MECEE 4602: Introduction to Robotics, Fall 2019

State Estimation[1]

In this chapter we focus on a problem we have so far assumed solved: knowing the **current state** of a robot, so we can create and follow plans.

What defines the state of a robot is application- and formulation-specific. Consider the case of motion planning for a robot arm, as we have studied it in previous chapters. In its simplest form, "state" can be considered to mean the current position of all the joints, or, in the notation we've used so far, the vector $\boldsymbol{q}$. This is enough to plan and execute trajectories, if we only care about following them closely and reaching the goal. For more complicated problems though (such as controlling the dynamic properties of the trajectory), the state of the robot relevant to our problem could also include velocities and accelerations. We have not spent a lot of time discussing where information about the robot state comes from because joint positions are often given to us by the encoders of the robot, with high accuracy and low latency. If state only comprises positions, then the state of the robot is always reliably provided to us.

The situation is different in the case of a mobile robot. Assume we are still performing motion planning exactly as in the case of the arm, but the "state" of the robot now comprises its position and orientation in the world. It is unlikely we will have a sensor that directly provides us this information, with high accuracy and low latency. Rather, this is computed based on what many sensors are telling us (wheel encoders, IMUs, GPS, vision, etc.). Each of those sensors is noisy, so their collective estimate of position will have noise as well. Informally, state estimation means computing a "best guess" of the robot's state based on a history of sensor readings, and also keeping track of the uncertainty associated with this guess.

1 Dynamic System Model

Formally, we use the vector $\boldsymbol{x}$ to denote the robot state (unfortunately, this overloads our previous nomenclature which used $\boldsymbol{x}$ for end-effector position in Cartesian Space). For a robot arm where state is comprised of joint positions, then $\boldsymbol{x} = \boldsymbol{q}$. If joint velocity is also part of the state, then $\boldsymbol{x} = [\boldsymbol{q}, \dot{\boldsymbol{q}}]^T$. For a mobile robot where state is comprised of position and orientation on the map, then $\boldsymbol{x} = [x, y, \theta]^T$. In general, $\boldsymbol{x}$ can have an arbitrary number of dimensions.

In general, the state $\boldsymbol{x}$ is a function of time. If we model the system in continuous fashion, then the continuous variable t denotes the current time, and $\boldsymbol{x} = \boldsymbol{x}(t)$. If we use discrete modeling (as we will here), then we use the integer variable k to denote the current time step, and $\boldsymbol{x} = \boldsymbol{x}(k)$.

Formally, we model the system as follows:

- $\boldsymbol{x}(k) \in \mathcal{R}^n$ - the robot **state** (at time step k).
- $\boldsymbol{u}(k) \in \mathcal{R}^m$ - the **command** sent to the robot. For a mobile robot, this can comprise for example the velocities commanded to each wheel.
- $\boldsymbol{y}(k) \in \mathcal{R}^p$ - the **sensor measurements** from the robot. Note that, often, you are not directly measuring (all of the) variables that comprise the state; you are measuring a subset of these, or some other variables that are somehow related to the state.

[1] These notes are complied primarily from Choset et al., Principles of Robot Motion, MIT Press, 2005. See this source for complete derivations of all the steps we skip here.

The behavior of our system is governed by the following equations:

$$\boldsymbol{x}(k+1) = f(\boldsymbol{x}(k), \boldsymbol{u}(k)) + \boldsymbol{v}(k) \quad (1)$$

$$\boldsymbol{y}(k) = h(\boldsymbol{x}(k)) + \boldsymbol{w}(k) \quad (2)$$

where $f : \mathcal{R}^n \times \mathcal{R}^m \to \mathcal{R}^n$ is the **dynamics** or **forward model**, and $h : \mathcal{R}^n \to \mathcal{R}^p$ is the **sensor model**. $\boldsymbol{v}$ and $\boldsymbol{w}$ represent random noise added to both the dynamics and the sensor model.

Note that we are making the assumption that the state at the next time step **depends only on the state and action taken at the current time step**, and not on the history of previous time steps. (In other contexts, such as Reinforcement Learning, this is known as the Markovian property.) Care must be chosen to select the state vector such that this property is maintained. Often this means that the velocity of the robot must be made part of the state vector in addition to its position.

2 Linear Models

The formulation above is very general, but also very difficult to use in practice. The functions f and h might simply not be computable for a system, or they might be highly non-linear. We thus begin by studying one of the simplest kinds of systems we can find: fully linear systems.

Assuming a **fully linear system with no noise**, the governing equations become:

$$\boldsymbol{x}(k+1) = \boldsymbol{F}(k)\boldsymbol{x}(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) \quad (3)$$

$$\boldsymbol{y}(k) = \boldsymbol{H}(k)\boldsymbol{x}(k) \quad (4)$$

Here, $\boldsymbol{F}$, $\boldsymbol{G}$ and $\boldsymbol{H}$ are all matrices. $\boldsymbol{F}$ shows how the state evolves over time, $\boldsymbol{G}$ defines how our control input affects the state, and $\boldsymbol{H}$ defines how our sensors capture the state. We assume $\boldsymbol{H}$ is full row rank, but not necessarily square (thus we cannot simply recover $\boldsymbol{x}$ from $\boldsymbol{y}$).

