ARTIFICIAL INTELLIGENCE FOR ROBOTICS

Simulation of a Self Driving Car

ABSTRACT

An autonomous car, also known as a robotic car, or informally as driverless or self-driving, is an autonomous vehicle capable of fulfilling the human transportation capabilities of a traditional car. As an autonomous vehicle, it is capable of sensing its environment and navigating without human input. Robotic cars exist mainly as prototypes, but are likely to become more widespread in the near future.

Autonomous vehicles sense their surroundings with such techniques as radar, lidar, GPS, and computer vision. Advanced control systems interpret sensory information to identify appropriate navigation paths, as well as obstacles and relevant signage. Some autonomous vehicles update their maps based on sensory input, allowing them to find their way through uncharted environments.

BENEFITS

Anticipated benefits of autonomous cars include:

1. Fewer traffic collisions, due to an autonomous system's increased reliability and decreased reaction time compared to human drivers.

2. Increased roadway capacity and reduced traffic congestion due to reduced need for safety gaps and the ability to better manage traffic flow.

3. Relief of vehicle occupants from driving and navigation chores.

4. Higher speed limit for autonomous cars

5. Removal of constraints on occupants' state – in an autonomous car, it would not matter if the occupants were under age, over age, blind, distracted, intoxicated, or otherwise impaired.

6. Alleviation of parking scarcity as cars could drop off passengers, park far away where space is not scarce, and return as needed to pick up passengers.

7.Elimination of redundant passengers – humans are not required to take the car anywhere, as the robotic car can drive independently to wherever it is required. This would be especially relevant to trucks, taxis and car-sharing services.

8. Reduction of space required for vehicle parking.

9. Reduction in the need for traffic police and vehicle insurance.

10. Reduction of physical road signage – autonomous cars could receive necessary communication electronically (although physical signs may still be required for any human drivers).

11. Improved fuel efficiency.

Motivation

According to the World Health Organization , more than 1.2 million lives are lost every year in road traffic accidents. We believe our technology has the potential to cut that number, perhaps by as much as half. In 2010, there were approximately six million vehicle crashes leading to 32,788 traffic deaths, or approximately 15 deaths per 100,000 people. Vehicle crashes are the leading cause of death for Americans aged 4–34. And of the 6 million crashes, 93 percent are attributable to human error. The economic impact of crashes is also significant. More than 2.3 million adult drivers and passengers were treated in U.S. emergency rooms in 2009.

According to research from the American Automobile Association (AAA), traffic crashes cost Americans $299.5 billion annually.The goal is not merely to make self-driving vehicles as “safe” as human drivers, who, as the evidence shows, are not very safe at all. The goal is to develop “crash-less” cars.

Even aging boomers are increasingly distracted by cellphones and other gadgets; they, too, will soon move beyond safe driving age. Among the boomers we interviewed, even those who owned premium cars said they would willingly give up driving to work in exchange for an easier commute.

Self-driving cars open up new possibilities and new markets, and not just for those who are legally eligible to drive, but also for younger people, older people, and those with disabilities .For them self-driving promises greater freedom and mobility and greater control over their lives.

In terms of time efficiency, the U.S Department of Transportation estimates that people spend on average 52 minutes each working day commuting.

The automotive industry has been—and continues to be—a critical component of the U.S economy, employing 1.7 million people (across manufacturers, suppliers, and dealers) and providing $500 billion in annual compensation, as well as accounting for approximately 3 percent of GDP.

But mobility is increasingly expensive and inefficient. First, of course, is the total cost of vehicle ownership, which can bring the price of a $21,000 car driven an average of 15,000 miles per year to more than $40,000 over five years—for a machine that sits unused on average, almost 22 hours out of every day.

We also pay heavily to build and maintain our roads. The U.S. Department of Transportation (USDOT) estimates that new construction of four-lane highways in an urban area costs between $8 million and $12 million per mile. Even resurfacing that road, at an estimated $1.25 million per mile, can be daunting for cash-strapped governments.

The average American commuter now spends 250 hours a year behind the wheel of a vehicle; whether the value of that time is measured in lost productivity, lost time pursuing other interests, or lost serenity, the cost is high.

Today, those commuters inch along during rush hour traffic; they drive in circles around city streets looking for parking spaces; and, according to a report published by the MIT Media Lab, “In congested urban areas, about 40 percent of total gasoline use is in cars looking for parking.”

**Probabilistic Robotics**

Probabilistic robotics is a new approach to robotics that pays tribute to the uncertainty in robot perception and action. They key idea of probabilistic robotics is to represent uncertainty explicitly, using the calculus of probability theory. Put differently, instead of relying on a single “best guess” as to what might be the case in the world, probabilistic algorithms represent information by probability distributions over a whole space of possible hypotheses. By doing so, they can represent ambiguity and degree of belief in a mathematically sound way, enabling them to accommodate all sources of uncertainty listed above. Moreover, by basing control decisions on probabilistic information, these algorithms degrade nicely in the face of the various sources of uncertainty described above, leading to new solutions to hard robotics problems.

Let us illustrate the probabilistic approach with a motivating example: mobile robot localization. Localization is the problem of estimating a robot’s coordinates in an external reference frame from sensor data, using a map of the environment. Figure 1.1 illustrates the probabilistic approach to mobile robot localization. The specific localization problem studied here is known as *global localization*, where a robot is placed somewhere in the environment and has to localize itself from scratch. In the probabilistic paradigm, the robot’s momentary estimate (also called *belief*) is represented by a probability density function over the space of all locations. This is illustrated in the first diagram in Figure 1.1, which shows a uniform distribution (the *prior*) that corresponds to maximum uncertainty. Suppose the robot takes a first sensor measurement and observes that it is next to a door. The resulting belief, shown in the second diagram in Figure 1.1, places high probability at places next to doors and low probability elsewhere. Notice that this distribution possesses three peaks, each corresponding to one of the (indistinguishable) doors in the environment. Furthermore, the resulting distribution assigns high probability to three distinct locations, illustrating

that the probabilistic framework can handle multiple, conflicting hypotheses that naturally arise in ambiguous situations. Finally, even non-door locations possess non-zero probability. This is accounted by the uncertainty inherent in sensing: With a small, non-zero probability, the robot might err and actually not be next to a door. Now suppose the robot moves. The third diagram in Figure 1.1 shows the effect of robot motion on its belief, assuming that the robot moved as indicated. The belief is shifted in the direction of motion. It is also smoothed, to account for the inherent uncertainty in robot motion. Finally, the fourth and last diagram in Figure 1.1 depicts the belief after observing another door. This observation leads our algorithm to place most of the probability mass on a location near one of the doors, and the robot is now quite confident as to where it is.

*Uncertainity in Robotics*

*Factors*:

1. Environment

Robotics Systems have in common that they are situated in the physical world, perceive their environments through sensors, and manipulate their environment through things that move.

Hence, the main goal in building any robotics system is to develop a robust software that enables the robot to treat with the numerous challenges arising in unstructured and dynamic environments.

1. Sensors

Sensors are considered as the five senses of robots to collect information about the environment, there are two primary factors that affect on the limitations of sensors:

1. Range and resolution of the sensor.
2. The noise that affect on the measurements.

for example ultrasonic sensor, the measurements of ultrasonic are very poor in accuracy in contrast the accuracy of laser sensors.

1. Robot

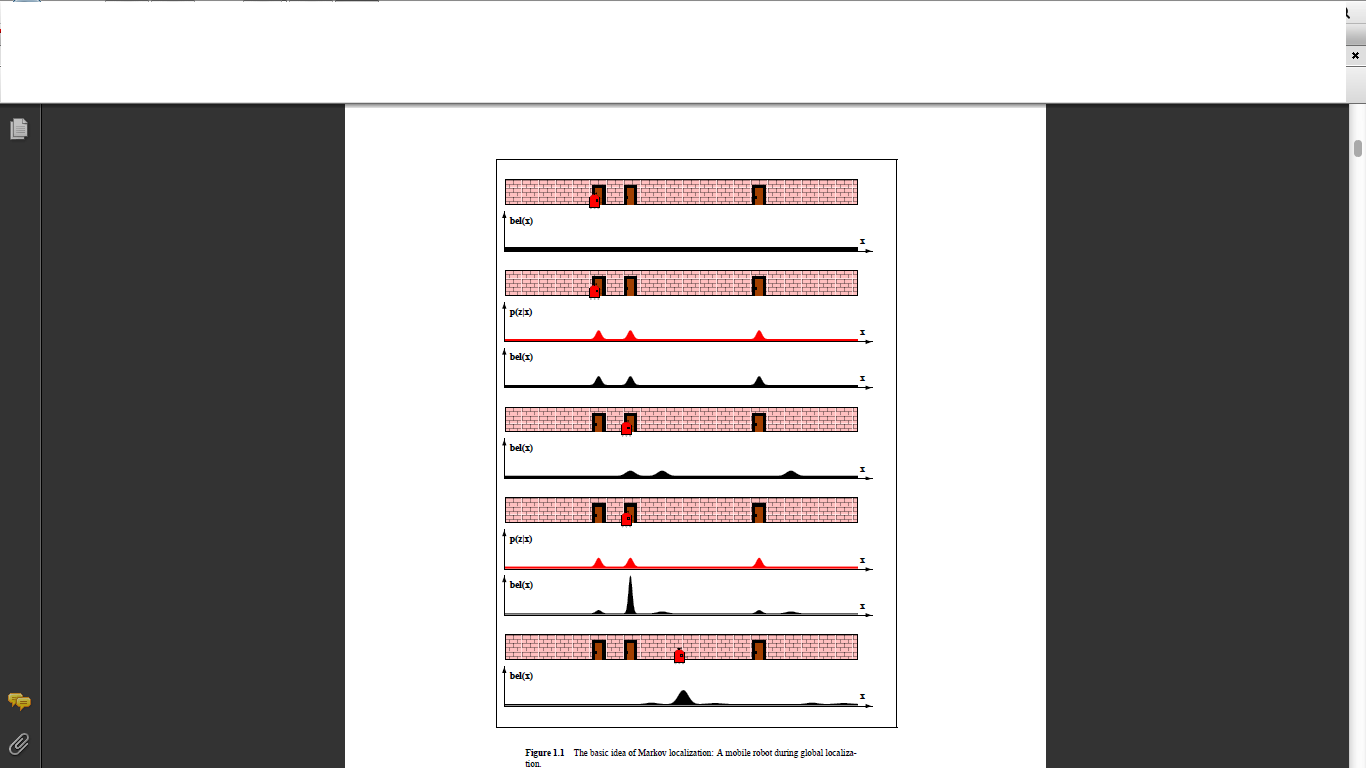
Robot actuation involves motors that are, at least to some extent, unpredictable, due effects like control noise and wear-and-tear. Some actuators, such as heavy-duty industrial robot arms, are quite accurate. Others, like low-cost mobile robots, can be extremely inaccurate.

