k + \frac{\alpha}{2}\right)$ or $\left|\phi - \theta_{j,sens}\right| > \frac{\beta}{2}$

return l_0

if ${z_t}^k < z_{max}$ and $\left|r - {z_t}^k\right| < \frac{\alpha}{2}$

return l_{occ}

if $r \leq {z_t}^k$

return l_{free}

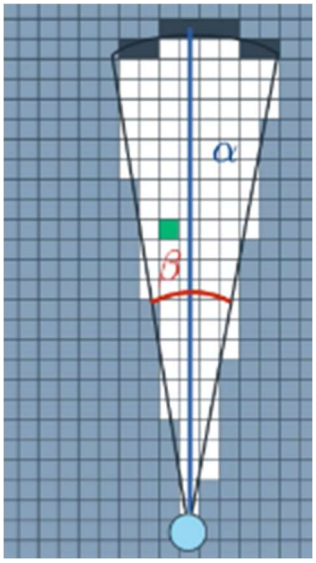


Figure 2.21 2D map Inverse Sensor Model

- First, the range for the center of mass of the cell is computed. Here, x_i and y_i are the center of mass of the cell m_i. R is the range for the center of mass, computed with respect to the center of mass and robot position.
- Next, the beam index k is computed with respect to the center of mass of the cell and the robot pose.
- Now, two parameters are introduced. Beta is the opening angle of the mean and alpha, it's the width of the region.
- If a cell lies outside the measurement range of the sensor beam or more than alpha over two behind its detected range, then the state of this cell is considered as unknown and its initial or prior state l0 and its log odds form is returned.
- Now, the cell is considered as occupied if it ranges between negative and positive alpha over two of the detected range.
- Finally, the cell is considered as free if the range of this cell is shorter than the measured range by more than alpha over two.

- Multi-Sensor Fusion: So far, we've covered mapping for robots equipped with only one sensor, such as mobile robots equipped with an ultrasonic sensor or LIDAR sensor, or even an RGB-D sensor. Sometimes the mobile robot might be equipped with a combination of these sensors. Mapping with a mobile robot equipped with a combination of these sensors leads to a more precise map. But how would we combine the information from a LIDAR and RGB-D sensor or any other combination into a single map? Well, an intuitive approach would be to implement the Occupancy Grid Mapping Algorithm and sell for each sensor model. But this will fail since each sensor has different characteristics. In addition to that, sensors are sensitive to different obstacles. For example, an RGB-D camera might detect an obstacle at a particular location, but this object might not be seen by the laser beams of a LIDAR, and thus, the LIDAR will process it as free space. The best approach to the multi-sensor fusion problem is to build separate maps for each sensor type independently of each other and then integrate them. Here we can see two independent 2-by-2 maps created by each sensor. Within the same environment, both maps do not look the same since each sensor has a different sensitivity. Now let's denote the map built by the kth sensor as MK and combine the maps. If the measurements are independent of each other, we can easily combine the maps using De Morgan's Law. The resultant map now combines the estimated occupancy values of each cell. To obtain the most likely map, we need to compute the maximum value of each cell. Another approach would be to perform a null operation between the values of each cell. The resulting map now perfectly combines the measurement from different sensors.