While very few real systems are actually linear, there is still significant value in studying linear systems. We will see applications of this work to non-linear systems after figuring our how we can "solve" linear versions.

2.1 Example: 1D robot

Consider a 1D robot, or a robot moving along a straight line. We assume that the state comprises the position and velocity of the robot:

$$\boldsymbol{x} = [x, v]^T \in \mathcal{R}^2 \quad (5)$$

The input is a force $u \in \mathcal{R}$ applied to the robot (and thus a scalar). We know that the force is proportional to acceleration, which in turn is the derivative of velocity, so

$$v(k+1) = v(k) + \frac{T}{m}u(k) \quad (6)$$

where T is the length of a time step and m is the mass of the robot. Finally, since velocity is the derivative of position, we also know that

$$x(k+1) = x(k) + Tv(k) \quad (7)$$

Combining these in matrix form, we get:

$$\begin{aligned} \boldsymbol{x}(k+1) &= \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} \boldsymbol{x}(k) + \begin{bmatrix} 0 \\ \frac{T}{m} \end{bmatrix} u(k) && (8) \\ &= \boldsymbol{F}\boldsymbol{x}(k) + \boldsymbol{G}u(k) && (9) \end{aligned}$$

Since $\boldsymbol{F}$ and $\boldsymbol{G}$ do not depend on the time step, we have dropped the functional notation for these matrices.

Assuming that our sensor only captures robot velocity, we get the additional relationship:

$$\begin{aligned} y(k) &= \begin{bmatrix} 0 & 1 \end{bmatrix} \boldsymbol{x}(k) && (10) \\ &= \boldsymbol{H}\boldsymbol{x}(k) && (11) \end{aligned}$$

3 State Observers

Consider the following problem statement: assume that the state of the robot is known at the first time step of a sequence (we know what state we initialized the robot in). In addition, we know the command that has been sent to the robot at every time step so far (since we have control over the commands), as well as the sensor data that the robot has seen at every time step. Based on this information, we would like to predict the state of the robot at the next time step.

$$\begin{aligned} \boldsymbol{x}(0) &\quad \texttt{known} && (12) \\ \boldsymbol{u}(0)...\boldsymbol{u}(k) &\quad \texttt{known} && (13) \\ \boldsymbol{y}(0)...\boldsymbol{y}(k) &\quad \texttt{known} && (14) \\ \boldsymbol{x}(k+1) &= ? && (15) \end{aligned}$$

The goal of a state observer is, at all times, to try and estimate the state of the robot. From this point on, we must distinguish between the state as estimated by the observer, and the true state of the system. The former will appear much more often in our equations (since the latter is generally unknowable). Thus, for compactness, from this point on, we will use $\boldsymbol{x}_T(k)$ to denote the true state, and simply $\boldsymbol{x}(k)$ **to denote the estimate of the state as provided by the state observer**.

Using the Markovian property, we immediately notice that a state observer that keeps a running estimate of the state at the current time step does not need to "remember" any information from prior time steps. A state observer, observing a system modeled as the one above, generally works as follows:

- Assume $\boldsymbol{x}(k)$ represents the current estimate of state. This estimate might be precise (identical to the true state), but that is not always the case.
- At every time step, the observer makes a **prediction** of how its estimate will evolve to the next time step. This prediction is made based just on the current command $\boldsymbol{u}(k)$ and the model of system dynamics, without taking sensor data into account.
- Once new sensor data $\boldsymbol{y}(k+1)$ becomes available, it **updates** its prediction to take into account the new information. The updated prediction then becomes the state **estimate** for the next time step, $\boldsymbol{x}(k+1)$ at which point the cycle restarts. Everything except the current state estimate can be forgotten.

In an ideal system, where the prediction component is perfectly accurate, the future state can always be computed based on the current state. However, such systems are rarely encountered in practice: models are rarely perfectly accurate. This could be compensated for with an ideal set of sensors, that provide enough information to perfectly reconstruct the state; that is also a rarely encountered situation (illustrated in our case by the fact that we can not simply invert $\boldsymbol{H}$ to obtain $\boldsymbol{x}$). Thus, in practice, the predicted state $\boldsymbol{x}(k+1)$, and the sensor data we get once we arrive at time step $k+1$, are not in perfect agreement. We need the update step to reconcile them.

3.1 State Observer for Linear Systems

In the linear case, assume that $\hat{\boldsymbol{x}}$ is our prediction of the state at the next time step obtained via our model:

$$\hat{\boldsymbol{x}}(k+1) = \boldsymbol{F}\boldsymbol{x}(k) + \boldsymbol{G}\boldsymbol{u}(k) \tag{16}$$

The problem arises if this prediction is in conflict with sensor data we actually get at time $k+1$:

$$\boldsymbol{y}(k+1) \neq \boldsymbol{H}\hat{\boldsymbol{x}}(k+1) \tag{17}$$

Assuming that $\boldsymbol{H}$ is not full column rank, there is a range of possible $\boldsymbol{x}(k+1)$ which could all produce the measurement $\boldsymbol{y}(k+1)$. Which of those should we use?

