19 Beliefs

A POMDP is an MDP with state uncertainty. The agent receives a potentially imperfect *observation* of the current state rather than the true state. From the past sequence of observations and actions, the agent develops an understanding of the world. This chapter discusses how the *belief* of the agent can be represented by a probability distribution over the underlying state. Various algorithms are presented for updating our belief based on the observation and action taken by the agent.[1] We can perform exact belief updates if the state space is discrete or if certain linear Gaussian assumptions are met. In cases where these assumptions do not hold, we can use approximations based on linearization or sampling.

[1] Different methods for belief updating are discussed in the context of robotic applications by S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

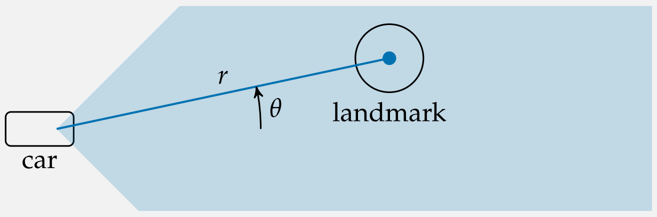
19.1 Belief Initialization

There are different ways to represent our beliefs. In this chapter, we will discuss *parametric* representations where the belief distribution is represented by a set of parameters of a fixed distribution family, such as the categorical or the multivariate normal distribution. We will also discuss *non-parametric* representations where the belief distribution is represented by particles, or points sampled from the state space. Associated with different belief representations are different procedures for updating the belief based on the action taken by the agent and the observation.

Before the agent takes any actions or makes any observations, we start with an initial belief distribution. If we have some prior information about where the agent might be in the state space, we can encode this in the initial belief. We generally want to use diffuse initial beliefs in the absence of information to avoid being overly confident in the agent being in a region of the state space where it might not actually be. A strong initial belief focused on states that are far from the true state can lead to poor state estimates even after many observations.

A diffuse initial prior can cause difficulties, especially for non-parametric representations of the belief, where the state space can only be very sparsely sampled. In some cases, it may be useful to wait until an informative observation is made to initialize our beliefs. For example, in robot navigation problems, we might want to wait until the sensors detect a known *landmark* and then initialize the belief appropriately. The landmark can help narrow down the relevant region of the state space so we can focus our sampling of the space in the area consistent with the landmark observation. Example 19.1 illustrates this concept.

Consider an autonomous car equipped with a localization system that uses camera, radar, and lidar data to track its position. The car is able to identify a unique landmark at a range r and bearing θ from its current pose.

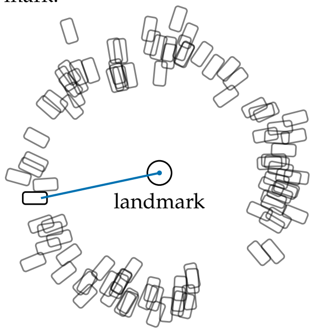


The range and bearing measurements have zero-mean Gaussian noise with variance ν_r and ν_θ, respectively, and the landmark is known to be at (x, y). Given a measurement r and θ, we can produce a distribution over the car's position $(\hat{x}, \hat{y})$ and orientation $\hat{\psi}$:

$$\begin{array}{ccc} \hat{r} \sim \mathcal{N}(r, \nu_r) & \hat{\theta} \sim \mathcal{N}(\theta, \nu_\theta) & \hat{\phi} \sim \mathcal{U}(0, 2\pi) \\ \hat{x} \leftarrow x + \hat{r}\cos\hat{\phi} & \hat{y} \leftarrow y + \hat{r}\sin\hat{\phi} & \hat{\psi} \leftarrow \hat{\phi} - \hat{\theta} + \pi \end{array}$$

where $\hat{\phi}$ is the angle of the car from the landmark in the global frame.

Example 19.1. Generating an initial non-parametric belief based on a landmark observation. In this case, the autonomous car could be anywhere in a ring around the landmark:



19.2 *Discrete State Filter*

In a POMDP, the agent does not directly observe the underlying state of the environment. Instead, the agent receives an observation, which belongs to some *observation space* $\mathcal{O}$, at each time step. The probability of observing o given the agent took action a and transitioned to state s' is given by $O(o \mid a, s')$. If $\mathcal{O}$

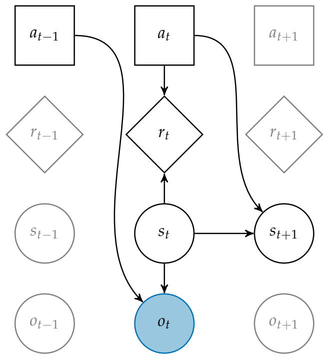


Figure 19.1. A dynamic decision network for the POMDP problem formulation.

is continuous, then $O(o \mid a, s')$ is a probability density. Figure 19.1 shows the dynamic decision network associated with POMDPs. Algorithm 19.1 provides an implementation of the POMDP data structure.

```
struct POMDP
    γ   # discount factor
    𝒮   # state space
    𝒜   # action space
    𝒪   # observation space
    T   # transition function
    R   # reward function
    O   # observation function
    TRO # sample transition, reward, and observation
end
```

Algorithm 19.1. A data structure for POMDPs. We will use the `TRO` field to sample the next state, reward, and observation given the current state and action: `s′, r, o = TRO(s, a)`.

A kind of inference known as *recursive Bayesian estimation* can be used to update our belief distribution over the current state given the most recent action and observations. We use $b(s)$ to represent the probability (or probability density for continuous state spaces) assigned to state s. A particular belief b belongs to a *belief space* $\mathcal{B}$, which contains all possible beliefs.

When the state and observation spaces are finite, we can use a *discrete state filter* to perform this inference exactly. Beliefs for problems with discrete state spaces can be represented using categorical distributions, where a probability mass is assigned to each state. This categorical distribution can be represented as a vector of length $|\mathcal{S}|$ and is often called a *belief vector*. In cases where b can be treated as a vector, we will use $\mathbf{b}$. In this case, $\mathcal{B} \subset \mathbb{R}^{|\mathcal{S}|}$. Sometimes $\mathcal{B}$ is referred to as a *probability simplex* or *belief simplex*.

Because a belief vector represents a probability distribution, the elements must be strictly non-negative and must sum to one:

$$b(s) \geq 0 \text{ for all } s \in \mathcal{S} \qquad \sum_s b(s) = 1 \tag{19.1}$$

In vector form, we have

$$\mathbf{b} \geq \mathbf{0} \qquad \mathbf{1}^\top \mathbf{b} = 1 \tag{19.2}$$

The belief space for a POMDP with three states is given in figure 19.2. An example discrete POMDP problem is given in example 19.2.