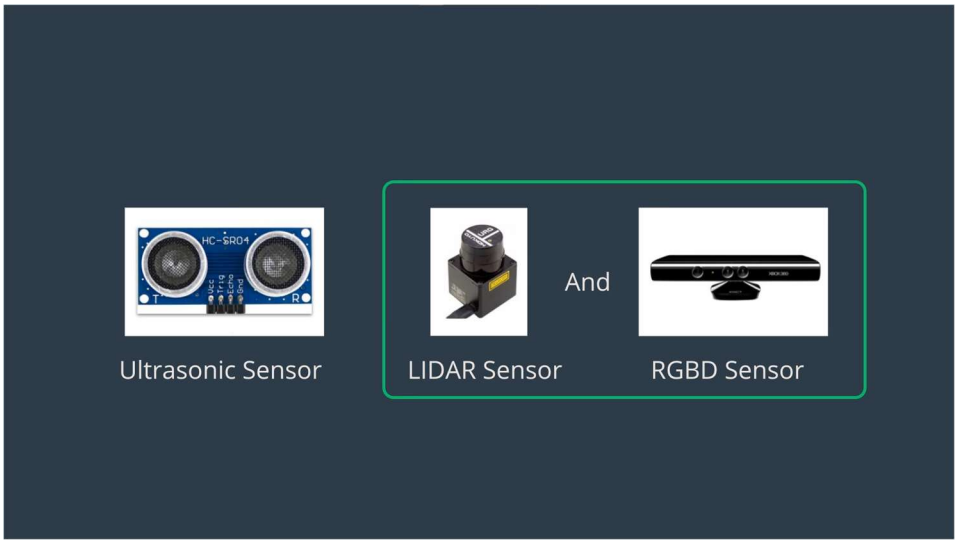


Figure 2.22 Sensor Fusion for mapping

2.2.c 3D Map

So far, we heard about two-dimensional maps, describing a slice of the 3D world. In resource-constrained systems, it can be very computationally expensive to build and maintain these maps. 3D representations are even more costly. Robots live in the 3D world, and we want to represent that world and the 3D structures within it as accurately and reliably as possible. 3D mapping would give us the most reliable collision avoidance, and motion and path planning, especially for flying robots or mobile robots with manipulators.

First, let's talk briefly about how we collect this 3D data, then we will move on to how it is represented. To create 3D maps, robots sense the environment by taking 3D range measurements. This can be done using numerous technologies:

- 3D lidar: Can be used, which is a single sensor with an array of laser beams stacked horizontally. Alternatively, a 2D lidar can be tilted (horizontally moving up and down) or rotated (360 degrees) to obtain 3D coverage.
- RGBD camera: Is a single visual camera combined with a laser rangefinder or infrared depth sensor and allows for the determination of the depth of the image, and ultimately the distance from an object. A stereo camera is a pair of offset cameras and can be used to directly infer the distance of close objects, in the same way as humans do with their two eyes.
- Single camera system: Is cheaper and smaller, but the software algorithms needed for monocular SLAM are much more complex. Depth cannot be directly inferred from the sensor data of a single image from a single camera. Instead, it is calculated by analyzing data from a sequence of frames in a video.

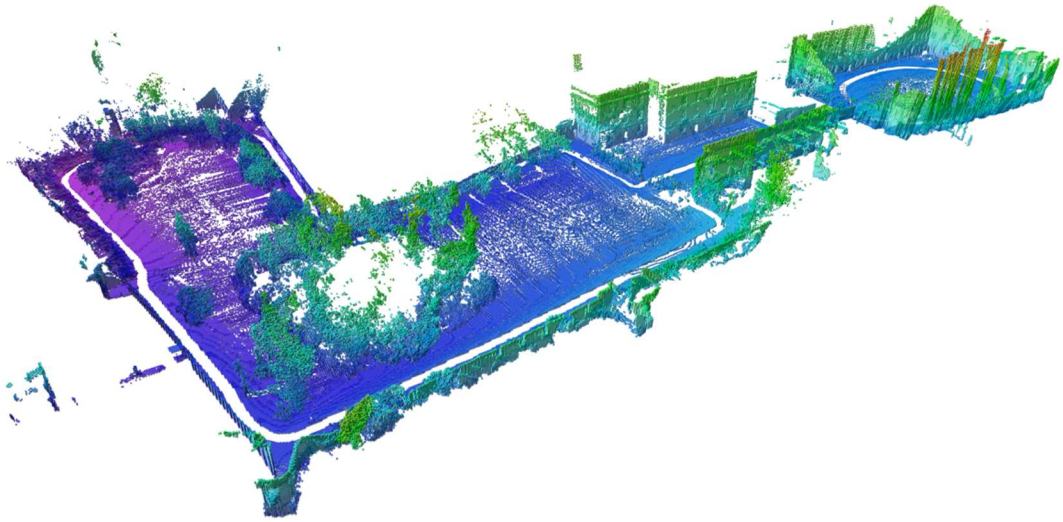


Figure 2.23 3D Occupancy Grid Mapping

Some of the desired characteristics of an optimal representation:

- Probabilistic data representations can be used to accommodate for sensor noise and dynamic environments.
- It is important to be able to distinguish data that represents an area that is free space versus an area that is unknown or not yet mapped. This will enable the robot to plan an unobstructed path and build a complete map.
- Memory on a mobile robot is typically a limited resource, so memory efficiency is very important. The map should also be accessible in the robot's main memory, while mapping a large area over a long period of time. To accomplish this, we need a data representation that is compact and allows for efficient updates and queries.

Some data representations of 3D environments:

- point clouds: A 3D point cloud is a set of data points corresponding to range measurements, each at defined x y z coordinates. A disadvantage with point cloud data is that information only exists about where things are in the world. The data looks the same whether the space is unoccupied or unknown. Point clouds also store a large amount of measurement points. And with each scan, we need to allocate more memory, so they are not memory efficient.

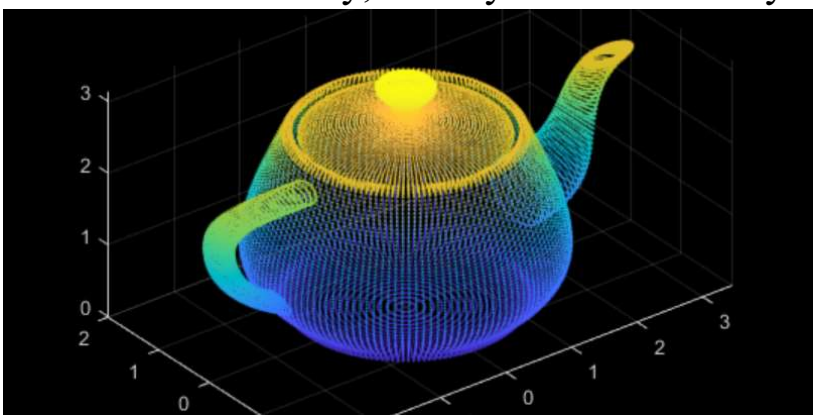


Figure 2.24 3D Point Clouds

- voxels grid: Is a volumetric data representation using a grid of cubic volumes of equal size. This is a probabilistic representation. So, we can estimate whether the voxel grid is occupied, free, or unknown space. One drawback of a 3D voxel grid is that the size of the area must be known or approximated before measurement, which may not always be possible. A second drawback is that the complete map must be allocated in memory. So, the overall memory requirement is high.

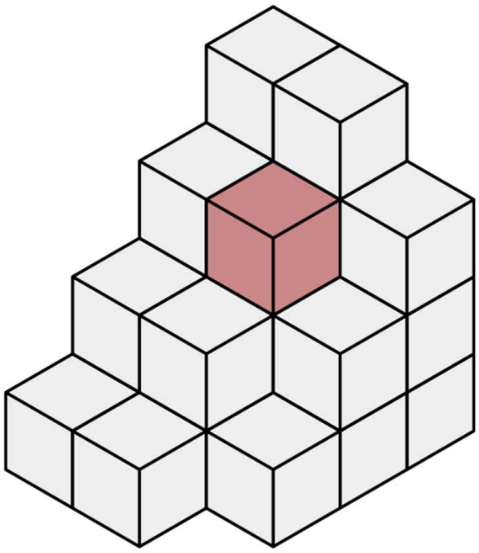


Figure 2.25 3D voxels grid

- octrees: Are a memory efficient tree-based data representation. On the left, we will see the volumetric representation and, on the right, the tree-based representations. These trees can be dynamically expanded to different resolutions and different areas, where every voxel can be subdivided into eight voxels recursively. The size of the map doesn't need to be known beforehand, because mapped volumes aren't initialized until we need to add new measurements. Octrees have been used to adapt occupancy grid mapping from 2D to 3D, introducing probabilistic representation of occupied versus free space.