An intuitive answer is to use the value of the state $\boldsymbol{x}(k+1)$ that produces the expected sensor values $\boldsymbol{y}(k+1)$, and is also closest to our prediction $\hat{\boldsymbol{x}}(k+1)$. In other words, we are looking to add the smallest possible offset $\Delta\boldsymbol{x}$ to our prediction such that it matches sensor values:

$$\begin{aligned} \boldsymbol{y}(k+1) &= \boldsymbol{H}\left[\hat{\boldsymbol{x}}(k+1) + \Delta\boldsymbol{x}\right] && (18)\\ \boldsymbol{H}\Delta\boldsymbol{x} &= \boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1) && (19) \end{aligned}$$

We recall that a full row rank matrix will accept a right-inverse, and that using the right inverse will give us the lowest norm solution to a linear system. Applying this we get:

$$\Delta\boldsymbol{x} = \boldsymbol{H}^T(\boldsymbol{H}\boldsymbol{H}^T)^{-1}\left[\boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1)\right] \tag{20}$$

and thus we can update our prediction to get the next state estimate as:

$$\begin{aligned} \boldsymbol{x}(k+1) &= \hat{\boldsymbol{x}}(k+1) + \Delta\boldsymbol{x} && (21)\\ &= \boldsymbol{F}\boldsymbol{x}(k) + \boldsymbol{G}\boldsymbol{u}(k) + \boldsymbol{H}^T(\boldsymbol{H}\boldsymbol{H}^T)^{-1}\left[\boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1)\right] && (22) \end{aligned}$$

Note that here we are fully trusting our sensor data. We are simply using the prediction to help us choose from between the multiple possible explanations for the sensor data. In contrast, we do not fully trust our prediction and are willing to modify it to match sensor response. In a sense, we are saying that the sensor data is perfect (albeit incomplete) and the model we use for prediction is not an entirely correct depiction of our system.

4 Probabilistic Reasoning

One of the most important features of state estimation is its ability to reason about uncertainty. Many robotic systems in the real world inevitably have a measure of uncertainty. The best results are obtained when this uncertainty is explicitly modeled. This allows us to keep track of it, understand the sources of the uncertainty, and, if all else fails, know when uncertainty is too high to take an action.

4.1 Probability Density Functions

At the simplest level, reasoning about uncertainty means replacing exact variable values with **Probability Density Functions** (PDF). For a continuous variable x, f_x is the PDF of x if the following is true:

$$\texttt{Pr}(a \leq x \leq b) = \int_{x=a}^{b} f_x(x)dx \tag{23}$$

where $\texttt{Pr}(a \leq x \leq b)$ is the probability that the value of x falls in the $[a, b]$ interval.

When using this type of probabilistic reasoning, we accept that **we never know the true value of a variable, we just know its PDF**. Of course, if we are highly confident that x is equal to, for example, $\bar{x}$, then the PDF has a narrow spike centered on $\bar{x}$ and is close to 0 elsewhere. Still, we must give up reasoning based on exact variable values.

4.2 Gaussian Distribution

A very common type of PDF, often used in practice, is the one that defines the **Gaussian distribution**. A variable x is said to follow a Gaussian distribution if its PDF is defined as:

$$f_x(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\bar{x})^2}{2\sigma^2}} \tag{24}$$

where $\bar{x}$ is the mean of the distribution, and σ^2 is the variance (σ is also called the standard deviation).

When a variable obeys this distribution, we replace the idea of knowing the value of the variable with knowing, at any given time, the mean and variance of its PDF, $\bar{x}$ and σ^2. Informally, $\bar{x}$ tells us what we think the value of the variable is, and σ^2 tells us how likely x is to deviate from this guess, which we can use as a measure of confidence in our guess.

If we are dealing with a multidimensonal variable $\boldsymbol{x} \in \mathcal{R}^n$, we use a multi-variate Gaussian distribution:

$$f_x(\boldsymbol{x}) = \frac{1}{\sqrt{2\pi|P|}} e^{-\frac{1}{2}(\boldsymbol{x}-\bar{\boldsymbol{x}})P^{-1}(\boldsymbol{x}-\bar{\boldsymbol{x}})^T} \tag{25}$$

where $\bar{\boldsymbol{x}}$ is the mean vector and P is the covariance matrix. $\bar{\boldsymbol{x}}$ serves a similar role to the one-dimensional case, giving us a place where the distribution is centered, or a guess. The covariance matrix tells us how likely each individual variable in $\boldsymbol{x}$ is to deviate from the guess, and also if different variables in $\boldsymbol{x}$ are independent of each other or are somehow related.

As in the one-dimensional case, probabilistic reasoning means that, instead of attempting to know the one true value of the multi-dimensional variable $\boldsymbol{x}$, we attempt to know, at any given time, the mean and covariance of its PDF, or $\bar{\boldsymbol{x}}$ and P.