1. Model

Models are inherently inaccurate. Models are abstractions of the real world. As such, they only partially model the underlying physical processes of the robot and its environment. Model errors are a source of uncertainty that has largely been ignored in robotics, despite the fact that most robotic models used in state-or-the-art robotics systems are rather crude.

1. Computation

Robots are real-time systems, which limits the amount of computation that can be carried out. Many state-of-the-art algorithms are approximate, achieving timely response through sacrificing accuracy some of these algorithms are in the reference of the post.



Project Phases

1. Localization(Markov and Monte-Carlo Methods)
2. Tracking ( Kalman Filters)
3. Particle Filters
4. Planning(A\* Search)
5. PID(Controller)
6. SLAM(Simultaneous Localization and Mapping)

Localization (Markov and Monte-Carlo Methods)

Mobile robot localization is the problem of determining the pose of a robot

relative to a given map of the environment. It is often called *position estimation* or *position tracking*. Mobile robot localization is an instance of the general localization problem, which is the most basic perceptual problem in robotics. This is because nearly all robotics tasks require knowledge of the location of the robots and the objects that are being manipulated (although not necessarily within a global map). Localization can be seen as a problem of coordinate transformation. Maps are described in a global coordinate system, which is independent of a robot’s pose. Localization is the process of establishing correspondence between the map coordinate system and the robot’s local coordinate system. Knowing this coordinate transformation enables the robot to express the location of objects of interests within its own coordinate frame—a necessary prerequisite for robot navigation.

Unfortunately—and herein lies the problem of mobile robot localization—the pose can usually *not* be sensed directly. Put differently, most robots do not possess a (noise free!) sensor for measuring pose. The pose has therefore to be inferred from data. A key difficulty arises from the fact that a single sensor measurement is usually insufficient to determine the pose. Instead, the robot has to integrate data over time to determine its pose. To see why this is necessary, just picture a robot located inside a building where many corridors look alike. Here a single sensor measurement (e.g., a range scan) is usually insufficient to disambiguate the identity of the corridor.

A TAXONOMY OF LOCALIZATION PROBLEMS

Not every localization problem is equally hard. To understand the difficulty of a localization problem, we will now discuss a brief taxonomy of localization problems. This taxonomy will divide localization problems along a number of important dimensions pertaining to the nature of the environment and the initial knowledge that a robot may possess relative to the localization problem.

Local Versus Global Localization

Localization problems are characterized by the type of knowledge that is available

initially and at run-time. We distinguish three types of localization problems with an increasing degree of difficulty.

* **Position tracking.** Position tracking assumes that the *initial* robot pose is known. Localizing the robot can be achieved by accommodating the noise in robot motion. The effect of such noise is usually small. Hence, methods for position tracking often rely on the assumption that the pose error is small. The pose uncertainty is often approximated by a unimodal distribution (e.g., a Gaussian). The position tracking problem is a *local* problem, since the uncertainty is local and confined to region near the robot’s true pose.
* **Global localization.** Here the initial pose of the robot is unknown. The robot is initially placed somewhere in its environment, but it lacks knowledge of where it is. Approaches to global localization cannot assume boundedness of the pose error. As we shall see later in this chapter, unimodal probability distributions are usually inappropriate. Global localization is more difficult than position tracking; in fact, it subsumes the position tracking problem.
* **Kidnapped robot problem.** This problem is a variant of the global localization problem, but one that is even more difficult. During operation, the robot can get kidnapped and teleported to some other location. The kidnapped robot problem is more difficult than the global localization problem, in that the robot might believe it knows where it is while it does not. In global localization, there robots knows that it doesn’t know where it is. One might argue that robots are rarely kidnapped in practice. The practical importance of this problem, however, arises from the observation that most state-of-the-art localization algorithms cannot be guaranteed never to fail. The ability to recover from failures is essential for truly autonomous robots. Testing a localization algorithm by kidnapping it measures its ability to recover from global localization failures.

Static Versus Dynamic Environments

A second dimension that has a substantial impact on the difficulty of localization is the environment. Environments can be static or dynamic.

* **Static environments.** Static environments are environments where the only variable quantity (state) is the robot’s pose. Put differently, only the robot moves in static environment. All other objects in the environments remain at the same location forever. Static environments have some nice mathematical properties that make them amenable to efficient probabilistic estimation.
* **Dynamic environments.** Dynamic environments possess objects other than the robot whose location or configuration changes over time. Of particular interest are changes that persist over time, and that impact more than a single sensor reading. Changes that are not measurable are of course of no relevance to localization, and those that affect only a single measurement are best treated as noise . Examples of more persistent changes are: people, daylight (for robots equipped with cameras), movable furniture, or doors. Clearly, most real environment are dynamic, with state changes occurring at a range of different speeds.

Obviously, localization in dynamic environments is more difficult than localization in static ones. There are two principal approaches for accommodating dynamics: First, dynamic entities might be included in the state vector. As a result, the Markov assumption might now be justified, but such an approach carries the burden of additional computational and modeling complexity; in fact, the resulting algorithm becomes effectively a mapping algorithm. Second, in certain situations sensor data can be filtered so as to eliminate the damaging effect of unmodeled dynamics.

Passive Versus Active Approaches

A third dimension that characterizes different localization problems pertains to the fact whether or not the localization algorithm controls the motion of the robot. We distinguish two cases:

* **Passive localization.** In passive approaches, the localization module only *observes* the robot operating. The robot is controlled through some other means, and the robot’s motion is not aimed at facilitating localization. For example, the robot might move randomly or perform its every days tasks.
* **Active localization**. Active localization algorithms control the robot so as to minimize the localization error and/or the costs arising from moving a poorly localized robot into a hazardous place.

Single-Robot Versus Multi-Robot

A fourth dimension of the localization problem is related to the number of robots

involved.

* **Single-robot localization.** The most commonly studied approach to localization deals with a single robot only. Single robot localization offers the convenience that all data is collected at a single robot platform, and there is no communication issue.
* **Multi-robot localization.** The localization problem naturally arises to teams of robots. At first glance, each robot could localize itself individually, hence the multi-robot localization problem can be solved through single-robot localization. If robots are able to detect each other, however, there is the opportunity to do better. This is because one robot’s belief can be used to bias another robot’s belief if knowledge of the relative location of both robots is available. The issue of multi-robot localization raises interesting, non-trivial issues on the representation of beliefs and the nature of the communication between them.

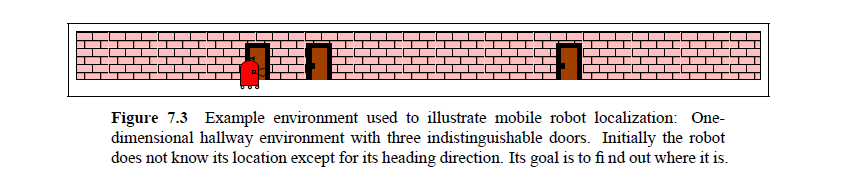
MARKOV LOCALIZATION

Probabilistic localization algorithms are variants of the Bayes filter. The straightforward application of Bayes filters to the localization problem is called *Markov localization* depicts the basic algorithm. This algorithm is derived from

the algorithm **Bayes filter** Notice that **Markov localization** also requires a map m as input. Just like the Bayes filter, Markov localizationtransforms a probabilistic belief at time t = 1 into a belief at time t. Markov localizationaddresses the global localization problem, the position tracking problem, and thekidnapped robot problem in static environments.

The initial belief, bel(x0), reflects the initial knowledge of the robot’s pose. It is setdifferently depending on the type of localization problem.

* **Position tracking.** If the initial pose is known, bel(x0) is initialized by a pointmass distribution. Let \_x0 denote the (known) initial pose. Then Point-mass distributions are discrete and therefore do not possess a density. In practice the initial pose is often just known in approximation. The belief bel(x0) is then usually initialized by a narrow Gaussian distribution centered around x0.
* **Global localization**. If the initial pose is unknown, bel(x0) is initialized by a uniform distribution over the space of all legal poses in the map.