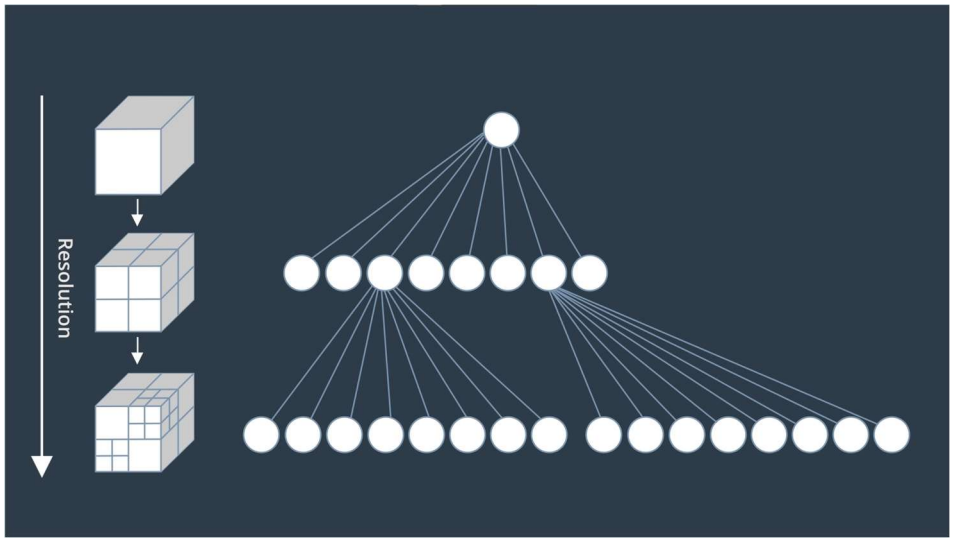


Figure 2.26 3D octrees

- elevation maps.
- multilevel surface maps.

2.3 SLAM

After learning about mapping, we will move along to learn about SLAM. Simultaneous Localization and Mapping. In SLAM, a robot must construct a map of the environment, while simultaneously localizing itself relative to this map. This problem is more challenging than localization and mapping, since neither the map nor the robot’s poses are provided. With noise in the robot's motions and measurements, the map and robot's pose will be uncertain, and the errors in the robot's pose estimates and map will be correlated.

The accuracy of the map depends on the accuracy of the localization and vice versa. SLAM is often called the chicken or the egg problem because the map is needed for localization, and the robot's pose needed for mapping. SLAM truly is a challenge, but it's fundamental to mobile robotics. For robots to be useful to society, they must be able to move in environments that they've never seen before.

SLAM algorithms generally fall into five categories:

- Extended Kalman Filter or EKF.
- Sparse Extended Information Filter or SEIF.
- Extended Information Form or EIF.
- FastSLAM.
- GraphSLAM.

2.3.a SLAM Challenges

We are now going to talk a much more challenging problem of estimating both the map and the robot poses in real-world environments. This problem is called Simultaneous Localization and Mapping or SLAM.

Some roboticists referred to it as CLAM or Concurrent Localization and Mapping. While solving localization problems, we assume they know enough and estimate the robot poses. Whereas, in solving mapping problems, we are provided with the exact robot poses and estimated the map of the environment.

Now, we will combine our knowledge from both localization and mapping to solve the most fundamental problem in robotics. In SLAM, we will map the environment, giving the noisy measurements, and localize the robot relative to its own map giving the controls. This makes it a much more difficult problem than localization or mapping since both the map and the poses are now unknown to us. In real-world environments, we will primarily be faced with SLAM problems, and we will aim to estimate the map and localize the robot.