4.3 Expected Value and Variance

Here we formalize the notions of mean, variance and covariance that we have already introduced above. The expected value of a variable is defined as:

$$E_x = \int_{\boldsymbol{x}\in\mathcal{R}^n} \boldsymbol{x} f_x(\boldsymbol{x})dx \tag{26}$$

Informally, if we sample $\boldsymbol{x}$ from its distribution multiple times, and we average all of those samples, we expect that the average (or mean) will approach the value of E_x as the number of samples grows. For this reason, the expected value is often called the mean of the distribution. It is often denoted

by $\bar{\boldsymbol{x}}$. This is the same name we have used before for the mean of the Gaussian distribution, but this is not a coincidence: one can check that the expected value of a Gaussian distribution is indeed its mean.

The variance of a scalar variable x is defined as the expected value of $(x - \bar{x})^2$. It tells us how much x is expected to deviate from its mean $\bar{x}$, in absolute terms. Again, we can check that the variance of the Gaussian distribution is indeed equal to σ^2.

Finally, the covariance of a vector $\boldsymbol{x} \in \mathcal{R}^n$ is defined as the expected value of $(\boldsymbol{x} - \bar{\boldsymbol{x}})(\boldsymbol{x} - \bar{\boldsymbol{x}})^T$. Denoted here by P, it is an $n \times n$ matrix. As described above, every diagonal entry P_{ii} is equal to the variance of the i-th entry in $\boldsymbol{x}$, denoted by $\boldsymbol{x}_i$. A non-diagonal entry P_{ij} tells us the covariance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$. $P_{ij} = 0$ means that $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ are independent of each other.

5 Probabilistic State Observation

We now return to the state observation problem, with a probabilistic perspective. This means that we are no longer trying to estimate the exact state, but rather its PDF. Assuming a Gaussian Distribution, the PDF is determined by its mean $\bar{\boldsymbol{x}}$ and covariance matrix $\boldsymbol{P}$. To simplify notation, we will drop the bar on $\bar{\boldsymbol{x}}$. **In everything that follows, $\boldsymbol{x}$ refers to the mean of the state PDF, while $\boldsymbol{P}$ denotes its covariance.**

As mentioned before, we have given up on the idea of knowing the "true" value of the state $\boldsymbol{x}_T$. $\boldsymbol{x}$ and $\boldsymbol{P}$ form our probabilistic estimate of it. All we know is that, if we do our computations correctly, the expected value of $(\boldsymbol{x}_T - \boldsymbol{x})^2$ is equal to $\boldsymbol{P}$

Compared to the initial (non-probabilistic) formulation, the main complication is that now we also have to keep track of $\boldsymbol{P}$, not just $\boldsymbol{x}$. At any time, **the covariance can be considered a measure of how confident we are in our state estimate**. It is itself a function of time, so $\boldsymbol{P} = \boldsymbol{P}(k)$. Given $\boldsymbol{x}(k), \boldsymbol{P}(k)$ and $\boldsymbol{y}(k+1)$, we must come up with the best estimate for $\boldsymbol{x}(k+1)$ and $\boldsymbol{P}(k+1)$ given our linear model. Furthermore, the probabilistic framework also allows us to formally include the possibility of noise in our model:

$$\boldsymbol{x}_T(k+1) = \boldsymbol{F}(k)\boldsymbol{x}_T(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) + v(k) \tag{27}$$

where $v(k)$ is assumed to be noise sampled from a Gaussian distribution with zero mean ("white noise") and covariance matrix $\boldsymbol{V}(k)$.

Again, we start with a "prediction" $\hat{\boldsymbol{x}}(k+1)$ based on our current estimate. The expected value for the noise is zero, hence:

$$\hat{\boldsymbol{x}}(k+1) = \boldsymbol{F}(k)\boldsymbol{x}(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) \tag{28}$$

Our "predicted" covariance matrix $\hat{\boldsymbol{P}}(k+1)$ simply follows the definition of covariance looking at how far the true state $\boldsymbol{x}(k+1)$ is expected to stray from our prediction $\hat{\boldsymbol{x}}(k+1)$. Here we need to differentiate between the true value of the state $\boldsymbol{x}_T$ (which we never actually know) and our estimate of it $\boldsymbol{x}$.

$$\hat{\boldsymbol{P}}(k+1) = E[\ (\boldsymbol{x}_T(k+1) - \hat{\boldsymbol{x}}(k+1))\ (\boldsymbol{x}_T(k+1) - \hat{\boldsymbol{x}}(k+1))^T\ ] \tag{29}$$

We expand the terms inside the expectation, taking advantage of the fact that the expectation operator is linear, the expected value of the noise signal is 0, as well as the definition of covariance at time k:

$$E[\boldsymbol{v}(k)] = 0 \tag{30}$$

$$E[\boldsymbol{v}(k)\boldsymbol{v}(k)^T] = \boldsymbol{V}(k) \tag{31}$$

$$E[(\boldsymbol{x}_T(k) - \boldsymbol{x}(k))\,(\boldsymbol{x}_T(k) - \boldsymbol{x}(k))^T] = \boldsymbol{P}(k) \tag{32}$$

Using all of these equalities, we finally arrive at:

$$\hat{\boldsymbol{P}}(k+1) = \boldsymbol{F}(k)\boldsymbol{P}(k)\boldsymbol{F}(k)^T + \boldsymbol{V}(k) \tag{33}$$

We now have "predictions" for both the state and its covariance at the next time step, denoted by $\hat{\boldsymbol{x}}(k+1)$ and $\hat{\boldsymbol{P}}(k+1)$ respectively. We must still "update" these predictions based on sensor information.