****

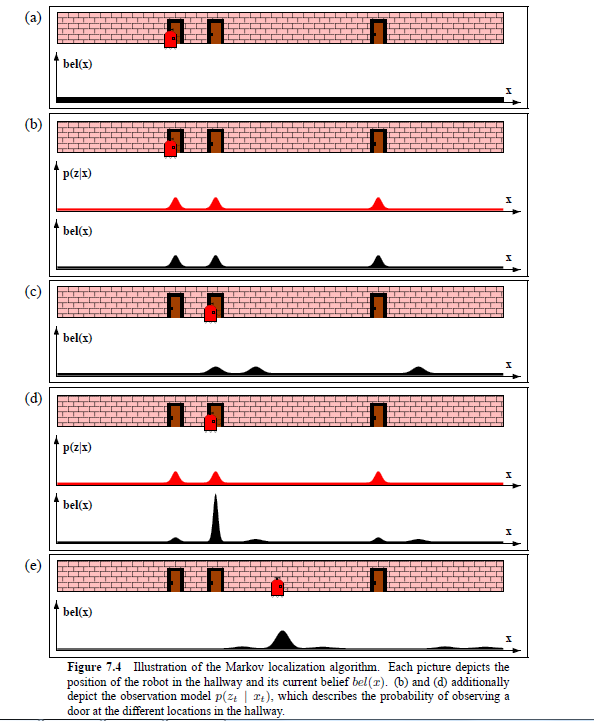
* **Other.** Partial knowledge of the robot’s position can usually easily be transformed into an appropriate initial distribution. For example, if the robot is known to start next to a door, one might initialize bel(x0) using a density that is zero except for places near doors, where it may be uniform. If it is known to be located in a specific corridor, one might initialize bel(x0) by a uniform distribution in the area of the corridor and zero anywhere else.

ILLUSTRATION OF MARKOV LOCALIZATION

Figure 7.3 depicts our one-dimensional hallway with three identically looking doors. The initial belief bel(x0) is uniform over all poses, as illustrated by the uniform density in Figure 7.4a. As the robot queries its sensors and notices that it is adjacent to one of the doors, it multiplies its belief bel(x0) by p(zt | xt,m), as stated in Line 4 of our algorithm. The upper density in Figure 7.4b visualizes p(zt | xt,m) for the hallway example. The lower density is the result of multiplying this density into the robot’s uniform prior belief. Again, the resulting belief is multi-modal, reflecting the residual uncertainty of the robot at this point. As the robot moves to the right, indicated in Figure 7.4c, Line 3 of the Markov locations

algorithm convolves its belief with the motion model p(xt | ut; xt-1). The

motion model p(xt | ut, xt-1) is not focused on a single pose but on a whole continuum of poses centered around the expected outcome of a noise-free motion. The effect is visualized in Figure 7.4c, which shows a shifted belief that is also flattened out, as a result of the convolution. The final measurement is illustrated in Figure 7.4d. Here the Markov localization algorithm multiplies the current belief with the perceptual probability p(zt | xt). At this point, most of the probability mass is focused on the correct pose, and the robot is quite confident of having localized itself. Figure 7.4e illustrates the robot’s belief after having moved further down the hallway. We already noted that Markov localization is independent of the underlying representation of the state space. In fact, Markov localization can be implemented using any of the representations discussed in Chapter 2. We will now consider three different representations and devise practical algorithms that can localize mobile robots in real time. We begin with Kalman filters, which represent beliefs by their first and second moment. We then continue with discrete, grid representations and finally introduce algorithms using particle filters.



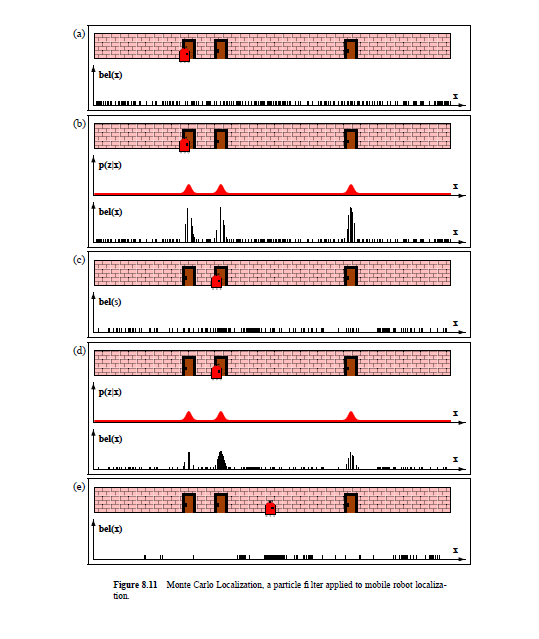
MONTE CARLO LOCALIZATION

Markov localization, MCL is applicable to both local and global localization problems. Despite its relatively short existence, MCL has already become one of the most popular localization algorithms in robotics. It is easy to implement, and tends to work well across a broad range of localization problems.

Properties of MCL

MCL can approximate almost any distribution of practical importance. The accuracy of the approximation is easily determined by the size of the particle set

M. Increasing the total number of particles increases the accuracy of the approximation. The number of particles M is a parameter that enables the user to trade off the accuracy of the computation and the computational resources necessary to run MCL. A common strategy for settingM is to keep sampling until the next pair ut and zt has arrived. In this way, the implementation is adaptive with regards to the computational resources: the faster the underlying processor, the better the localization algorithm. Such a resource- adaptivity is difficult to achieve for grid localization and Gaussian techniques. A final advantage of MCL pertains to the non-parametric nature of the approximation. As our illustrative results suggest, MCL can represent complex multi-modal probability distributions, and blend them seamlessly with focused Gaussian-style distributions . This provides MCL with the ability to solve global localization problems with high accuracy position tracking.



Random Particle MCL: Recovery from Failures

MCL, in its present form, solves the global localization problem but cannot recover from robot kidnapping, or global localization failures. As the position is acquired, particles at places other than the most likely pose gradually disappear. At some point, particles only “survive” near a single pose, and the algorithm is unable to recover if this pose happens to be incorrect.

Fortunately, this problem can be solved by a rather simple heuristic. The idea of

this heuristic is to add random particles to the particle sets. Such an “injection” of

random particles can be justified mathematically by assuming that the robot might get kidnapped with a small probability, thereby generating a fraction of random states in the motion model. Even if the robot does not get kidnapped, however, the random particles add an additional level of robustness.

The approach of adding particles raises two questions. First, how many particles

should be added at each iteration of the algorithm and, second, from which distribution should we generate these particles? One might add a fixed number of random particles at each iteration. A better idea is to add particles based on some estimate of the localization a accuracy. One way to implement this idea is to monitor the probability of sensor measurements and relate it to the average measurement probability (which is easily learned from data).

The second problem of determining which sample distribution to use, can be addressed in two ways. One can draw particles according to a uniform distribution over the pose space, and then weight them with the current observation. For some sensor models, however, it is impractical to generate particles directly in accordance to the measurement distribution.

LOCALIZATION IN DYNAMIC ENVIRONMENTS

A key limitation of all localization algorithms discussed thus far arises from the static world assumption, or Markov assumption. Most interesting environments are populated by people, and hence exhibit dynamics not modeled by the state xt. To some extent, probabilistic approaches are robust to such unmodeled dynamics, due to their ability to accommodate sensor noise. However, as previously noted, the type sensor

noise accommodated in the probabilistic filtering framework must be independent at each time step, whereas unmodeled dynamics induce effects on the sensor measurements over multiple time steps. When such effects are paramount, probabilistic localization algorithms that rely on the static world assumption may fail. An example is as follows : a mobile tour-guide robot, navigating in museums full of people. The people— their locations, velocities, intentions etc.—are hidden state relative to the localization algorithm which is not captured in the algorithms discussed thus far. Why is this problematic?

Imagine people lining up in a way that suggests the robot is facing a wall. As

a result, with each single sensor measurement the robot increases its belief of being next to a wall. Since information is treated as independent, the robot will ultimately assign high likelihood to poses near walls. Such an effect is possible with independent sensor noise, but its likelihood is vanishingly small.

There exist two fundamental techniques for dealing with dynamic environments. The first technique includes the hidden state into the state estimated by the filter; the other preprocesses sensor measurements to eliminate measurements affected by hidden state. The former methodology is mathematically the more general one: Instead of just estimating the robot’s pose, one can define filter that also estimates people’s positions, their velocities, etc. In fact, we will later on discuss such an approach, as an extension to a mobile robot mapping algorithm.

The principle disadvantage of estimating the hidden state variables lies in its computational complexity: Instead of estimating 3 variables, the robot must now calculate posteriors over a much larger number of variables. In fact, the number of variables itself is a variable, as the number of people may vary over time. Thus, the resulting algorithm will be substantially more involved than the localization algorithms discussed thus far.

The alternative—preprocessing sensor data—works well in certain limited situations, which includes situations where people’s presence may affect range finders or (to a lesser extent) camera images.

The idea is to investigate the *cause* of a sensor measurement, and to reject those likely to be affected by unmodeled environment dynamics. The sensor models discussed thus far all address different, alternative ways by which a measurement can come into existence. If we manage to associate specific ways with the presence of unwanted dynamiceffects—such as people—all we have to do it to discard those measurements that are with high likelihood caused by such an unmodeled entity.

THE KALMAN FILTER

Linear Gaussian Systems

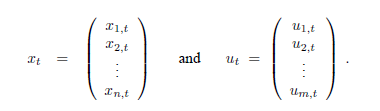
Probably the best studied technique for implementing Bayes filters is the *Kalman filter (KF)*. The Kalman filter was invented in the 1950s by Rudolph Emil Kalman, as atechnique for filtering and prediction in linear systems. The Kalman filter implementsbelief computation for continuous states. It is not applicable to discrete or hybrid statespaces.The Kalman filter represents beliefs by the moments representation: At time t, thebelief is represented by the the mean \_t and the covariance \_t. Posteriors are Gaussianif the following three properties hold, in addition to the Markov assumptions of theBayes filter.