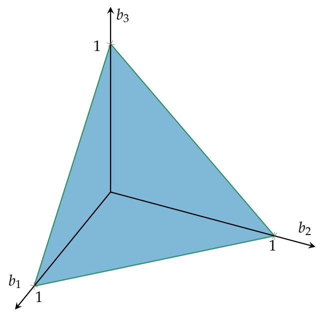


Figure 19.2. The set of valid belief vectors for problems with three states. Though the state space is discrete, the belief space is continuous.

If an agent with belief b takes an action a and receives an observation o, the new belief b' can be calculated as follows due to the independence assumptions

The *crying baby problem* is a simple POMDP with two states, three actions, and two observations. Our goal is to care for a baby, and we do so by choosing at each time step whether to feed the baby, sing to it, or ignore it.

The baby becomes hungry over time. One does not directly observe whether the baby is hungry, but instead receives a noisy observation in the form of whether or not the baby is crying. A hungry baby cries 80 % of the time, whereas a sated baby cries 10 % of the time. Singing to the baby yields a perfect observation. The state, action, and observation spaces are:

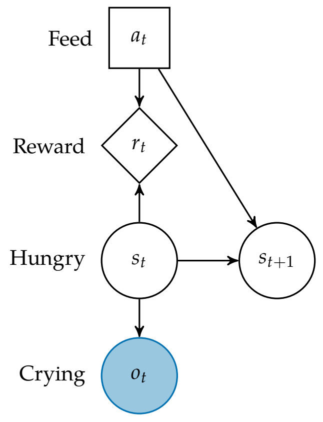
$$\begin{aligned}\mathcal{S} &= \{\text{hungry}, \text{sated}\}\\ \mathcal{A} &= \{\text{feed}, \text{sing}, \text{ignore}\}\\ \mathcal{O} &= \{\text{crying}, \text{quiet}\}\end{aligned}$$

The transition dynamics are:

$$\begin{aligned}T(\text{sated} \mid \text{hungry}, \text{feed}) &= 100\,\%\\ T(\text{hungry} \mid \text{hungry}, \text{sing}) &= 100\,\%\\ T(\text{hungry} \mid \text{hungry}, \text{ignore}) &= 100\,\%\\ T(\text{sated} \mid \text{sated}, \text{feed}) &= 100\,\%\\ T(\text{hungry} \mid \text{sated}, \text{sing}) &= 10\,\%\\ T(\text{hungry} \mid \text{sated}, \text{ignore}) &= 10\,\%\end{aligned}$$

The reward function assigns -10 reward if the baby is hungry and -5 reward for feeding the baby. Thus, feeding a hungry baby results in -15 reward. Singing to a sated baby makes it happy to receive the attention, and thus yields 5 reward. Singing to a hungry baby takes extra effort, and thus incurs -2 reward. As baby caretakers, we seek the optimal infinite horizon policy with discount factor $\gamma = 0.9$.

Example 19.2. The crying baby problem is a simple POMDP used to demonstrate decision making with state uncertainty.



in figure 19.1:

$$
\begin{aligned}
b'(s') &= P(s' \mid b, a, o) && (19.3)\\
&\propto P(o \mid b, a, s')P(s' \mid b, a) && (19.4)\\
&\propto O(o \mid a, s')P(s' \mid b, a) && (19.5)\\
&\propto O(o \mid a, s')\sum_s P(s' \mid a, b, s)P(s \mid b, a) && (19.6)\\
&\propto O(o \mid a, s')\sum_s T(s' \mid s, a)b(s) && (19.7)
\end{aligned}
$$

An example of updating discrete beliefs is given in example 19.3, and the belief update is implemented in algorithm 19.2. The success of the belief update depends upon having accurate observation and transition models. In cases where these models are not known well, it is generally advisable to use simplified models with more diffuse distributions to help prevent overconfidence that leads to brittleness in the state estimates.

19.3 Linear Gaussian Filter

We can adapt equation (19.7) to handle continuous state spaces as follows:

$$b'(s') \propto O(o \mid a, s') \int T(s' \mid s, a)b(s)\,\mathrm{d}s \qquad (19.8)$$

The integration above can be challenging unless we make some assumptions about the form of T, O, and b. A special type of filter known as the *Kalman filter* (algorithm 19.3)[2] provides an exact update under the assumption that T and O are linear–Gaussian and b is Gaussian:[3]

$$
\begin{aligned}
T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) &= \mathcal{N}(\mathbf{s}' \mid \mathbf{T}_s\mathbf{s} + \mathbf{T}_a\mathbf{a}, \mathbf{\Sigma}_s) && (19.9)\\
O(\mathbf{o} \mid \mathbf{s}') &= \mathcal{N}(\mathbf{o} \mid \mathbf{O}_s\mathbf{s}', \mathbf{\Sigma}_o) && (19.10)\\
b(\mathbf{s}) &= \mathcal{N}(\mathbf{s} \mid \boldsymbol{\mu}_b, \mathbf{\Sigma}_b) && (19.11)
\end{aligned}
$$

The Kalman filter begins with a *predict step*, which uses the transition dynamics to get a predicted distribution with the following mean and covariance:

$$
\begin{aligned}
\boldsymbol{\mu}_p &\leftarrow \mathbf{T}_s\boldsymbol{\mu}_b + \mathbf{T}_a\mathbf{a} && (19.12)\\
\mathbf{\Sigma}_p &\leftarrow \mathbf{T}_s\mathbf{\Sigma}_b\mathbf{T}_s^\top + \mathbf{\Sigma}_s && (19.13)
\end{aligned}
$$

[2] Named after the Hungarian-American electrical engineer Rudolf E. Kálmán (1930–2016) who was involved in the early development of this filter.

[3] R. E. Kálmán, "A New Approach to Linear Filtering and Prediction Problems," *ASME Journal of Basic Engineering*, vol. 82, pp. 35–45, 1960. A comprehensive overview of the Kalman filter and its variants is provided by Y. Bar-Shalom, X. R. Li, and T. Kirubarajan, *Estimation with Applications to Tracking and Navigation*. Wiley, 2001.

The crying baby problem (example 19.2) assumes a uniform initial belief state: $[b(\text{sated}), b(\text{hungry})] = [0.5, 0.5]$.