Again, the problem arises if our predicted state does not agree with sensor data. Again, we would like to modify the prediction by some $\Delta\boldsymbol{x}$ to bring it inline with sensor information. Previously, any change in prediction was just as good as any other, so we simply looked for the **smallest** $\Delta\boldsymbol{x}$ that would achieve this. Now however we have a measure of confidence in our prediction, namely $\hat{\boldsymbol{P}}$. We thus want the **most likely** state estimate that also agrees with sensor data. Basically, we want to maximize the probability of $\hat{\boldsymbol{x}}+\Delta\boldsymbol{x}$ while still agreeing with sensor data. This is equivalent to:

$$\texttt{minimize} \quad \Delta\boldsymbol{x}^T\hat{\boldsymbol{P}}(k+1)\Delta\boldsymbol{x} \tag{34}$$

$$\texttt{subject to} \quad \boldsymbol{y}(k+1) = \boldsymbol{H}[\hat{\boldsymbol{x}}(k+1) + \Delta\boldsymbol{x}] \tag{35}$$

To get this result, we will use the "$\boldsymbol{P}$-weighted" pseudo-inverse of $\boldsymbol{H}$ (the exact derivation of this result is beyond the scope of this handout). Since, in all the derivations that follow, we mostly ever use $\boldsymbol{H}(k+1)$, we will refer to it in shorthand as $\boldsymbol{H}$.

$$\Delta\boldsymbol{x} = \hat{\boldsymbol{P}}\boldsymbol{H}^T(\boldsymbol{H}\hat{\boldsymbol{P}}\boldsymbol{H}^T)^{-1}\left[\boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1)\right] \tag{36}$$

Note that if $\boldsymbol{P} = \boldsymbol{I}$ (we do not have any meaningful covariance information) this reduces to the case above with no probabilistic reasoning.

Now we can update our prediction of the state simply as:

$$\boldsymbol{x}(k+1) = \hat{\boldsymbol{x}}(k+1) + \Delta\boldsymbol{x} \tag{37}$$

To simplify notation, we use the following symbols:

$$\nu = \boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1) \tag{38}$$

$$\boldsymbol{R} = \hat{\boldsymbol{P}}\boldsymbol{H}^T(\boldsymbol{H}\hat{\boldsymbol{P}}\boldsymbol{H}^T)^{-1} \tag{39}$$

Here, ν is sometimes called the "innovation": how much we "new" sensor data we are seeing that deviates from our prediction. Using this notation, the state update step is:

$$\boldsymbol{x}(k+1) = \hat{\boldsymbol{x}}(k+1) + \boldsymbol{R}\nu \tag{40}$$

There is one step left: to update our prediction of the covariance $\hat{\boldsymbol{P}}(k+1)$. The complete derivation is beyond the scope of this handout, but this update step is:

$$\boldsymbol{P}(k+1) = \hat{\boldsymbol{P}}(k+1) - \boldsymbol{R}\boldsymbol{H}\hat{\boldsymbol{P}}(k+1) \tag{41}$$

To summarize, we have two steps to obtain our next estimate of the state and its covariance. First we generate our predictions based on our model:

$$\hat{\boldsymbol{x}}(k+1) = \boldsymbol{F}(k)\boldsymbol{x}(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) \tag{42}$$

$$\hat{\boldsymbol{P}}(k+1) = \boldsymbol{F}(k)\boldsymbol{P}(k)\boldsymbol{F}(k)^T + \boldsymbol{V}(k) \tag{43}$$

and then we update them based on new sensor data:

$$\begin{aligned} \boldsymbol{x}(k+1) &= \hat{\boldsymbol{x}}(k+1) + \boldsymbol{R}\nu & (44)\\ \boldsymbol{P}(k+1) &= \hat{\boldsymbol{P}}(k+1) - \boldsymbol{R}\boldsymbol{H}\hat{\boldsymbol{P}}(k+1) & (45)\\ \nu &= \boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1) & (46)\\ \boldsymbol{R} &= \hat{\boldsymbol{P}}\boldsymbol{H}^T(\boldsymbol{H}\hat{\boldsymbol{P}}\boldsymbol{H}^T)^{-1} & (47) \end{aligned}$$

This model also has the role of "filtering" the noise associated with our model, but keeping track of how it affects the uncertainty in our state estimation. However, we are still fully trusting our sensor data. This model is of limited use in practice; rather, it is a stepping stone to the more complete model presented next.