1. The next state probability p(xt | ut; xt-1) must be a *linear* function in its arguments with added Gaussian noise. This is expressed by the following equation:

xt = Atxt-1 + Btut + Ɛt .

Here xt and xt-1 are state vectors, and ut is the control vector at time t. In our

notation, both of these vectors are vertical vectors, that is, they are of the form



At and Bt are matrices. At is a square matrix of size n \_ n, where n is the

dimension of the state vector xt. Bt is of size n\_m, withmbeing the dimension

of the control vector ut. By multiplying the state and control vector with the

matrices At and Bt, respectively, the state transition function becomes *linear* in

its arguments. Thus, Kalman filters assume linear system dynamics.

The random variable "t in (3.2) is a Gaussian random vector that models the randomness in the state transition. It is of the same dimension as the state vector. Its mean is zero and its covariance will be denoted Rt. A state transition probability of the form is called a *linear Gaussian*, to reflect the fact that it is linear in its arguments with additive Gaussian noise.

Equation defines the state transition probability p(xt | ut, xt-1). This probability

is obtained by plugging Equation into the definition of the multivariate normal distribution (3.1). The mean of the posterior state is given by

Atxt-1 + Btut and the covariance by Rt:





2. The measurement probability p(zt j xt) must also be *linear* in its arguments, with

added Gaussian noise:



Here Ct is a matrix of size k \_ n, where k is the dimension of the measurement

vector zt. The vector \_t describes the measurement noise. The distribution of \_t

is a multivariate Gaussian with zero mean and covariance Qt. The measurement

probability is thus given by the following multivariate normal distribution:

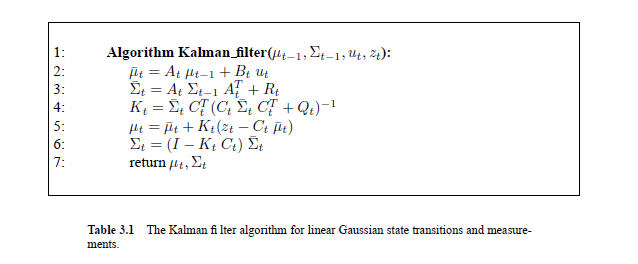


3. Finally, the initial belief bel(x0) must be normal distributed. We will denote the

mean of this belief by \_0 and the covariance by ∑0:



These three assumptions are sufficient to ensure that the posterior bel(xt) is always a Gaussian, for any point in time t. The proof of this non-trivial result can be found below, in the mathematical derivation of the Kalman filter.

The Kalman Filter Algorithm

The Kalman filter algorithm is depicted in Table 3.1. Kalman filters represent the

belief bel(xt) at time t by the mean \_t and the covariance \_t. The input of the Kalman filter is the belief at time t - 1, represented by ut-1 and ∑t-1. To update these parameters, Kalman filters require the control ut and the measurement zt. The output is the belief at time t, represented by ut and ∑t.

In Lines 2 and 3, the predicted belief is calculated representing the belief

bel(xt) one time step later, but before incorporating the measurement zt. This belief is obtained by incorporating the control ut. The mean is updated using the deterministic version of the state transition function (3.2), with the mean \_t􀀀1 substituted for the state xt-1. The update of the covariance considers the fact that states depend on previous states through the linear matrix At. This matrix is multiplied twice into the covariance, since the covariance is a quadratic matrix. The belief bel(xt) is subsequently transformed into the desired belief bel(xt) in Lines 4 through 6, by incorporating the measurement zt. The variable Kt, computed in Line 4 is called *Kalman gain*. It specifies the degree to which the measurement is incorporated into the new state estimate. Line 5 manipulates the mean, by adjusting it in proportion to the Kalman gain Kt and the deviation of the actual measurement, zt, and the measurement predicted according to the measurement probability (3.5). Finally, the new covariance of the posterior belief is calculated in Line 6, adjusting for the information gain resulting from the measurement. The Kalman filter is computationally quite efficient. For today’s best algorithms, the complexity of matrix inversion is approximately O(d2:8) for a matrix of size d \_ d. Each iteration of the Kalman filter algorithm, as stated here, is lower bounded by (approximately) O(k2:8), where k is the dimension of the measurement vector zt. This (approximate) cubic complexity stems from the matrix inversion in Line 4. It is also at least in O(n2), where n is the dimension of the state space, due to the multiplication in Line 6 (the matrix KtCt may be sparse). In many applications—such as the robot mapping applications discussed in later chapters—-the measurement space is much lower dimensional than the state space, and the update is dominated by the O(n2) operations.

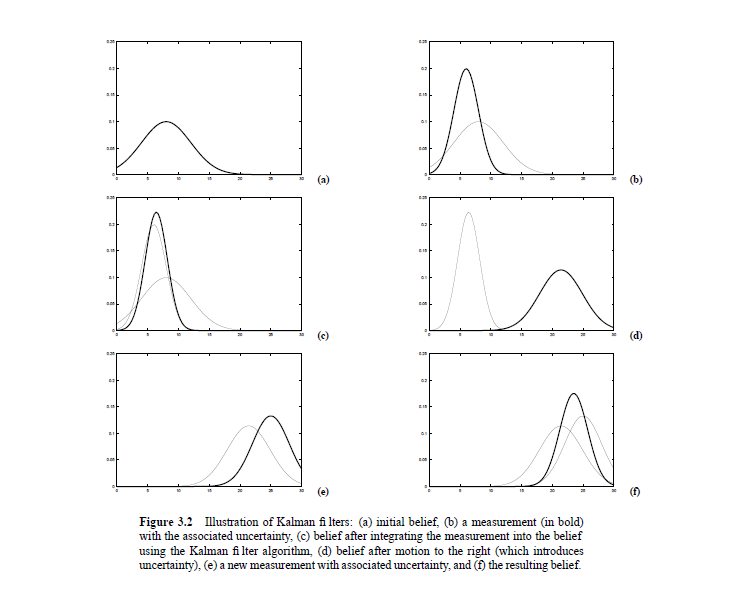
Illustration

Figure 3.2 illustrates the Kalman filter algorithm for a simplistic one-dimensional localization scenario. Suppose the robot moves along the horizontal axis in each diagram in Figure 3.2. Let the prior over the robot location be given by the normal distribution shown in Figure 3.2a. The robot queries its sensors on its location (e.g., a GPS system), and those return a measurement that is centered at the peak of the bold Gaussian in Figure 3.2b. This bold Gaussian illustrates this measurement: Its peak is the value predicted by the sensors, and its width (variance) corresponds to the uncertainty in the measurement. Combining the prior with the measurement, via Lines 4 through 6 of the Kalman filter algorithm in Table 3.1, yields the bold Gaussian in Figure 3.2c. This belief’s mean lies between the two original means, and its uncertainty radius is smaller than both contributing Gaussians. The fact that the residual uncertainty is smaller than

the contributing Gaussians may appear counter-intuitive, but it is a general characteristic of information integration in Kalman filters.

Next, assume the robot moves towards the right. Its uncertainty grows due to the fact that the next state transition is stochastic. Lines 2 and 3 of the Kalman filter provides us with the Gaussian shown in bold in Figure 3.2d. This Gaussian is shifted by the amount the robot moved, and it is also wider for the reasons just explained. Next, the robot receives a second measurement illustrated by the bold Gaussian in Figure 3.2e, which leads to the posterior shown in bold in Figure 3.2f.

As this example illustrates, the Kalman filter alternates a *measurement update step* (Lines 5-7), in which sensor data is integrated into the present belief, with a *prediction step* (or control update step), which modifies the belief in accordance to an action.The update step decreases and the prediction step increases uncertainty in the robot’sbelief.



THE PARTICLE FILTER

**Basic Algorithm**

The particle filter is an alternative nonparametric implementation of the Bayes filter. Just like histogram filters, particle filters approximate the posterior by a finite number of parameters. However, they differ in the way these parameters are generated, and in which they populate the state space. The key idea of the particle filter is to represent the posterior bel(xt) by a set of random state samples drawn from this posterior. Figure illustrates this idea for a Gaussian. Instead of representing the distribution by a parametric form (the exponential function that defines the density of a normal distribution), particle filters represent a distribution by a set of samples drawn from this distribution. Such a representation is approximate, but it is nonparametric, and therefore

can represent a much broader space of distributions than, for example,

Gaussians. In particle filters, the samples of a posterior distribution are called *particles* and are Denoted



Each particle is a concrete instantiation of the state at time t, that is, a hypothesis as to what the true world state may be at time t. Here M denotes the number of particles in the particle set Xt. In practice, the number of particles M is often a large number, e.g., M = 1; 000. In some implementations M is a function of t or of other quantities related to the belief bel(xt).

The intuition behind particle filters is to approximate the belief bel(xt) by the set of particles Xt. Ideally, the likelihood for a state hypothesis xt to be included in the

particle set Xt shall be proportional to its Bayes filter posterior bel(xt):



As a consequence of above, the denser a subregion of the state space is populated by samples, the more likely it is that the true state falls into this region. As we will discuss below, the property holds only asymptotically for M = 1 for the standard particle filter algorithm. For finite M, particles are drawn from a slightly different distribution. In practice, this difference is negligible as long as the number of particles is not too small (e.g., M ≥100).