Suppose we ignore the baby and the baby cries. We update our belief according to equation (19.7):

$$\begin{aligned} b'(\text{sated}) &\propto O(\text{crying} \mid \text{ignore}, \text{sated}) \sum_s T(\text{sated} \mid s, \text{ignore}) b(s) \\ &\propto 0.1(0.0 \cdot 0.5 + 0.9 \cdot 0.5) \\ &\propto 0.045 \\ b'(\text{hungry}) &\propto O(\text{crying} \mid \text{ignore}, \text{hungry}) \sum_s T(\text{hungry} \mid s, \text{ignore}) b(s) \\ &\propto 0.8(1.0 \cdot 0.5 + 0.1 \cdot 0.5) \\ &\propto 0.440 \end{aligned}$$

After normalizing, our new belief is approximately $[0.0928, 0.9072]$. A crying baby is likely to be hungry.

Suppose we then feed the baby and the crying stops. Feeding deterministically caused the baby to be sated, so the new belief is $[1, 0]$.

Finally, we sing to the baby and the baby is quiet. Equation (19.7) is used again to update the belief, resulting in $[0.9759, 0.0241]$. A sated baby only becomes hungry 10 % of the time, and this percentage is further reduced by not observing any crying.

Example 19.3. Discrete belief updating in the crying baby problem.

```
function update(b::Vector{Float64}, 𝒫, a, o)
    𝒮, T, O = 𝒫.𝒮, 𝒫.T, 𝒫.O
    b′ = similar(b)
    for (i′, s′) in enumerate(𝒮)
        po = O(a, s′, o)
        b′[i′] = po * sum(T(s, a, s′) * b[i] for (i, s) in enumerate(𝒮))
    end
    if sum(b′) ≈ 0.0
        fill!(b′, 1)
    end
    return normalize!(b′, 1)
end
```

Algorithm 19.2. A method that updates a discrete belief based on equation (19.7), where `b` is a floating point vector. If the given observation has a zero likelihood, a uniform distribution is returned.

In the *update step*, we use this predicted distribution with the current observation to update our belief:

$$\mathbf{K} \leftarrow \mathbf{\Sigma}_p \mathbf{O}_s^\top \left(\mathbf{O}_s \mathbf{\Sigma}_p \mathbf{O}_s^\top + \mathbf{\Sigma}_o\right)^{-1} \tag{19.14}$$

$$\boldsymbol{\mu}_b \leftarrow \boldsymbol{\mu}_p + \mathbf{K}\left(\mathbf{o} - \mathbf{O}_s \boldsymbol{\mu}_p\right) \tag{19.15}$$

$$\mathbf{\Sigma}_b \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\mathbf{\Sigma}_p \tag{19.16}$$

where $\mathbf{K}$ is called the *Kalman gain*.

```
struct KalmanFilter
    μb # mean vector
    Σb # covariance matrix
end

function update(b::KalmanFilter, 𝒫, a, o)
    μb, Σb = b.μb, b.Σb
    Ts, Ta, Os = 𝒫.Ts, 𝒫.Ta, 𝒫.Os
    Σs, Σo = 𝒫.Σs, 𝒫.Σo
    # predict
    μp = Ts*μb + Ta*a
    Σp = Ts*Σb*Ts' + Σs
    # update
    K = Σp*Os'/(Os*Σp*Os' + Σo)
    μb′ = μp + K*(o - Os*μp)
    Σb′ = (I - K*Os)*Σp
    return KalmanFilter(μb′, Σb′)
end
```

Algorithm 19.3. The Kalman filter, which updates beliefs in the form of Gaussian distributions. The current belief is represented by μb and Σb, and the POMDP contains matrices that define linear Gaussian dynamics.

Kalman filters are often applied to systems that do not actually have linear–Gaussian dynamics and observations. A variety of different modifications to the basic Kalman filter have been proposed to better accommodate such systems.[4]

[4] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

19.4 *Extended Kalman Filter*

The *extended Kalman filter* (*EKF*), is a simple extension of the Kalman filter to problems whose dynamics are nonlinear with Gaussian noise:

$$T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}(\mathbf{s}' \mid \mathbf{f}_T(\mathbf{s}, \mathbf{a}), \mathbf{\Sigma}_s) \tag{19.17}$$

$$O(\mathbf{o} \mid \mathbf{s}') = \mathcal{N}(\mathbf{o} \mid \mathbf{f}_O(\mathbf{s}'), \mathbf{\Sigma}_o) \tag{19.18}$$

where $\mathbf{f}_T(\mathbf{s}, \mathbf{a})$ and $\mathbf{f}_O(\mathbf{s}')$ are differentiable functions.

Consider the following linear–Gaussian transition and observation functions:

$$T(\mathbf{s}' \mid \mathbf{s}, \mathbf{a}) = \mathcal{N}\left(\mathbf{s}' \mid \mathbf{s} + \mathbf{a}, \frac{1}{10}\begin{bmatrix} 1 & 1/2 \\ 1/2 & 1 \end{bmatrix}\right)$$
$$O(\mathbf{o} \mid \mathbf{s}') = \mathcal{N}\left(\mathbf{o} \mid \mathbf{s}', \frac{1}{20}\begin{bmatrix} 1 & -1/2 \\ -1/2 & 2 \end{bmatrix}\right)$$

Suppose our initial belief is $\mathbf{b} = \mathcal{N}([-0.75, 1], \mathbf{I})$. We take action $\mathbf{a} = [0.5, -0.5]$ and observe $\mathbf{o} = [0.3, 0.5]$.

Using the Kalman filter update equations, our new belief is:

$$\mathbf{b}' = \mathcal{N}\left(\begin{bmatrix} 0.184 \\ 0.571 \end{bmatrix}, \begin{bmatrix} 0.037 & -0.011 \\ -0.011 & 0.050 \end{bmatrix}\right)$$

using the Kalman gain:

$$\mathbf{K} = \begin{bmatrix} 0.789 & 0.110 \\ 0.128 & 0.716 \end{bmatrix}$$

Note that because the observation dynamics are more certain, the new belief is centered closer to the observation than to the ideal $\mathbf{s}'$ predicted by the transition dynamics applied to the previous belief mean and action.

Example 19.4. Using a Kalman filter to update the belief in a linear–Gaussian POMDP. While in this problem the state, action, and observations are all in $\mathbb{R}^2$, there is no requirement that they have the same dimensionality.

Exact belief updates through nonlinear dynamics are not guaranteed to produce new Gaussian beliefs, as shown in figure 19.3. The extended Kalman filter uses a local linear approximation to the nonlinear dynamics, thereby producing a new Gaussian belief that approximates the true updated belief. We can use similar update equations as the Kalman filter, but must compute the matrices $\mathbf{T}_s$ and $\mathbf{O}_s$ at every iteration based on our current belief.