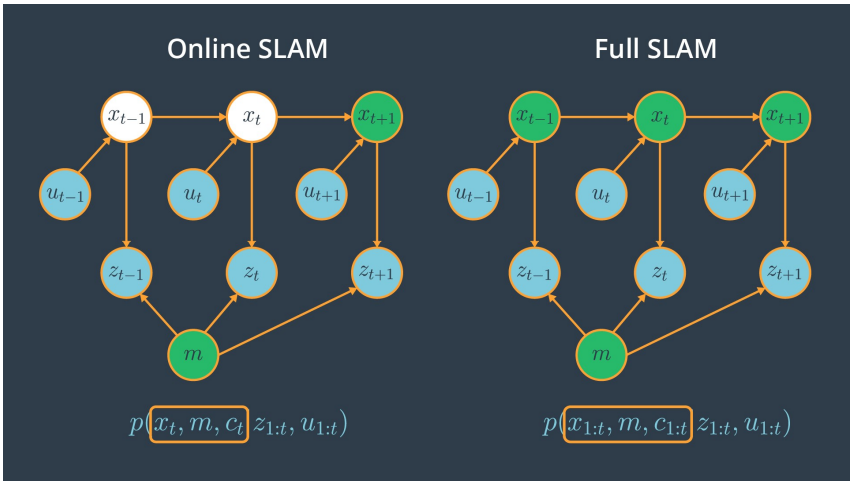


Figure 2.26 Online SLAM vs. Full Slam

Computing the full posterior composed of the robot pose, the map and the correspondence under SLAM poses a big challenge in robotics mainly due to the continuous and discrete portion.

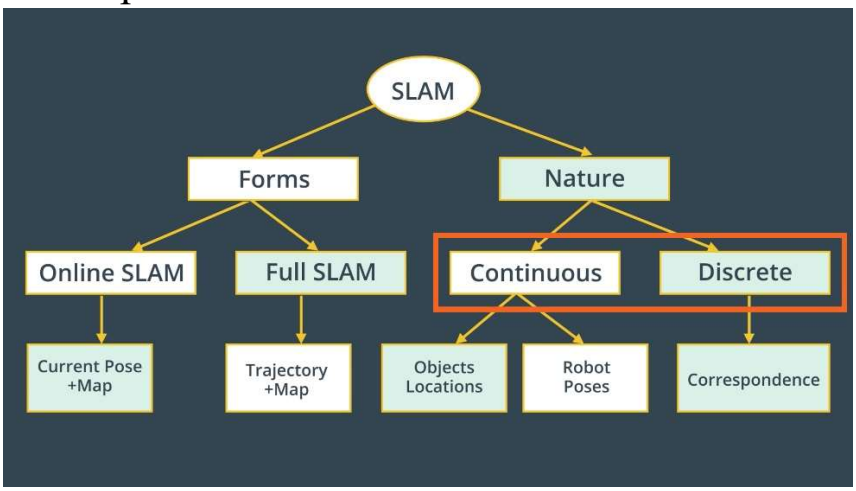


Figure 2.27 SLAM Challenges

- Continuous: The continuous parameter space composed of the robot poses and the location of the objects is highly dimensional. While mapping the environment and localizing itself, the robot will encounter many objects and must keep track of each one of them. Thus, the number of variables will increase with time, and this makes the problem highly dimensional and challenging to compute the posterior.
- Discrete: The discrete parameter space is composed of the corresponding values and is also highly dimensional due to many correspondence variables. Not only that, but the correspondence values also increase exponentially over time since the robot will keep sensing the environment and relating the newly detected objects to the previously detected ones. Even if you assume known correspondence values, the posterior over maps is still highly dimensional as we saw in the mapping lessons.

2.3.d FastSLAM

The FastSLAM algorithm solves the Full SLAM problem with known correspondences.

- Estimating the Trajectory: FastSLAM estimates a posterior over the trajectory using a particle filter approach. This will give an advantage to SLAM to solve the problem of mapping with known poses.
- Estimating the Map: FastSLAM uses a low dimensional Extended Kalman Filter to solve independent features of the map which are modeled with local Gaussian.