Just like all other Bayes filter algorithms discussed thus far, the particle filter algorithm constructs the belief bel(xt) recursively from the belief bel(xt-1) one time step earlier. Since beliefs are represented by sets of particles, this means that particle filters As a consequence of (4.23), the denser a subregion of the state space is populated by samples, the more likely it is that the true state falls into this region. As we will discuss below, the property (4.23) holds only asymptotically for M = 1 for the standard particle filter algorithm. For finite M, particles are drawn from a slightly different distribution. In practice, this difference is negligible as long as the number of particles is not too small (e.g., M ≥ 100). Just like all other Bayes filter algorithms discussed thus far, the particle filter algorithm

constructs the belief bel(xt) recursively from the belief bel(xt-1) one time step

earlier. Since beliefs are represented by sets of particles, this means that particle filters construct the particle set Xt recursively from the set Xt-1. The most basic variant of the particle filter algorithm is stated in Table 4.3. The input of this algorithm is the particle set Xt-1, along with the most recent control ut and the most recent measurement zt. The algorithm then first constructs a temporary particle set \_ X which is reminiscent (but not equivalent) to the belief bel(xt). It does this by systematically processing each particle x[m] t-1 in the input particle set Xt-1 as follows.

1. Line 4 generates a hypothetical state x[m] t for time t based on the particle x[m]t-1 and the control ut. The resulting sample is indexed by m, indicating that it is generated from the m-th particle in Xt-1. This step involves sampling from the next state distribution p(xt | ut, xt-1). To implement this step, one needs to be able to sample from p(xt | ut, xt-1). The ability to sample from the state

transition probability is not given for arbitrary distributions p(xt | ut, xt-1).

However, many major distributions in this book possess efficient algorithms for

generating samples. The set of particles resulting from iterating Step 4 M times

is the filter’s representation of bel(xt).

2. Line 5 calculates for each particle x[m] t the so-called *importance factor*, denoted w[m] t . Importance factors are used to incorporate the measurement zt into the particle set. The importance, thus, is the probability of the measurement zt under the particle x[m] t , that is, w[m] t = p(zt | x[m] t ). If we interpret w[m] t as the *weight* of a particle, the set of weighted particles represents (in approximation) the Bayes filter posterior bel(xt).

3. The real “trick” of the particle filter algorithm occurs in Lines 8 through 11 in . These lines implemented what is known as *resampling* or *importance resampling*.

The algorithm draws with replacement M particles from the temporary set \_ Xt. The probability of drawing each particle is given by its importance weight.

Resampling transforms a particle set of M particles into another particle set of the

same size. By incorporating the importance weights into the resampling process,

the distribution of the particles change: whereas before the resampling step, they

were distribution according to bel(xt), after the resampling they are distributed

(approximately) according to the posterior bel(xt) = \_ p(zt | x[m]

t )bel(xt). In fact, the resulting sample set usually possesses many duplicates, since particles are drawn with replacement. More important are the particles that are *not* contained in Xt: those tend to be the particles with lower importance weights.

The resampling step has the important function to force particles back to the posterior bel(xt). In fact, an alternative (and usually inferior) version of the particle filter would never resample, but instead would maintain for each particle an importance weight that is initialized by 1 and updated multiplicatively:

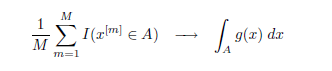


Such a particle filter algorithm would still approximate the posterior, but many of its particles would end up in regions of low posterior probability. As a result, it would require many more particles; how many depends on the shape of the posterior. The resampling step is a probabilistic implementation of the Darwinian idea of *survival* *of the fittest*: It refocuses the particle set to regions in state space with high posterior probability. By doing so, it focuses the computational resources of the filter algorithm to regions in the state space where they matter the most.

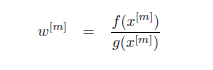
Importance Sampling

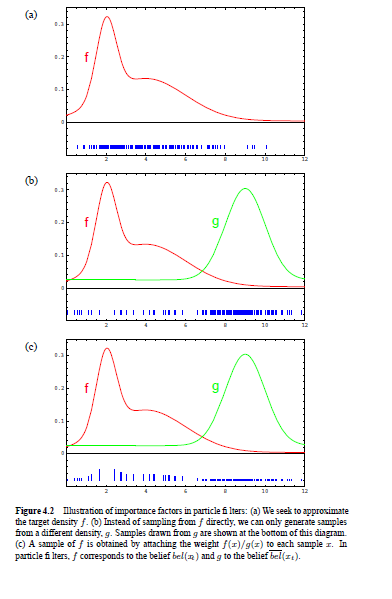
For the derivation of the particle filter, it shall prove useful to discuss the resampling step in more detail. Figure 4.2 illustrates the intuition behind the resampling step.

Figure 4.2a shows a density function f of a probability distribution called the *target* *distribution*. What we would like to achieve is to obtain a sample from f. However, sampling from f directly may not be possible. Instead, we can generate particles from a related density, labeled g in Figure 4.2b. The distribution that corresponds to the density g is called *proposal distribution*. The density g must be such that f(x) > 0 implies g(x) > 0, so that there is a non-zero probability to generate a particle when sampling from g for any state that might be generated by sampling from f. However, the resulting particle set, shown at the bottom of Figure 4.2b, is distributed according to g, not to f. In particular, for any interval A \_ range(X) (or more generally, any Borel set A) the empirical count of particles that fall into A converges to the integral of g under A:

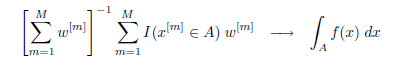


To offset this difference between f and g, particles x[m] are weighted by the quotient





This is illustrated by Figure 4.2c: The vertical bars in this figure indicate the magnitude of the importance weights. Importance weights are the non-normalized probability mass of each particle. In particular, we have



where the first term serves as the normalizer for all importance weights. In other

words, even though we generated the particles from the density g, the appropriately weighted particles converge to the density f.

The specific convergence involves an integration over a set A. Clearly, a particle set represents a discrete distribution, whereas f is continuous in our example. Because of this, there is no density that could be associated with a set of particles. The convergence, thus, is over the cumulative distribution function of f, not the density itself (hence the integration over A). A nice property of importance sampling is that it converges to the true density if g(x) > 0 whenever f(x) > 0. In most cases, the rate of convergence is in where M is the number of samples. The constant factor depends on the similarity of f(s) and g(s).

In particle filters, the density f corresponds to the target belief bel(xt). Under the

(asymptotically correct) assumption that the particles in Xt-1 are distributed according to bel(xt-1), the density g corresponds to the product distribution:



This distribution is called the *proposal distribution*.

Properties of the Particle Filter

Particle filters are approximate and as such subject to approximation errors. There are four complimentary sources of approximation error, each of which gives rise to improved versions of the particle filter.

**1.** The first approximation error relates to the fact that only finitely many particles

are used. This artifact introduces a systematic *bias* in the posterior estimate.

To see, consider the extreme case of M = 1 particle. In this case, the loop in

Lines 3 through 7 in Table 4.3 will only be executed once, and \_ Xt will contain

only a single particle, sampled from the motion model. The key insight is that

the resampling step (Lines 8 through 11 in Table 4.3) will now *deterministically*

accept this sample, regardless of its importance factor w[m] t . Put differently, the

measurement probability p(zt | x[m]t ) plays no role in the result of the update,

and neither does zt. Thus, if M = 1, the particle filter generates particles from

the probability



instead of the desired posterior p(x| u1:t, z1:t). It flatly ignores all measurements.

How can this happen?

The culprit is the normalization, implicit in the resampling step. When sampling

in proportion to the importance weights (Line 9 of the algorithm), w[m] t becomes its own normalizer if M = 1:



In general, the problem is that the non-normalized values wt[m] are drawn from

anM-dimensional space, but after normalization they reside in a space of dimension M-1. This is because after normalization, them-th weight can be recovered from them=1 other weights by subtracting those from 1. Fortunately, for larger values of M, the effect of loss of dimensionality, or degrees of freedom, becomes less and less pronounced.

**2.** A second source of error in the particle filter relates to the randomness introduced in the resampling phase. To understand this error, it will once again be

useful to consider the extreme case, which is that of a robot whose state does not

change. Sometimes, we know for a fact that xt = xt-1. A good example is that

of mobile robot localization, for a non-moving robot. Let us furthermore assume

that the robot possesses no sensors, hence it cannot estimate the state, and that

it is unaware of the state. Initially, our particle set X0 will be generated from

the prior; hence particles will be spread throughout the state space. The random

nature of the resampling step (Line 8 in the algorithm) will regularly fail to draw

a state sample x[m]. However, since our state transition is deterministic, no new

states will be introduced in the forward sampling step (Line 4). The result is quite

daunting: With probability one, M identical copies of a single state will survive;

the diversity will disappear due to the repetitive resampling. To an outside observer, it may appear that the robot has uniquely determined the world state—an apparent contradiction to the fact that the robot possesses no sensors.

This example hints at an important limitation of particle filters with immense

practical ramifications. In particular, the resampling process induces a loss of diversity in the particle population, which in fact manifests itself as approximation

error. Such error is called *variance* of the estimator: Even though the variance of

the particle set itself decreases, the variance of the particle set as an estimator of

the true belief increases. Controlling this variance, or error, of the particle filter

is essential for any practical implementation.