The local linear approximation to the dynamics, or *linearization*, is given by first-order Taylor expansions in the form of Jacobians.[5] For the state matrix, the Taylor expansion is conducted at $\boldsymbol{\mu}_b$ and the current action, whereas for the observation matrix, it is computed at the predicted mean $\boldsymbol{\mu}_p = \mathbf{f}_T(\boldsymbol{\mu}_b)$.

[5] The Jacobian of a multivariate function $\mathbf{f}$ with n inputs and m outputs is an $m \times n$ matrix where the (i, j)th entry is $\partial f_i / \partial x_j$.

The extended Kalman filter is implemented in algorithm 19.4. Although it is an approximation, it is fast and performs well on a variety of real-world problems. The EKF does not generally preserve the true mean and variance of the posterior, and it does not model multimodal posterior distributions.

```julia
struct ExtendedKalmanFilter
    μb # mean vector
    Σb # covariance matrix
end

import ForwardDiff: jacobian
function update(b::ExtendedKalmanFilter, 𝒫, a, o)
    μb, Σb = b.μb, b.Σb
    fT, fO = 𝒫.fT, 𝒫.fO
    Σs, Σo = 𝒫.Σs, 𝒫.Σo
    # predict
    μp = fT(μb, a)
    Ts = jacobian(s->fT(s, a), μb)
    Os = jacobian(fO, μp)
    Σp = Ts*Σb*Ts' + Σs
    # update
    K = Σp*Os'/(Os*Σp*Os' + Σo)
    μb′ = μp + K*(o - fO(μp))
    Σb′ = (I - K*Os)*Σp
    return ExtendedKalmanFilter(μb′, Σb′)
end
```

Algorithm 19.4. The extended Kalman filter, an extension of the Kalman filter to problems with nonlinear Gaussian dynamics. The current belief is represented by mean `μb` and covariance `Σb`, and the POMDP defines the nonlinear dynamics via the mean transition dynamics `fT` and mean observation dynamics `fO`. The Jacobians are obtained using the `ForwardDiff.jl` package.

19.5 *Unscented Kalman Filter*

The *unscented Kalman filter* (*UKF*)[6] is another extension to the Kalman filter to problems that are nonlinear with Gaussian noise.[7] Unlike the extended Kalman

[6] S. J. Julier and J. K. Uhlmann, "Unscented Filtering and Nonlinear Estimation," *Proceedings of the IEEE*, vol. 92, no. 3, pp. 401–422, 2004.

[7] According to Uhlmann, the term "unscented" comes from a label on a deodorant container that he saw on someone's desk. He used that term to avoid calling it the Uhlmann filter. For a historical perspective, see `ethw.org/First-Hand:The_Unscented_Transform`.

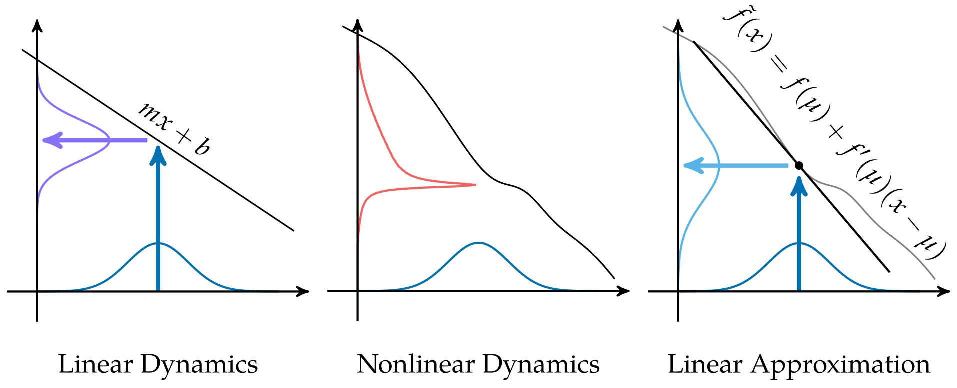


Figure 19.3. Updating a Gaussian belief with a linear transform (left) produces another Gaussian distribution. Updating a Gaussian belief with a nonlinear transform (center) does not in general produce a Gaussian distribution. The extended Kalman filter uses a linear approximation of the transform (right), thereby producing another Gaussian distribution that approximates the posterior.

filter, the unscented Kalman filter is derivative-free, and relies on a deterministic sampling strategy to approximate the effect of a distribution undergoing a (typically nonlinear) transformation.

The unscented Kalman filter was developed to estimate the effect of passing a random variable $\mathbf{x}$ through a nonlinear distribution $\mathbf{f}(\mathbf{x})$, producing the random variable $\mathbf{x}'$. We would like to estimate the mean $\boldsymbol{\mu}'$ and covariance $\boldsymbol{\Sigma}'$ of the distribution over $\mathbf{x}'$. The unscented transform allows for more information of $p(\mathbf{x})$ to be used than the mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$ of the distribution over $\mathbf{x}$.[8]

[8] We need not necessarily assume that the prior distribution is Gaussian.

An *unscented transform* passes a set of *sigma points* S through $\mathbf{f}$ and uses the transformed points to approximate the transformed mean $\boldsymbol{\mu}'$ and covariance $\boldsymbol{\Sigma}'$. The mean and covariance are reconstructed using a vector of weights $\mathbf{w}$:

$$\boldsymbol{\mu}' = \sum_i w_i \mathbf{s}_i \tag{19.19}$$

$$\boldsymbol{\Sigma}' = \sum_i w_i (\mathbf{s}_i - \boldsymbol{\mu}')(\mathbf{s}_i - \boldsymbol{\mu}')^\top \tag{19.20}$$

These weights must sum to 1 in order to provide an unbiased estimate, but need not all be positive.

The updated mean and covariance matrix given by the unscented transform through $\mathbf{f}$ are thus:

$$\boldsymbol{\mu}' = \sum_i w_i \mathbf{f}(\mathbf{s}_i) \tag{19.21}$$

$$\boldsymbol{\Sigma}' = \sum_i w_i \big(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\big)\big(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\big)^\top \tag{19.22}$$

An example set of sigma points and associated weight vector is given in example 19.5.

A simple set of $2n$ sigma points for an n-dimensional distribution is given by:

$$\mathbf{s}_{2i} = \boldsymbol{\mu} + \sqrt{n\boldsymbol{\Sigma}}_i \text{ for } i \text{ in } 1:n$$
$$\mathbf{s}_{2i-1} = \boldsymbol{\mu} - \sqrt{n\boldsymbol{\Sigma}}_i \text{ for } i \text{ in } 1:n$$

where the i subscript indicates the ith column of the square root matrix.