6 The Kalman Filter

To complete our model, assume now that our sensor information is also affected by white noise $\boldsymbol{w}(k)$ with covariance $\boldsymbol{W}$:

$$\begin{aligned} \boldsymbol{x}(k+1) &= \boldsymbol{F}(k)\boldsymbol{x}(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) + \boldsymbol{v}(k) & (48)\\ \boldsymbol{y}(k) &= \boldsymbol{H}(k)\boldsymbol{x}(k) + \boldsymbol{w}(k) & (49) \end{aligned}$$

Since the only difference from the previous case is that we have assumed noisy measurement, the prediction step stays unchanged:

$$\begin{aligned} \hat{\boldsymbol{x}}(k+1) &= \boldsymbol{F}(k)\boldsymbol{x}(k) + \boldsymbol{G}(k)\boldsymbol{u}(k) & (50)\\ \hat{\boldsymbol{P}}(k+1) &= \boldsymbol{F}(k)\boldsymbol{P}(k)\boldsymbol{F}(k)^T + \boldsymbol{V}(k) & (51) \end{aligned}$$

However, the update has to be different, since we no longer trust our sensor data completely. Once again, a complete derivation of the update step is beyond our scope here; informally it is based on merging the uncertainty of our prediction with the uncertainty of our sensor information to compute the "most likely" sensor data. The update is then performed based on this "most likely" sensor data. The final expression for the update step is:

$$\begin{aligned} \boldsymbol{x}(k+1) &= \hat{\boldsymbol{x}}(k+1) + \boldsymbol{R}\nu & (52)\\ \boldsymbol{P}(k+1) &= \hat{\boldsymbol{P}}(k+1) - \boldsymbol{R}\boldsymbol{H}\hat{\boldsymbol{P}}(k+1) & (53)\\ \nu &= \boldsymbol{y}(k+1) - \boldsymbol{H}\hat{\boldsymbol{x}}(k+1) & (54)\\ \boldsymbol{S} &= \boldsymbol{H}\hat{\boldsymbol{P}}(k+1)\boldsymbol{H}^T + \boldsymbol{W}(k+1) & (55)\\ \boldsymbol{R} &= \hat{\boldsymbol{P}}(k+1)\boldsymbol{H}^T\boldsymbol{S}^{-1} & (56) \end{aligned}$$

Together, these equations define the **Kalman Filter**. This filter has important theoretical properties: if the assumptions are met (linear system, white noise), it provides the best possible estimate of the state. Furthermore, it also has good behavior in practice.

Fundamentally, the role of the Kalman Filter is to "merge" information from multiple sources (our own model, various sensors), based on the confidence in each of these sources (expressed as a covariance matrix) and to give us its best guess of the state, as well as a measure of confidence in this guess.

7 The Extended Kalman Filter

The Kalman Filter, as derived above, applied to linear systems. However, many real-life systems are not linear. How do we perform state estimation in such cases? Do we need fundamentally different methods that do not assume linearity? Such methods do exist, but before using them we notice that we can still apply the Kalman filter if we simply **linearize** the system around its current state.

Consider a general, non-linear system defined as:

$$\boldsymbol{x}(k+1) = f(\boldsymbol{x}(k), \boldsymbol{u}(k)) + \boldsymbol{v}(k) \tag{57}$$

$$\boldsymbol{y}(k) = h(\boldsymbol{x}(k)) + \boldsymbol{w}(k) \tag{58}$$

Note that we have replaced the matrices $\boldsymbol{F}$, $\boldsymbol{G}$ and $\boldsymbol{H}$ by the general (and potentially non-linear) functions $f : \mathcal{R}^n \times \mathcal{R}^m \rightarrow \mathcal{R}^n$ and $h : \mathcal{R}^n \rightarrow \mathcal{R}^p$. The noise models $\boldsymbol{v}$ and $\boldsymbol{w}$ remain the same (normal distribution, zero mean, known covariance).

We can still carry out the prediction of the state estimate as before, using the new function that defines the dynamics of our system:

$$\hat{\boldsymbol{x}}(k+1) = f(\boldsymbol{x}(k), \boldsymbol{u}(k)) \tag{59}$$

For the prediction of the covariance, we need to linearize the function f around the old state estimate $\boldsymbol{x}(k)$:

$$\boldsymbol{F}(k) = \frac{\partial f}{\partial \boldsymbol{x}}\Big|_{x=x(k)} \tag{60}$$

Recall that, just as in the case of the Jacobian, the derivative of a multivariate function with multidimensional output yields a matrix, in this case $\boldsymbol{F} \in \mathcal{R}^{n \times n}$.

We can now use the same expression as before for the prediction of the covariance:

$$\hat{\boldsymbol{P}}(k+1) = \boldsymbol{F}(k)\boldsymbol{P}(k)\boldsymbol{F}(k)^T + \boldsymbol{V}(k) \tag{61}$$

For the update step, we must also linearize our new sensor model h. Note that we carry out this linearization around the new prediction of the state estimate $\hat{\boldsymbol{x}}(k+1)$:

$$\boldsymbol{H}(k+1) = \frac{\partial h}{\partial \boldsymbol{x}}\Big|_{x=\hat{x}(k+1)} \tag{62}$$

This yields the matrix $\boldsymbol{H} \in \mathcal{R}^{p \times n}$.