There exist two major strategies for variance reduction. First, one may reduce the

frequency at which resampling takes place. When the state is known to be static

(xt = xt-1) one should never resample. This is the case, for example, in mobile

robot localization: When the robot stops, resampling should be suspended (and

in fact it is usually a good idea to suspend the integration of measurements as

well). Even if the state changes, it is often a good idea to reduce the frequency of

resampling. Multiple measurements can always be integrated via multiplicatively

updating the importance factor as noted above. More specifically, it maintains the

importance weight in memory and updates them as follows:



The choice of when to resample is intricate and requires practical experience:

Resampling too often increases the risk of losing diversity. If one samples too

infrequently, many samples might be wasted in regions of low probability. A

standard approach to determining whether or not resampling should be performed is to measure the variance of the importance weights. The variance of the weights relates to the efficiency of the sample based representation. If all weights are identical, then the variance is zero and no resampling should be performed. If, on the other hand, the weights are concentrated on a small number of samples, then the weight variance is high and resampling should be performed. The second strategy for reducing the sampling error is known as *low variance* *sampling*. Table 4.4 depicts an implementation of a low variance sampler. The basic idea is that instead of selecting samples independently of each other in the resampling process (as is the case for the basic particle filter in Table 4.3), the selection involves a sequential stochastic process. Instead of choosing M random numbers and selecting those particles that correspond to these random numbers, this algorithm computes a single random number and selects samples according to this number but still with a probability proportional to the sample weight. This is achieved by drawing a random number r in the interval [0;M􀀀1[, where M is the number of samples to be drawn at time t. The algorithm in Table 4.4 then selects particles by repeatedly adding the fixed amount M􀀀1 to r and by choosing the particle that corresponds to the resulting number. Any number u in [0; 1] points to exactly one particle, namely the particle i for which



The while loop in Table 4.4 serves two tasks, it computes the sum in the righthand

side of this equation and additionally checks whether i is the index of the

first particle such that the corresponding sum of weights exceeds u. The selection

is then carried out in Line 12. This process is also illustrated in Figure 4.3.

The advantage of the low-variance sampler is threefold. First, it covers the space

of samples in a more systematic fashion than the independent random sampler.

This should be obvious from the fact that the dependent sampler cycles through

all particles systematically, rather than choosing them independently at random.

Second, if all the samples have the same importance factors, the resulting sample

set \_ Xt is equivalent to Xt so that no samples are lost if we resample without

having integrated an observation into Xt. Third, the low-variance sampler has

a complexity of O(M). Achieving the same complexity for independent sampling

is difficult; obvious implementations require a O(logM) search for each

particle once a random number has been drawn, which results in a complexity of

O(M logM) for the entire resampling process. Computation time is of essence

when using particle filters, and often an efficient implementation of the resampling process can make a huge difference in the practical performance. For these reasons, most implementations of particle filters in robotics tend to rely on mechanisms like the one just discussed.

In general, the literature on efficient sampling is huge. Another popular option

is *stratified sampling*, in which particles are grouped into subsets. The number

of samples in each subset can be kept the same over time, regardless of the total

weight of the particles contained in each subset. Such techniques tend to perform

well when a robot tracks multiple, distinct hypotheses with a single particle filter.

**3.** A third source of error pertains to the divergence of the proposal and target distribution. We already hinted at the problem above, when discussing importance sampling. In essence, particles are generated from a proposal distribution that does not consider the measurement (cf., Equation (4.28)). The target distribution, which is the familiar Bayes filter posterior, depends of course on the measurement. The efficiency of the particle filter relies crucially on the ’match’ between the proposal and the target distribution. If, at one extreme, the sensors of the robot are highly inaccurate but its motion is very accurate, the target distribution will be similar to the proposal distribution and the particle filter will be efficient. If, on the other hand, the sensors are highly accurate but the motion is not, these distributions can deviate substantially and the resulting particle filter can become arbitrarily inefficient. An extreme example of this would be a robot with *deterministic* sensors. For most deterministic sensors, the support of the measurement probability p(z j x) will be limited to a submanifold of the state space. For example, consider a mobile robot that performs localization with noise-free range sensors. Clearly, p(z j x) will be zero for almost every state x, with the exceptions of those that match the range measurement z exactly. Such a situation can be fatal: the proposal distribution will practically never generate a sample x which *exactly* corresponds to the range measurement z. Thus, all importance weights will be zero with probability one, and the resampling step becomes ill conditioned. More generally, if p(z j x) is degenerate, meaning that its support is restricted to a manifold of a smaller dimension than the dimension of the state space, the plain particle filter algorithm is inapplicable. There exist a range of techniques for overcoming this problem. One simpleminded technique is to simply assume more noise in perception than there actually is. For example, one might use a measurement model p(z j x) that overestimates the actual noise in the range measurements. In many implementations, such a step improves the accuracy of the particle filter—despite the oddity of using a knowingly incorrect measurement probability. Other techniques involve modifications of the proposal distribution in ways that incorporate the measurement. Such techniques will be discussed in later chapters of this book.

**4.** A fourth and final disadvantage of the particle filter is known as the *particle deprivation* *problem*. When performing estimation in a high-dimensional space, there may be no particles in the vicinity to the correct state. This might be because the number of particles is too small to cover all relevant regions with high likelihood. However, one might argue that this ultimately must happen in any particle filter, regardless of the particle set size M. Particle deprivation occurs as the result of random resampling; an unlucky series of random numbers can wipe out all particles near the true state. At each resampling step, the probability for this to happen is larger than zero (although it is usually exponentially small in

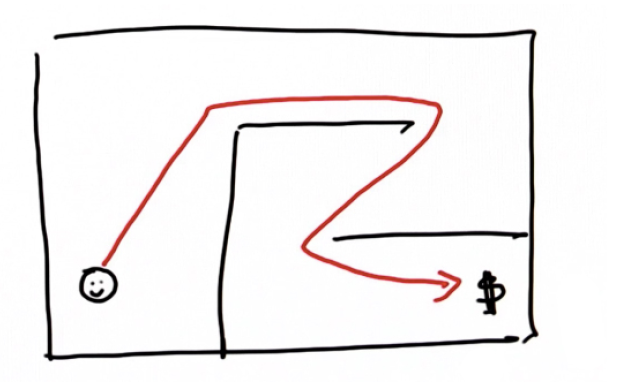
M). Thus, we only have to run the particle filter long enough. Eventually, we

will generate an estimate that is arbitrarily incorrect. In practice, problems of this nature only tend to arise when M is small relative to the space of all states with high likelihood. A popular solution to this problem is to add a small number of randomly generated particles into the set after each resampling process, regardless of the actual sequence of motion and measurement commands. Such a methodology can reduce (but not fix) the particle deprivation problem, but at the expense of an incorrect posterior estimate. The advantage of adding random samples lies in its simplicity: The software modification necessary to add random samples in a particle filter is minimal. As a rule of thumb, adding random samples should be considered a measure of last resort, which should only be applied if all other techniques for fixing a deprivation problem have failed.

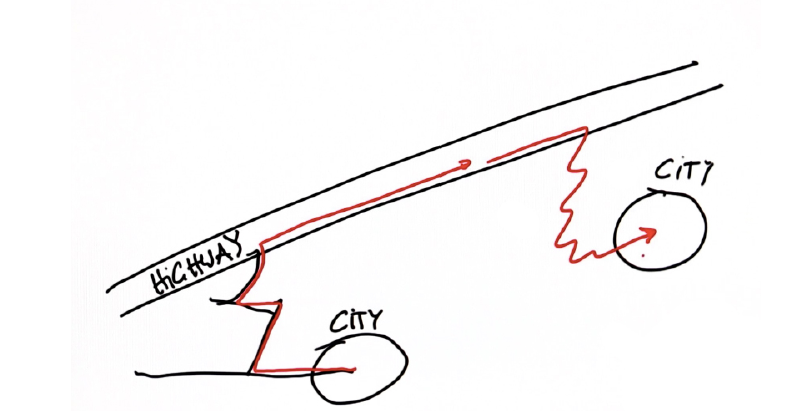
This discussion showed that the quality of the sample based representation increases with the number of samples. An important question is therefore how many samples should be used for a specific estimation problem. Unfortunately, there is no perfect answer to this question and it is often left to the user to determine the required number of samples. As a rule of thumb, the number of samples strongly depends on the dimensionality of the state space and the uncertainty of the distributions approximated by the particle filter. For example, uniform distributions require many more samples than distributions focused on a small region of the state space. A more detailed discussion on sample sizes will be given in the context of robot localization, when we consider adaptive particle filters .

Planning (A\* Search)

The fundamental problem in motion planning is that a robot might live in a world that looks like theone pictured below and will want to find a goal like $. The robot has to have a plan to get to its goal, as indicated by the red line route.



This same problem occurs for a self-driving car that lives in a city and has to find its way to its target location by navigating through a network of streets and along the highway.



For the self-driving car, other obstacles besides winding streets, may become a problem. For example, in this bird's-eye view of the car's route (red arrows), the car would have to turn right, shift lanes and then make a left-hand turn, crossing opposing traffic to reach its goal.

Alternatively, the car could opt to take a detour (black arrows) if there is some obstacle in the lane it was trying to change to. The process of getting a robot from a start location to a goal location is referred to as **robot motion planning**, or just planning.

In the planning problem :

The Given material is**:**

Map

Starting Location

Goal Location

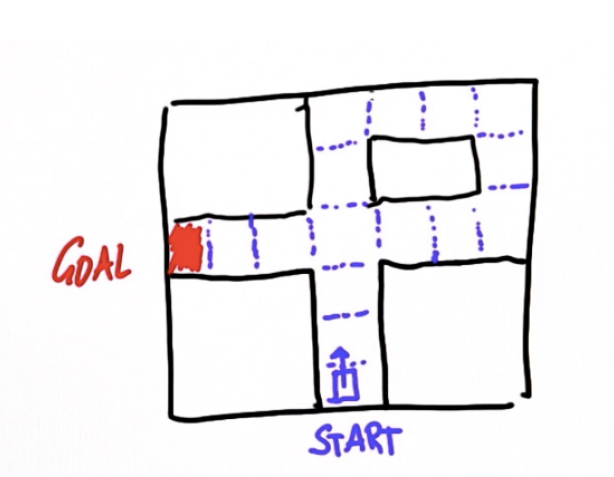
Cost (time it takes to drive a certain route)

Goal:

Find the minimum cost path.