If we use the weights $w_i = 1/2n$, the reconstructed mean is:

$$\sum_i w_i \mathbf{s}_i = \sum_{i=1}^n \frac{1}{2n}\left(\boldsymbol{\mu} + \sqrt{n\boldsymbol{\Sigma}}_i\right) + \frac{1}{2n}\left(\boldsymbol{\mu} - \sqrt{n\boldsymbol{\Sigma}}_i\right) = \sum_{i=1}^n \frac{1}{n}\boldsymbol{\mu} = \boldsymbol{\mu}$$

and the reconstructed covariance is:

$$\begin{aligned}\sum_i w_i(\mathbf{s}_i - \boldsymbol{\mu}')(\mathbf{s}_i - \boldsymbol{\mu}')^\top &= 2\sum_{i=1}^n \frac{1}{2n}\left(\sqrt{n\boldsymbol{\Sigma}}_i\right)\left(\sqrt{n\boldsymbol{\Sigma}}_i\right)^\top \\ &= \frac{1}{n}\sum_{i=1}^n \left(\sqrt{n\boldsymbol{\Sigma}}_i\right)\left(\sqrt{n\boldsymbol{\Sigma}}_i\right)^\top \\ &= \sqrt{\boldsymbol{\Sigma}}\sqrt{\boldsymbol{\Sigma}}^\top \\ &= \boldsymbol{\Sigma}\end{aligned}$$

Example 19.5. A simple set of sigma points and weights that can be used to reconstruct the mean and covariance of a distribution.

The square root of a matrix $\mathbf{A}$ is a matrix $\mathbf{B}$ such that $\mathbf{B}\mathbf{B}^\top = \mathbf{A}$. In Julia, the `sqrt` method produces a matrix $\mathbf{C}$ such that $\mathbf{C}\mathbf{C} = \mathbf{A}$, which is not the same. One common square root matrix can be obtained from the Cholesky decomposition.

Sigma points and weights can be selected to incorporate additional information about the prior distribution. Suppose we always have a set of sigma points and weights that produce a mean $\boldsymbol{\mu}'$ and covariance $\boldsymbol{\Sigma}'$. We can construct a new set of sigma points by additionally including the mean $\boldsymbol{\mu}'$ as a sigma point. We scale the sigma points such that the covariance is preserved. This results in a scaled set of sigma points with different higher-order moments, but the same mean and covariance. Example 19.6 derives a set of sigma points by applying this technique to the sigma points in example 19.5.

We can include the mean $\boldsymbol{\mu}$ in the sigma points from example 19.5 to obtain a new set of $2n+1$ sigma points:

$$\begin{aligned}
\mathbf{s}_1 &= \boldsymbol{\mu} \\
\mathbf{s}_{2i} &= \boldsymbol{\mu} + \left(\sqrt{\frac{n}{1-w_1}\boldsymbol{\Sigma}}\right)_i \text{ for } i \text{ in } 1:n \\
\mathbf{s}_{2i+1} &= \boldsymbol{\mu} - \left(\sqrt{\frac{n}{1-w_1}\boldsymbol{\Sigma}}\right)_i \text{ for } i \text{ in } 1:n
\end{aligned}$$

where w_1 is the weight on the first sigma point. The weights on the remaining sigma points are $(1-w_1)/2n$. The reconstructed mean is still $\boldsymbol{\mu}$, and the reconstructed covariance is still $\boldsymbol{\Sigma}$.

We can vary w_1 to produce different sets of sigma points. Setting $w_1 > 0$ causes the sigma points to spread away from the mean. Setting $w_1 < 0$ moves the sigma points closer to the mean.

Example 19.6. An example of how a set of sigma points can be parameterized, allowing the designer to capture additional information about the prior distribution.

A common set of sigma points include the mean $\boldsymbol{\mu} \in \mathbb{R}^n$ and an additional $2n$ points formed from perturbations of $\boldsymbol{\mu}$ in directions determined by the covariance matrix $\boldsymbol{\Sigma}$:

$$\mathbf{s}_1 = \boldsymbol{\mu} \tag{19.23}$$

$$\mathbf{s}_{2i} = \boldsymbol{\mu} + \left(\sqrt{(n+\lambda)\boldsymbol{\Sigma}}\right)_i \text{ for } i \text{ in } 1:n \tag{19.24}$$

$$\mathbf{s}_{2i+1} = \boldsymbol{\mu} - \left(\sqrt{(n+\lambda)\boldsymbol{\Sigma}}\right)_i \text{ for } i \text{ in } 1:n \tag{19.25}$$

These sigma points are associated with the weights:

$$w_i = \begin{cases} \frac{\lambda}{n+\lambda} & \text{for } i = 1 \\ \frac{1}{2(n+\lambda)} & \text{otherwise} \end{cases} \tag{19.26}$$

The scalar *spread parameter* λ determines how far the sigma points are spread from the mean. It is recommended to use $\lambda = 2$, which is optimal for matching the fourth moment of Gaussian distributions. Several sigma point sets for different values of λ are shown in figure 19.4.

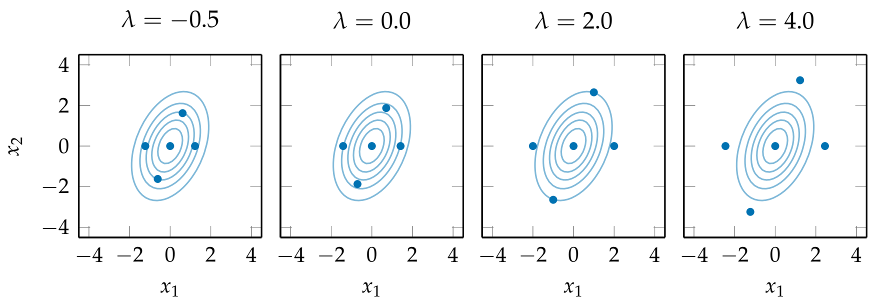


Figure 19.4. The effect of varying λ on the sigma points from equation (19.23) generated for a Gaussian distribution with zero mean and covariance `Σ = [1 1/2; 1/2 2]`.

The unscented Kalman filter performs two unscented transformations, one for the prediction step and one for the observation update. The original Kalman filter computes the Kalman gain using the observation covariance $\mathbf{O}_s$. The unscented Kalman filter does not have this observation matrix, and instead calculates the Kalman gain using a *cross covariance matrix*[9] using both prediction values and observation values. Algorithm 19.5 provides an implementation.

[9] A covariance matrix measures the variance between components of the same multi-dimensional variable. A cross covariance matrix measures the variance between two multi-dimensional variables. In algorithm 19.5, we compute a cross covariance matrix `Σpo`.