Armed with this linearized sensor model, we can now carry out the update step exactly as we did before, using Eqs. (52)-(56).

8 Particle Filters

With the EKF, we have removed the assumption on non-linearity on our system. Still, many limitations remain that are "baked in" the Kalman Filter framework. Linearization around the current state estimate could be a poor approximation of true non-linear system dynamics and sensor models. More importantly, the Kalman Filter framework uses a **model of uncertainty based on normal distributions**: we model our state estimate as a normal distribution, and we strive to compute its mean $\boldsymbol{x}$ and covariance $\boldsymbol{P}$. However, this model is highly restrictive in practical examples.

Consider a mobile robot that has two possible guesses for its location, referred to as $\boldsymbol{x}_0$ and $\boldsymbol{x}_1$. The robot is confident that one of those must be true (with high accuracy), but does now know which one. Such a case if often referred to as a **multimodal** distribution. A single normal distribution can not capture the PDF of such a state estimate. A mixture of multiple normal distributions could, but the PDF becomes that much more complicated; in addition, one can then encounter practical situations that are difficult to model even with mixtures of multiple Gaussians.

Can we keep track of our uncertainty in state estimation without committing to a particular model of the PDF? Yes, by simply keeping track of many, many hypotheses of our state. These hypotheses are called "particles", and they evolve based on how much uncertainty we have in our system. Each particle encodes one possible state of the system (with no measure of uncertainty). If many of our particles end up in the same region in state space, than that region has a high likelihood of being the "correct" one.

At a high level, using a particle filter can be summarized as:

- A particle is defined as one hypothesis of the state of the system.
- We start of with many particles, distributed based on how much information we have to begin with.
- At each time step, each particle "evolves" based on our (potentially non-linear) model of dynamics.
- When we get new sensor data, we process our entire particle set such that particles representing very unlikely states (based on sensor data) tend to be eliminated.
- At any given time, the distribution of our particles tells us where the most likely "true" state can be found.

To formalize, consider a system for which we have the following model of dynamics:

$$\boldsymbol{x}(k+1) = f(\boldsymbol{x}(k), \boldsymbol{u}(k)) \tag{63}$$

where f is a potentially non-linear function. Note that f might incorporate some model of noise as well.

At all times, a particle filter maintains a set of N particles $X(k) = \boldsymbol{x}_j(k), j \in 1..N$. At the beginning, $X(0)$ is initialized based on our initial knowledge of the system. If we know exactly our starting state $\boldsymbol{x}_s$, then all particles are initialized as $\boldsymbol{x}_j(0) = \boldsymbol{x}_s$. If we know nothing about the start state, then all $\boldsymbol{x}_j(0)$ are drawn randomly from a uniform distribution that covers our state space.

As we move from time step k to time step $k+1$, the job of the particle filter is to compute $X(k+1)$, based on the previous set $X(k)$, the command $\boldsymbol{u}(k)$, and the new sensor data $\boldsymbol{y}(k+1)$. More concretely, processing a single time step using a particle filter takes place as follows:

1. Each particle $\boldsymbol{x}_j$ evolves based on our models of dynamics:

$$\hat{\boldsymbol{x}}_j(k+1) = f(\boldsymbol{x}_j(k), \boldsymbol{u}(k)) \tag{64}$$

   We use the $\hat{}$ notation $\hat{\boldsymbol{x}}_j(k+1)$ to suggest that so far we have only done the equivalent of a "prediction" step from the previous framework. We have not yet taken into account any new sensor data, thus the set of particles $\hat{X}(k+1) = \hat{\boldsymbol{x}}_j(k+1), j \in 1..N$ is not yet the set of particles we will produce at the end of this time step.

2. When we receive new sensor data $\boldsymbol{y}(k+1)$, we use it to decide how "likely" each of our predicted particles is. A key aspect here is that we actually compute what intuitively would seem like an inverse measure, of how likely the new sensor data is for each given particle:

$$w_j = P(\boldsymbol{y}(k+1)|\hat{\boldsymbol{x}}_j(k+1)) \tag{65}$$

where $P(\boldsymbol{y}(k+1)|\hat{\boldsymbol{x}}_j(k+1))$ is the probability of seeing sensor data $\boldsymbol{y}(k+1)$ assuming that the true state is $\hat{\boldsymbol{x}}_j(k+1)$. We will use the "weight" w_j as a measure of how much confidence we now have in particle $\hat{\boldsymbol{x}}_j(k+1)$ now that we have seen new sensor data.

3. We prepare a new particle set $X(k+1)$ (this will be our output at the end of this step) and we initialize it as empty.