**Computing Cost**

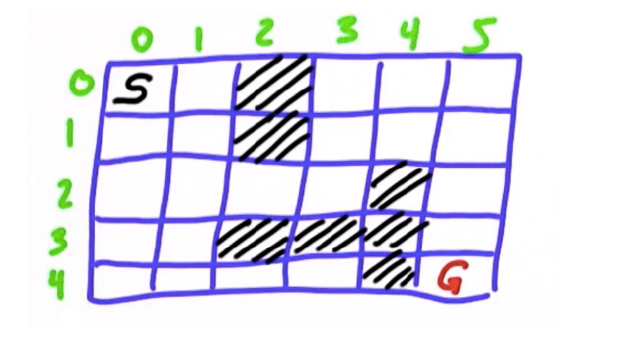
Suppose you live in a discrete world that is split up into grid cells and your initial location is facing forward, while your goal location is facing to the left.



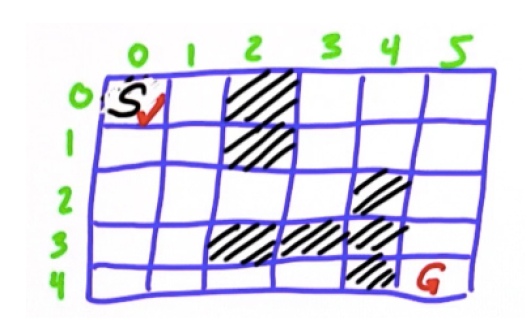
Assume that for each time step you can make one of two action -- either move forward or turn the vehicle. Each move costs exactly one unit.

**Writing a Search Program (First Search Program)**

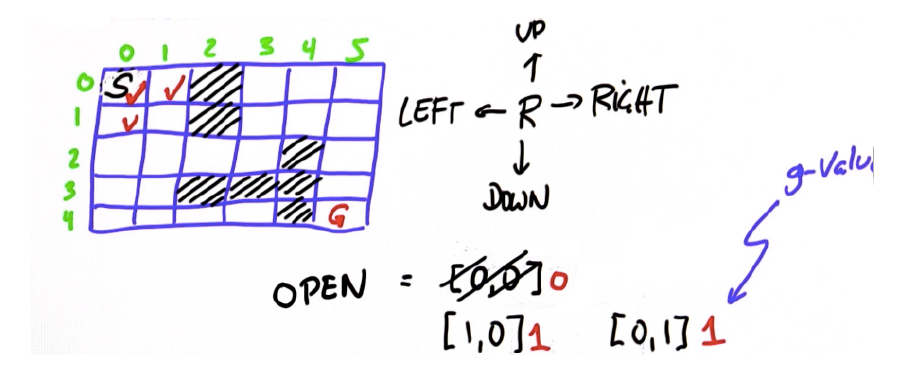
The first step towards writing this program is to name the grid cells. Across the top, name the columns zero to five, and along the side you can name the rows zero to four. Think about each grid cell as a data point, called a **node** and the goal point is called the **goal node**.



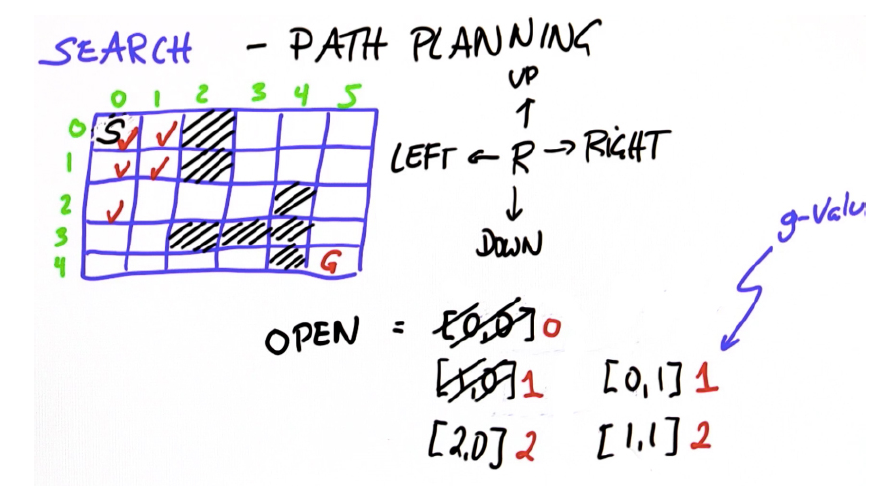
From here you will make a list of nodes that you will be investigating further -- that you will be expanding. Call this list **open**. The initial state of this list is **[0,0]**. You may want to check off the nodes that you have already picked, as shown here with the red check mark.



Now, you can test whether this node is your final goal node -- which, just by looking at the grid you can tell that it obviously is not. So, what you want to do next is to expand this node by taking it off the **open** list and looking at its successors, of which there are two: **[1,0]** and **[0,1]**. Then, you can check those cells off on your grid. Most importantly, remember to note how many expansions it takes to get to the goal -- this is called the **g-value**. By the end of your planning, the g-value will be the length of the path.



Continue to expand further. Always expand the cell with the smallest g-value, but since the two above are equivalent, it doesn't make a difference which one you expand first. In the image below, the first cell is expanded to include the unchecked cells, **[2,0]** and **[1,1]**, each with a g-value of two.



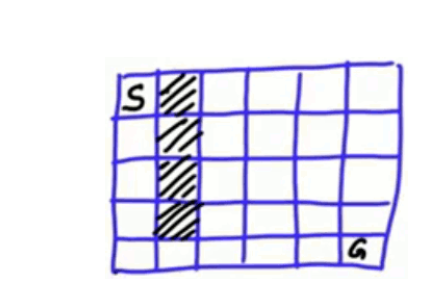
The next lowest g-value is the cell **[0,1]**, but since the surrounding cells, **[0,0]** and **[1,1]**, are already checked it cannot be expanded any further. The process of expanding the cells, starting with the one with the lowest g-value, should yield the same result you found as the number of moves it takes to get from the start to the goal -- 11.

A\* search algorithm (A Star)

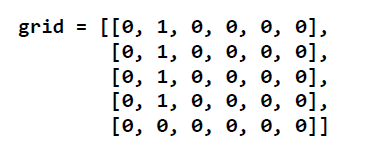
The A\* (A-star) algorithm can be used for our path finding problem. It is a variant of the search algorithm that we implemented, that is more efficient because you don’t need to expand every node. The A\* algorithm was first described by Peter Hart, Nils Nilsson and Bertram Raphael in 1968. If you understand the mechanism for searching by gradually expanding the nodes in the open list, A\* is *almost* the same thing, but not quite.

Difference

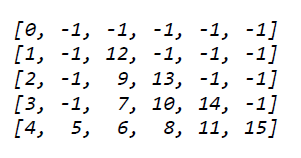
To illustrate the difference, consider the same world as before, but with a different obstacle configuration:



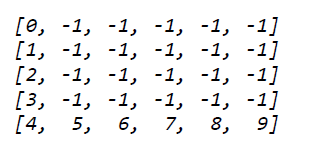
This is one example where A\* performs really well. Obviously, from the start node we have to go straight down. But from the corner we still have to search for the best path to the goal. Here is the same world configured in code:



When you run the code you wrote before and print out the expansion list, you will see that you have to travel down to node **4**, and then expand into the open space, until finally you hit the goal node at **15**:

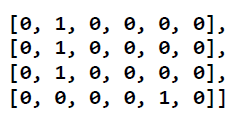


This took **16** expansions (including the initial zero-expansion) to get us to the goal node. If you use the A\* algorithm, you'll get this output:

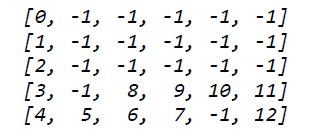


Notice how that took only **10** expansions to get to the goal. It expands down to **4** and then goes straight to goal, never expanding to the area above. The A\* algorithm somehow magically knows that any other path to the goal will be longer than going straight. Change the world to something more challenging. Put an obstacle right next to the goal.





Run the A\* search again to return:



When A\* encounters the obstacle, it expands up, but then goes right and then down to the goal again. So, it somehow does the minimum amount of work necessary to make the maximum progress towards the goal. This is the basic principle of A\*.

A\* uses a **heuristic function**, which is a function that has to be set up. If it’s all zeroes, A\* resorts back to the search algorithm already implemented. If we call the heuristic function ***h***, then each cell results in a value.

**Dynamic Programming (Dynamic Programming)**

Dynamic programming is an alternative method for planning. It has a number of advantages and disadvantages. Much like A\* it will find the shortest path. Given a map and one or more goal positions, it will output the best path from any possible starting location. This planning technique is not just limited to one starting location, but from any starting location. Consider this scenario: Our car, as indicated by the blue square at the bottom of the map, has a goal of the blue circle on the left side of the map. If you are able to perform a lane shift into the left lane, you will quickly reach your goal by turning left. However, there may be an obstacle such as a large truck in the left lane that does not allow us to make that turn. So we will instead have to go through the intersection and make three right turns in order to get into the lane that will lead to our goal. This scenario shows the stochastic (probabilistic) nature of the world and that there is a need to

plan not only for the most likely position, but for other positions as well. Dynamic programming is helpful because it does give us a solution for every location. Dynamic programming will give us an optimum action (called the **policy**) to perform for every navigable grid cell.