19.6 Particle Filter

Discrete problems with large state spaces or continuous problems with dynamics that are not well approximated by the linear–Gaussian assumption of the Kalman filter must often resort to approximation techniques to represent the belief and to perform the belief update. One common approach is to use a *particle filter*, which represents the belief state as a collection of states.[10] Each state in the approximate belief is called a *particle*.

[10] A tutorial on particle filters is provided by M. S. Arulampalam, S. Maskell, N. Gordon, and T. Clapp, "A Tutorial on Particle Filters for Online Nonlinear / Non-Gaussian Bayesian Tracking," *IEEE Transactions on Signal Processing*, vol. 50, no. 2, pp. 174–188, 2002.

A particle filter is initialized by selecting or randomly sampling a collection of particles that represent the initial belief. The belief update for a particle filter with m particles begins by propagating each state s_i by sampling from the transition distribution to obtain a new state s_i' with probability $T(s_i' \mid s_i, a)$. The new belief is constructed by drawing m particles from the propagated states weighted according to the observation function $w_i = O(o \mid a, s')$. This procedure is given in algorithm 19.6. Example 19.7 illustrates an application of a particle filter.

```
struct UnscentedKalmanFilter
    μb # mean vector
    Σb # covariance matrix
    λ  # spread parameter
end

function unscented_transform(μ, Σ, f, λ, ws)
    n = length(μ)
    Δ = sqrt((n + λ) * Σ)
    S = [μ]
    for i in 1:n
        push!(S, μ + Δ[:,i])
        push!(S, μ - Δ[:,i])
    end
    S′ = f.(S)
    μ′ = sum(w*s for (w,s) in zip(ws, S′))
    Σ′ = sum(w*(s - μ′)*(s - μ′)' for (w,s) in zip(ws, S′))
    return (μ′, Σ′, S, S′)
end

function update(b::UnscentedKalmanFilter, 𝒫, a, o)
    μb, Σb, λ = b.μb, b.Σb, b.λ
    fT, fO = 𝒫.fT, 𝒫.fO
    n = length(μb)
    ws = [λ / (n + λ); fill(1/(2*(n + λ)), 2n)]
    # predict
    μp, Σp, Sp, Sp′ = unscented_transform(μb, Σb, s->fT(s,a), λ, ws)
    Σp += 𝒫.Σs
    # update
    μo, Σo, So, So′ = unscented_transform(μp, Σp, fO, λ, ws)
    Σo += 𝒫.Σo
    Σpo = sum(w*(s - μp)*(s′ - μo)' for (w,s,s′) in zip(ws, So, So′))
    K = Σpo / Σo
    μb′ = μp + K*(o - μo)
    Σb′ = Σp - K*Σo*K'
    return UnscentedKalmanFilter(μb′, Σb′, λ)
end
```

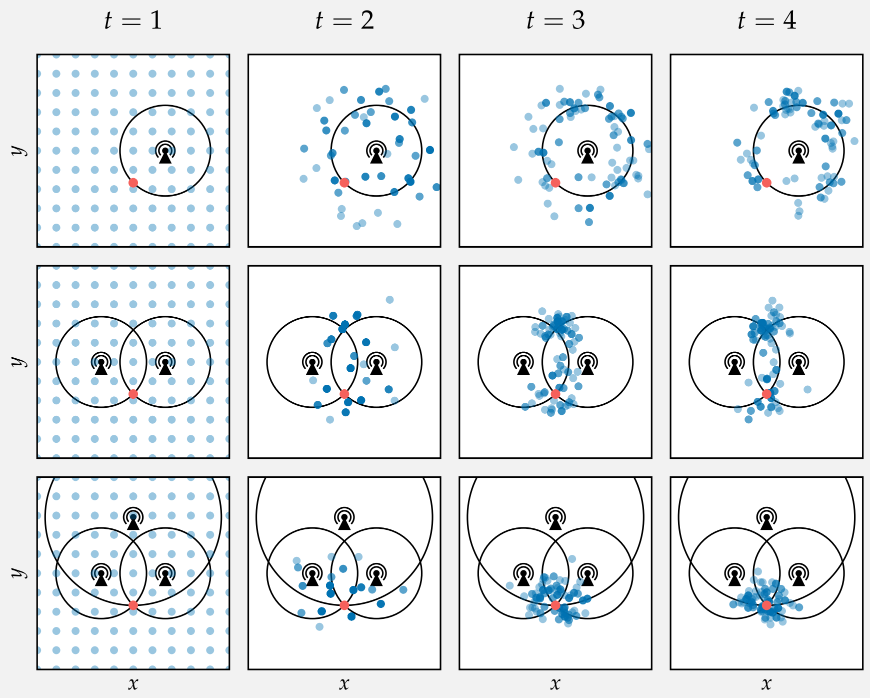
Algorithm 19.5. The unscented Kalman filter, an extension of the Kalman filter to problems with nonlinear Gaussian dynamics. The current belief is represented by mean `μb` and covariance `Σb`. The POMDP parameterizes the nonlinear dynamics with the mean transition dynamics `fT` and mean observation dynamics `fO`. The sigma points used in the unscented transforms are controlled by the scalars `α`, `β`, and `κ`.

Example 19.7. A particle filter applied to different beacon configurations.

Suppose we want to determine our position based on imperfect distance measurements to radio beacons whose locations are known. We remain approximately still for a few steps to collect independent measurements. The particle filter states are our potential locations. We can compare the ranges we would expect to measure for each particle to the observed ranges.

We assume individual range observations from each beacon are observed with zero-mean Gaussian noise. Our particle transition function adds zero-mean Gaussian noise since we remain only approximately still.

The images below show the evolution of the particle filter. The rows correspond to different numbers of beacons. The red dot indicates our true location and the blue dots are particles. The circles indicate the positions consistent with noiseless distance measurements from each sensor.



Three beacons are required to accurately identify our location. A strength of the particle filter is that it is able to represent the multimodal distributions that are especially apparent when there are only one or two beacons.

```
struct ParticleFilter
    states # vector of state samples
end

function update(b::ParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    D = SetCategorical(states, weights)
    return ParticleFilter(rand(D, length(states)))
end
```

Algorithm 19.6. A belief updater for particle filters, which updates a vector of states representing the belief based on the agent's action `a` and its observation `o`.

In problems with discrete observations, we can also perform particle belief updates with rejection. Here, each state s_i in the filter is propagated to obtain successor states s_i' as before. However, observations o_i are simultaneously sampled for each successor state. Any o_i that does not equal the true observation o is rejected, and a new s_i' and o_i are sampled until the observations match. This *particle filter with rejection* is implemented in algorithm 19.7.