The custom approach of representing the posterior with particle filter and Gaussian is known by the Rao-Blackwellized particle filter approach.
We've seen that the FastSLAM algorithm can solve the full SLAM problem with known correspondences. Since FastSLAM uses a particle filter approach to solve SLAM problems, some roboticists consider it a powerful algorithm capable of solving both the Full SLAM and Online SLAM problems.

- FastSLAM estimates the full robot path, and hence it solves the Full SLAM problem.
- On the other hand, each particle in FastSLAM estimates instantaneous poses, and thus FastSLAM also solves the Online SLAM problem.

Now, three different instances of the FastSLAM algorithm exist:

- FastSLAM 1.0: Simple and easy to implement, but this algorithm is known to be inefficient since particle filters generate sample inefficiency.
- FastSLAM 2.0: Overcomes the inefficiency of FastSLAM 1.0 by imposing a different distribution, which results in a low number of particles. Keep in mind that both FastSLAM 1.0 and 2.0 algorithms use a low dimensional Extended Kalman filter to estimate the posterior over the map features.
- Grid-based FastSLAM: The third instance of FastSLAM is really an extension to FastSLAM known as the grid-based FastSLAM algorithm, which adapts FastSLAM to grid maps.

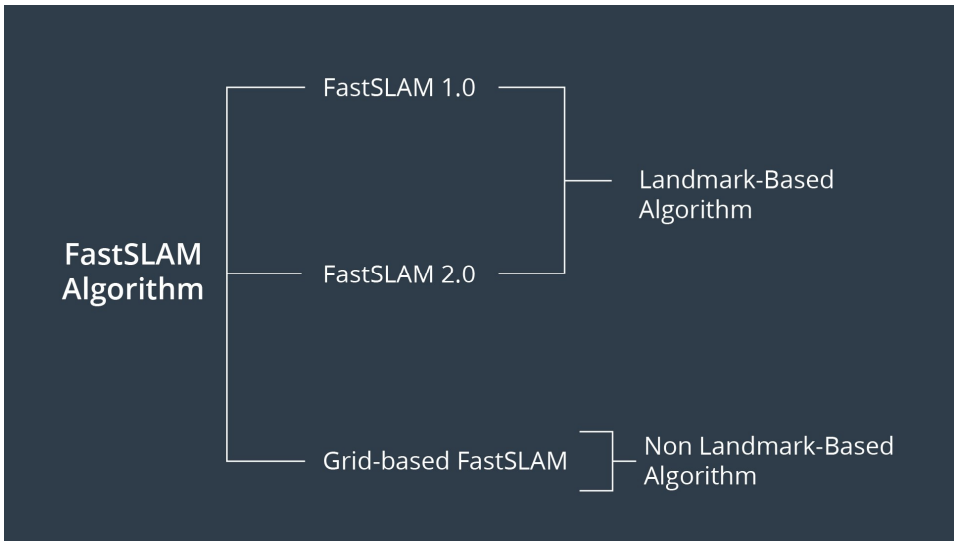


Figure 2.28 Adapting FastSLAM to Grid Maps

To adapt FastSLAM to grid mapping, we need three different techniques:

- Sampling Motion: $p(x_t, {x^k}_{t-1}, u_t)$
  Estimates the current pose given the k-th particle previous pose and the current controls u.
- Map Estimation: $p(m_t|z_t, {x^k}_t, {m^k}_{t-1})$
  Estimates the current map given the current measurements, the current k-th particle pose, and the previous k-th particle map
- Importance Weight: $p(z_t|{x^k}_t, m^k)$
  Estimates the current likelihood of the measurement given the current k-th particle pose and the current k-th particle map.