4. We hold a "lottery" to select a particle from $\hat{X}(k+1)$. This must be done such that the chances of any particle $\hat{\boldsymbol{x}}_j(k+1)$ of being selected are proportional to its weight w_j. We take the particle selected according to this lottery and place a copy of it in $X(k+1)$. We do not remove the original from $\hat{X}(k+1)$, so that it can be drawn again in the future.

5. We repeat the previous step until $X(k+1)$ has N particles in it.

At any given time, if we must choose the one most likely true state for our robot, we can do it by finding the region of space with the highest density of particles.

We see now that the particle filter follows the same "prediction/update" process that the Kalman Filter does. Step 1 is the prediction (based on the model), whereas Steps 2-5 represent the update (based on new sensor data).

The advantages of the particle filter include:

- Can represent any PDF for the state estimate. In fact, we are never explicitly representing the PDF (analytically). For the Kalman Filter, we assumed that the PDF was Gaussian, and we were trying just to compute the parameters of the Gaussian (mean and covariance). Here, we have no analytical model of the PDF. Formally, our particles represent samples drawn from the PDF - we thus represent the PDF through its samples, rather than the parameters of its analytical formulation. The particle filter is thus said to belong to the family of **non-parametric** state estimation methods.

- We can use any non-linear model of dynamics f, explicitly and without any approximation (such as linearization). Furthermore, we can easily model highly non-linear effects, such as collision with obstacles. The fact that a robot is not allows to go "inside" an obstacle is very difficult to model analytically. With a particle filter, we simply remove all the particles we find to be inside obstacles.

The most significant difficulty of using a particle filter shows itself in Step 2. We need to compute a measure of how likely we are to be seeing our sensor data, assuming a given system state. This is often referred to as a **sensor model** - it is the equivalent of Eq. (49) in the Kalman Filter. If an actual sensor model is difficult to compute, we can try to make up an approximation of Eq. (65): informally, the larger the difference between expected and received sensor data, the lower the computed probability should be. The overall performance of the particle filter will depend on the sensor model, which, in practice, must often be tuned until we get the desired behavior.

9 Bayesian Filtering

Both the Kalman Filter and the particle filter are special cases of the general family of Bayesian filters. These filters are named after Bayes' rule, a simple but fundamental rule in probabilistic reasoning:

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)} \tag{66}$$

This equation expresses $P(a|b)$, or the **conditional probability** of a being true given that we know b is true, as a function of the reverse conditional probability $P(b|a)$, or the probability of b being true if we know that a is true.

Fundamentally, any state estimator is looking the estimate the following:

$$P(\boldsymbol{x}(k+1)) = P(\ \boldsymbol{x}(k+1) \mid \boldsymbol{u}(0:k), \boldsymbol{y}(0:k+1)\ ) \tag{67}$$

In other words, we are looking to compute the PDF of the current state, given our entire history of commands and sensor data (and assuming, as a boundary condition, that $P(x(0))$ is known).

Through the application of Bayes' rule, this process is broken down into two steps (the complete derivation of which is beyond the scope of this handout):

1. Prediction:
$$\hat{P}(\boldsymbol{x}(k+1)) = \int P(\boldsymbol{x}(k+1)|\boldsymbol{x}(k), \boldsymbol{u}(k))P(\boldsymbol{x}(k))dx \tag{68}$$
2. Update:
$$P(\boldsymbol{x}(k+1)) = \eta P(\boldsymbol{y}(k+1)|\boldsymbol{x}(k+1))\hat{P}(\boldsymbol{x}(k+1)) \tag{69}$$

We can see that these steps compute $P(\boldsymbol{x}(k+1))$ based on $P(\boldsymbol{x}(k))$, $\boldsymbol{u}(k)$ and $\boldsymbol{y}(k+1)$. This technique is often referred to as recursive Bayesian filtering: these steps are applied in **recursive** fashion for as long as the system evolves.

Some of the terms in these equations have specific names:

- $P(\boldsymbol{x}(k))$ is the **prior** distribution, or the distribution of our state estimate at the beginning of the time step (the input to the time step algorithm).
- $P(\boldsymbol{x}(k+1))$ is the **posterior** distribution, or the distribution of our state estimate at the end of the time step (the output of the time step algorithm).
- $P(\boldsymbol{x}(k+1)|\boldsymbol{x}(k), \boldsymbol{u}(k))$ is referred to as the model of system dynamics, or the motion model of the system.
- $P(\boldsymbol{y}(k+1)|\boldsymbol{x}(k+1))$ is the sensor model or the measurement model of the system.
- η is the normalization constant, whose role is to ensure that probabilities integrate to 1 over their entire domain.

All the filters we've looked at so far simply make different choices of how to represent these various models. In the Kalman Filter, the prior and posterior are both normal distributions, and the dynamics and measurement models are both linear with white noise. In the particle filter, the prior and posterior are represented as particle sets (and not modeled analytically), the dynamic model can be any non-linear function, and a good measurement model depends on the skill of the person applying the filter to a specific problem. The most important aspect is still the ability to reason about the problem in probabilistic fashion, and explicitly consider the uncertainty inherent in any real-life robotics application.