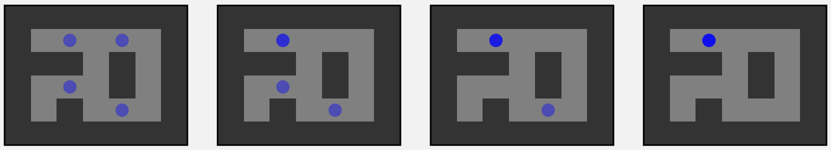
```
struct RejectionParticleFilter
    states # vector of state samples
end

function update(b::RejectionParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    states = similar(b.states)
    i = 1
    while i ≤ length(states)
        s = rand(b.states)
        s′ = rand(T(s,a))
        if rand(O(a,s′)) == o
            states[i] = s′
            i += 1
        end
    end
    return RejectionParticleFilter(states)
end
```

Algorithm 19.7. Updating a particle filter with rejection, which forces sampled states to match the input observation `o`.

As the number of particles in a particle filter increases, the distribution represented by the particles approaches the true posterior distribution. Unfortunately, particle filters can fail in practice. Low particle coverage and the stochastic nature of the resampling procedure can cause there to be no particles near the true state. This problem of *particle deprivation* can be somewhat mitigated by several strategies. A motivational example is given in example 19.8.

Spelunker Joe is lost in a grid-based maze. He lost his lantern, so can only observe his surroundings by touch. At any given moment, Joe can tell whether his location in the maze has walls in each cardinal direction. Joe is fairly confident in his ability to feel walls, so he assumes his observations are perfect.

Joe uses a particle filter to track his belief over time. At some point he stops to rest. He continues to run his particle filter to update his belief.



The initial belief has one particle in each grid location that matches his current observation of a wall to the north and south. Spelunker Joe does not move and does not gain new information, so his belief should not change over time. Due to the stochastic nature of resampling, subsequent beliefs may not contain all of the initial states. Over time, his belief will continue to lose states until it only contains a single state. It is possible that this state is not where Spelunker Joe is located.

Example 19.8. A particle filter run for enough time can lose particles in relevant regions of the state space due to the stochastic nature of resampling. The problem is more pronounced with fewer particles or when the particles are spread over a large state space.

19.7 Particle Injection

Particle injection involves injecting random particles to protect against particle deprivation. Algorithm 19.8 injects a fixed number of particles from a broader distribution, such as a uniform distribution over the state space.[11] While particle injection can help prevent particle deprivation, it also reduces the accuracy of the posterior belief represented by the particle filter.

[11] For robotic localization problems, it is common practice to inject particles from a uniform distribution over all possible robot poses, weighted by the current observation.

Instead of using a fixed number of injected particles at each update, we can take a more adaptive approach. When the particles are all being given very low weight, then we generally want to inject more particles. It might be tempting to choose the number of injected particles based solely on the mean weight of

```
struct InjectionParticleFilter
    states # vector of state samples
    m_inject # number of samples to inject
    D_inject # injection distribution
end

function update(b::InjectionParticleFilter, 𝒫, a, o)
    T, O, m_inject, D_inject = 𝒫.T, 𝒫.O, b.m_inject, b.D_inject
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    D = SetCategorical(states, weights)
    m = length(states)
    states = vcat(rand(D, m - m_inject), rand(D_inject, m_inject))
    return InjectionParticleFilter(states, m_inject, D_inject)
end
```

Algorithm 19.8. Particle filter update with injection, in which `m_inject` particles are sampled from the injection distribution `D_inject` to reduce the risk of particle deprivation.

the current set of particles. However, doing so can make the success of the filter sensitive to naturally low observation probabilities in the early periods when the filter is still converging or in moments of high sensor noise.[12]

Algorithm 19.9 presents an *adaptive injection* algorithm that keeps track of two exponential moving averages of the mean particle weight and bases the number of injections on their ratio.[13] If w_{mean} is the current mean particle weight, the two moving averages are updated according to

$$w_{\text{fast}} \leftarrow w_{\text{fast}} + \alpha(w_{\text{mean}} - w_{\text{fast}}) \tag{19.27}$$

$$w_{\text{slow}} \leftarrow w_{\text{slow}} + \alpha(w_{\text{mean}} - w_{\text{slow}}) \tag{19.28}$$

where $0 \leq \alpha_{\text{slow}} < \alpha_{\text{fast}} \leq 1$.

The number of injected samples in a given iteration is obtained by comparing the fast and slow mean particle weights:[14]

$$m_{\text{inject}} = \left\lfloor m \max\left(0, 1 - \nu \frac{w_{\text{fast}}}{w_{\text{slow}}}\right)\right\rceil \tag{19.29}$$

The scalar $\nu \geq 1$ allows us to tune the injection rate.

19.8 Summary

- Partially observable Markov decision processes (POMDPs) extend MDPs to include state uncertainty.

[12] S. Thrun, W. Burgard, and D. Fox, *Probabilistic Robotics*. MIT Press, 2006.

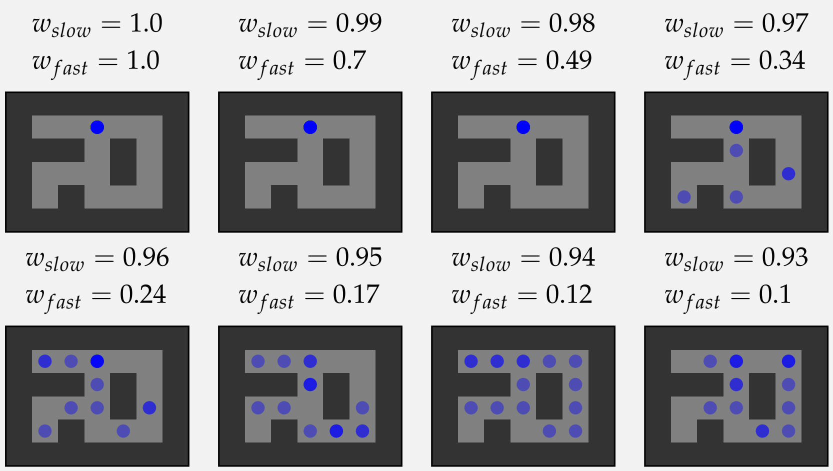
[13] D. E. Goldberg and J. Richardson, "An Experimental Comparison of Localization Methods," in *International Conference on Genetic Algorithms*, 1987.