PID (Controller)

Control theory is a vast field, and someone can actually take many classes just to master the PID controller. For now, you will learn and implement the basics of this controller without worrying too much about the theory behind it, just to get a

feeling for how it works. It will be fun, and you’ll be able to drive a car around!

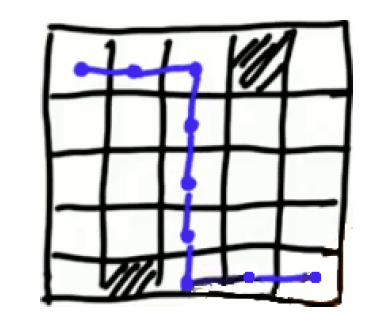
The Google car actually uses a version of this controller (described in this article) that is slightly more attuned to the specifics of the car, but the basic idea behind it remains the same. You’ll get the essence of what it means to control a car.

Consider the following car with the steerable front axle and two non-steerable wheels in the back.

You want the car to drive along the black line, the reference trajectory.

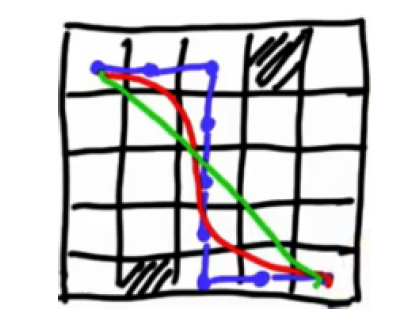
Assume the car has a fixed forward velocity, but you have the ability to set the steering angle of the car. The way we will control the steering so the car follows the reference trajectory is that you should steer in proportion to the cross-track error. The larger the error, the more you are willing to turn to the target trajectory. You can see that this works – as you get closer to the trajectory, you’ll steer slower and slower, and you’ll reach the trajectory!

It is easy to understand that the other two options are really bad. Constant steering would put you on a fixed circle, not a straight line. Random steering would essentially lead to something like a random walk behaviour, which is not a good idea either.

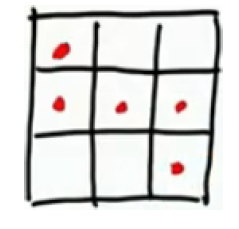


A path like this has a lot of disadvantages. You don’t want the robot to go straight, take a 90-degree turn, then go straight again. For one, a car can not even do this, and this route will force the robot to move really slowly around the corners. A much better path would look like the red line in this next picture. This is a much smoother path. In an extreme case, you might generate a path like the green

line.



Smoothing Algorithm



For smoothing purposes we will call each point **xi** , which goes from **x0** to **xN-1** and each **x** is really a 2-dimensional coordinate, but that should be immaterial to smoothing – whether it is in 1D, 2D or 3D.

Smoothing algorithm:

**1.** Initially, create variables, **yi**, that are the same as the **xi**. Remember, these are the nonsmooth

path locations the planner has found:

**yi = xi**

**2.**  Then you optimize by two criteria by minimizing these two expressions:

**(xi - yi)2 → min**

**(yi - yi+1)2 → min**

The first expression minimizes the distance between the original point and the smooth point, and

the second minimizes the distance between two consecutive smooth points.

**Path Smoothing**

How can you optimize these two terms? You can use **gradient descent**, that is, for every time step, you take a small step in the direction of minimizing the error. The expression for the first objective **(xi - yi)2 → min** is following:

**yi = yi + (xi - yi)**.

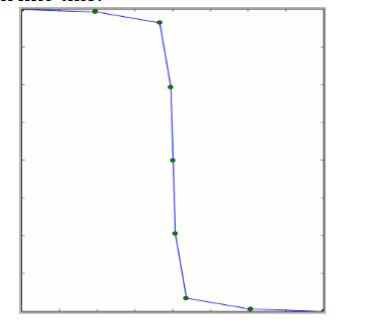
When you iterate, you assign to **yi** the old value of **yi** and add a term that is proportional to the deviation of **yi** to **xi**, weighted by the weight function alpha . This is not exactly the sameas before. You should set this to 0.5. The second term you can implement is one where you retain the old **y** variable, but move a little bit in the direction of **yi+1** and away from **yi**:

**yi = yi + \* (yi+1 - yi)**

But an even better implementation looks as follows:

**yi = yi + \* (yi+1 + yi-1 - 2 \* yi)**

This combines the step on the left with the step on the right, realizing that each **yi** occurs twice in the optimization term **(yi - yi+1)2 → min**. From here, you can implement it in single update rule where you wish **yi** to be as close to **yi-1** and simultaneously to be as close to **yi+1** by optimizing the combined term.



SIMULTANEOUS LOCALIZATION AND MAPPING

INTRODUCTION

This and the following chapters address one of the most fundamental problems in

robotics, the *simultaneous localization and mapping problem*. This problem is commonly abbreviated as *SLAM*, and is also known as *Concurrent Mapping and Localization*, or CML. SLAM problems arise when the robot does not have access to a map of the environment; nor does it have access to its own poses. Instead, all it is given are measurements z1:t and controls u1:t. The term “simultaneous localization and mapping” describes the resulting problem: In SLAM, the robot acquires a map of its environment while simultaneously localizing itself relative to this map. SLAM is significantly more difficult than all robotics problems discussed thus far: It is more difficult than localization in that the map is unknown and has to be estimated along the way. It is more difficult than mapping with known poses, since the poses are unknown and have to be estimated along the way.

From a probabilistic perspective, there are two main forms of the SLAM problem,

which are both of equal practical importance. One is known as the *online SLAM*

*problem*: It involves estimating the posterior over the momentary pose along with the map:

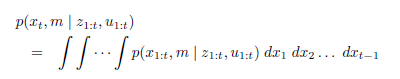


Here xt is the pose at time t, m is the map, and z1:t and u1:t are the measurements and controls, respectively. This problem is called the online SLAM problem since it only involves the estimation of variables that persist at time t. Many algorithms for the online SLAM problem are incremental: they discard past measurements and controls once they have been processed.

The second SLAM problem is called the *full SLAM problem*. In full SLAM, we seek

to calculate a posterior over the entire path x1:t along with the map, instead of just the current pose xt:

  
This subtle difference in the formulation of the SLAM problem between online and full SLAM has ramifications in the type algorithms that can be brought to bear. In particular, the online SLAM problem is the result of integrating out past poses from the full SLAM problem:



In online SLAM, these integrations are typically performed one-at-a-time, and they cause interesting changes of the dependency structures in SLAM that we will fully explore in the next chapter.

A second key characteristic of the SLAM problem has to do with the nature of the

estimation problem. SLAM problems possess a continuous and a discrete component.

The continuous estimation problem pertains to the location of the objects in the map and the robot’s own pose variables. Objects may be landmarks in feature-based representation, or they might be object patches detected by range finders. The discrete nature has to do with correspondence: When an object is detected, a SLAM algorithm must reason about the relation of this object to previously detected objects. This reasoning is typically discrete: Either the object is the same as a previously detected one, or it is not.

SLAM WITH EXTENDED KALMAN FILTERS

Setup and Assumptions

Historically the earliest, and perhaps the most influential SLAM algorithm is based on the extended Kalman filter, or EKF. In a nutshell, the EKF SLAM algorithm applies the EKF to online SLAM using maximum likelihood data association. In doing so, EKF SLAM is subject to a number of approximations and limiting assumptions:

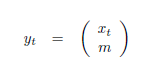
* **Feature-based maps.** Maps, in the EKF, are composed of point landmarks. For computational reasons, the number of point landmarks is usually small (e.g., smaller than 1,000). Further, the EKF approach tends to work well the less ambiguous the landmarks are. For this reason, EKF SLAM requires significant engineering of feature detectors, sometimes using artificial beacons or landmarks as features.
* **Gaussian noise.** As any EKF algorithm, EKF SLAM makes a Gaussian noise assumption for the robot motion and the perception. The amount of uncertainty in the posterior must be relatively small, since otherwise the linearization in EKFs tend to introduce intolerable errors.
* **Positive measurements.** It cannot process negative information that arises from the absence of landmarks in a sensor measurements. This is a direct consequence of the Gaussian belief representation

SLAM with Known Correspondence

The SLAM algorithm for the case with known correspondence addresses the continuous portion of the SLAM problem only. SLAM algorithm also

estimates the coordinates of all landmarks encountered along the way. This makes it necessary to include the landmark coordinates into the state vector.

For convenience, let us call the state vector comprising robot pose and the map the *combined state vector*, and denote this vector yt. The combined vector is given by





Here x, y, and \_ denote the robot’s coordinates at time t, mi;x;mi;y are the coordinates of the i-th landmark, for i = 1N, and si is its signature. The dimension of this state vector is 3N + 3, where N denotes the number of landmarks in the map. Clearly, for any reasonable number of N.

The fact that the Kalman gain is fully populated for all state variables—and not just the observed landmark and the robot pose—is important. In SLAM, observing a landmark does not just improve the position estimate of this very landmark, but that of other landmarks as well. This effect is mediated by the robot pose: Observing a landmark improves the robot pose estimate, and as a result it eliminates some of the uncertainty of landmarks previously seen by the same robot. The amazing effect here is that we do not have to model past poses explicitly—which would put us into the realm of the full SLAM problem and make the EKF a non-realtime algorithm. Instead, this dependence is captured in the Gaussian posterior, more specifically, in the off-diagonal covariance elements of the matrix ∑t.