Grid-based FastSLAM Algorithm (X_{t-1}, u_t, z_t):

$\bar{X}_t = X_t = \emptyset$
for k = 1 to M:
$x_t{}^{[k]} = sample\_motion\_update(u_t, x_{t-1}{}^{[k]})$
$\omega_t{}^{[k]} = measurement\_model\_map(z_t, x_t{}^{[k]}, m_{t-1}{}^{[k]})$
$m_t{}^{[k]} = update\_occupancy\_grid(z_t, x_t{}^{[k]}, m_{t-1}{}^{[k]})$
$\bar{X}_t = \bar{X}_t + \langle x_t{}^{[m]}, \omega_t{}^{[m]}, m_t{}^{[k]} \rangle$
endfor
for k = 1 to M:
draw i with probability $\propto \omega_t{}^{[i]}$
$X_t = x_t{}^{[i]} + m_t{}^{[i]}$
endfor
return X_t

The sampling motion, map estimation and important weight techniques are the essence of the Grid-based FastSLAM algorithm. Grid-based FastSLAM implements them to estimate both the map and the robot trajectory giving the measurements and the control. The Grid-based FastSLAM algorithm which looks very similar to the MCL algorithm with some additional statements concerning the map estimation. In fact, this algorithm is the result of combining the MCL algorithm and the occupancy grid mapping one. As in the MCL algorithm, the Grid-based FastSLAM algorithm is composed of two sections represented by two full loops:

- The first section includes the motion, sensor, and not update steps, and the second one includes the re-sampling process. At each iteration, the algorithm takes the previous belief or pose, the activation command, and the sensor measurements as input. Initially, the hypothetical belief is obtained by randomly generating M particles. each particle implements the three techniques covered earlier to estimate the k particles current pose, likelihood of the measurement, and the map. Each particle begins by implementing the sampling technique in the sample motion model to estimate the current pose of the k particle. Next, in the measurement update step, each particle implements the important Suede technique in the measurement model map function to estimate the current likelihood of the k particle measurement. Moving on to the map update step, here, each particle will implement the map estimation technique into updated occupancy grid map function to estimate the current k particle map. This map estimation problem will be solved under the occupancy grid mapping algorithm. Now, the newly estimated k particle pose map and likelihood of the measurements are all added to the hypothetical belief.
- The second section of the algorithm, re-sampling process happen through re-sampling wheel. Here, particles with measurement values close to the robot's measurement value survives and are redrawn in the next iteration, while the others die. The surviving particles poses end map are added to the system belief. Finally, the algorithm outputs the new belief, and another cycle of iterations starts implementing the newly completed belief, the next motion and the new sensor measurements.

2.3.e GraphSLAM

GraphSLAM is a slam algorithm that solves the full slam problem. This means that the algorithm recovers the entire path and map, instead of just the most recent pose and map. This difference allows it to consider dependencies between current and previous poses. One advantage that can be immediately appreciated is GraphSLAM's improved accuracy over FastSLAM. FastSLAM uses particles to estimate the robot's most likely pose. at any point in time, it's possible that there isn't a particle in the most likely location. In fact, the chances are slim to none especially, in large environments. Since GraphSLAM solves the full slam problem, this means that it can work with all the data at once to find the optimal solution. FastSLAM uses tidbits of information with a finite number of particles, so there is room for error.

The goal of GraphSLAM is to create a graph of all robots poses and features encountered in the environment and find the most likely robot's path and map of the environment. This task can be broken up into two sections:

- The front-end of GraphSLAM: looks at how to construct the graph using the odometry and sensory measurements collected by the robot. This includes interpreting sensory data, creating the graph, and continuing to add nodes and edges to it as the robot traverses the environment. Naturally, the front-end can differ greatly from application to application depending on the desired goal, including accuracy, the sensor used, and other factors. The front-end of GraphSLAM also has the challenge of solving the data association problem. In simpler terms, this means accurately identifying whether features in the environment have been previously seen.
- The back-end of GraphSLAM: is where the magic happens. The input to the back-end is the completed graph with all the constraints. And the output is the most probable configuration of robot poses and map features. The back-end is an optimization process that takes all the constraints and find the system configuration that produces the smallest error. The back end is a lot more consistent across applications.

The front-end and the back end can be completed in succession or can be performed alliteratively, with a back-end feeding an updated graph to the front-end for further processing.