[14] Note that $\lfloor x \rceil$ denotes the integer nearest to x.

```
mutable struct AdaptiveInjectionParticleFilter
    states   # vector of state samples
    w_slow   # slow moving average
    w_fast   # fast moving average
    α_slow   # slow moving average parameter
    α_fast   # fast moving average parameter
    ν        # injection parameter
    D_inject # injection distribution
end

function update(b::AdaptiveInjectionParticleFilter, 𝒫, a, o)
    T, O = 𝒫.T, 𝒫.O
    w_slow, w_fast, α_slow, α_fast, ν, D_inject =
        b.w_slow, b.w_fast, b.α_slow, b.α_fast, b.ν, b.D_inject
    states = [rand(T(s, a)) for s in b.states]
    weights = [O(a, s′, o) for s′ in states]
    w_mean = mean(weights)
    w_slow += α_slow*(w_mean - w_slow)
    w_fast += α_fast*(w_mean - w_fast)
    m = length(states)
    m_inject = round(Int, m * max(0, 1.0 - ν*w_fast / w_slow))
    D = SetCategorical(states, weights)
    states = vcat(rand(D, m - m_inject), rand(D_inject, m_inject))
    b.w_slow, b.w_fast = w_slow, w_fast
    return AdaptiveInjectionParticleFilter(states,
        w_slow, w_fast, α_slow, α_fast, ν, D_inject)
end
```

Algorithm 19.9. A particle filter with adaptive injection, which maintains fast and slow exponential moving averages `w_fast` and `w_slow` of the mean particle weight with smoothness factors `α_fast` and `α_slow`, respectively. Particles are only injected if the fast moving average of the mean particle weight is less than $1/\nu$ of the slow moving average. Recommended values from the original paper are `α_fast = 0.1`, `α_slow = 0.001`, and `ν = 2`.

Spelunker Joe from example 19.9 now moves one tile east and moves all particles in his particle filter one tile to the east as well. He now only senses walls to the north and east, and unfortunately, this observation does not agree with any of the updated particles in his filter. He decides to use adaptive injection to fix his particle deprivation problem. Below we see how his filter injects particles from a uniform random distribution, along with the values for the fast and slow filters.



Example 19.9. A particle filter with adaptive injection `α_slow = 0.01`, `α_fast = 0.3`, and `ν = 2.0` starting from a deprived state with 16 identical particles. The moving averages are initialized to 1 to reflect a long period of observations that perfectly match every particle in the filter. Over the next iterations, these moving averages change at different rates based on the quantity of particles that match the observation.

- The uncertainty requires agents in a POMDP to maintain a belief over their state.
- Beliefs for POMDPs with discrete state spaces can be represented using categorical distributions and can be updated analytically.
- Beliefs for linear Gaussian POMDPs can be represented using Gaussian distributions and can also be updated analytically.
- Beliefs for nonlinear continuous POMDPs can also be represented using Gaussian distributions but cannot typically be updated analytically. In this case the extended Kalman filter and the unscented Kalman filter can be used.
- Continuous problems can sometimes be modeled assuming they are linear Gaussian.
- Particle filters approximate the belief with a large collection of state particles.

19.9 Exercises

Exercise 19.1. Can every MDP be framed as a POMDP?

Solution: Yes. The POMDP formulation extends the MDP formulation by introducing state uncertainty in the form of the observation distribution. Any MDP can be framed as a POMDP with $\mathcal{O} = \mathcal{S}$ and $O(o \mid a, s') = (o = s')$.

Exercise 19.2. An autonomous vehicle represents its belief over its position using a multivariate normal distribution. It comes to a rest at a traffic light, and the belief updater continues to run while it sits. Over time, the belief concentrates and becomes extremely confident in a particular location. Why might this be a problem? How might this extreme confidence be avoided?

Solution: Yes, overconfidence in a belief can be a problem when the models or belief updates do not perfectly represent reality. The overconfident belief may have converged on a state that does not match the true state. Once the vehicle moves again, new observations may be inconsistent with the belief and result in poor estimates. To help address this issue, we can require that the values of the diagonal elements of the covariance matrix be above some threshold.

Exercise 19.3. Consider tracking our belief over the dud rate for widgets produced at a factory. We use a Poisson distribution to model the probability that k duds are produced in one day of factory operation given that the factory has a dud rate of λ:

$$P(k \mid \lambda) = \frac{1}{k!}\lambda^k e^{-\lambda} \ldots$$

Suppose that our initial belief over the dud rate follows a gamma distribution:

$$p(\lambda \mid \alpha, \beta) = \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha - 1} e^{-\beta\lambda}$$

where $\lambda \in (0, \infty)$ and the belief is parameterized by the shape $\alpha > 0$ and the rate $\beta > 0$. After a day of factory operation, we observe that $d \geq 0$ duds were produced. Show that our updated belief over the dud rate is also a gamma distribution.[15]

[15] The gamma distribution is a *conjugate prior* to the Poisson distribution. A conjugate prior is a family of probability distributions that remain within the same family when updated with an observation. Conjugate priors are useful for modeling beliefs because their form remains constant.

Solution: We seek the posterior distribution $p(\lambda \mid d, \alpha, \beta)$, which we can obtain through Bayes' rule:

$$\begin{aligned} p(\lambda \mid d, \alpha, \beta) &\propto p(d \mid \lambda) p(\lambda \mid \alpha, \beta) \\ &\propto \frac{1}{d!} \lambda^d e^{-\lambda} \; \frac{\beta^\alpha}{\Gamma(\alpha)} \lambda^{\alpha-1} e^{-\beta\lambda} \\ &\propto \lambda^{\alpha+d-1} e^{-(\beta+1)\lambda} \end{aligned}$$

This is a gamma distribution:

$$\begin{aligned} p(\lambda \mid \alpha + d, \beta + 1) &= \frac{(\beta+1)^{\alpha+d}}{\Gamma(\alpha+d)} \lambda^{\alpha+d-1} e^{-(\beta+1)\lambda} \\ &\propto \lambda^{\alpha+d-1} e^{-(\beta+1)\lambda} \end{aligned}$$

Exercise 19.4. Why are particle filters with rejection not used for updating beliefs in POMDPs with continuous observations?

Solution: Rejection sampling requires repeatedly sampling the transition and observation functions until the sampled observation matches the true observation. The probability of sampling any particular value in a continuous probability distribution is zero, making rejection sampling run forever. In practice, we would use a finite representation for continuous values such as 64-bit floating point numbers, but rejection sampling can run for an extremely long time for each particle.

Exercise 19.5. Explain why Spelunker Joe would not benefit from switching to a particle filter with adaptive injection with $\nu \geq 1$ in example 19.8.

Solution: The particle filter with adaptive injection injects new particles when $\nu w_{\text{fast}} / w_{\text{slow}}$ is less than 1. Spelunker Joe assumes perfect observations and has a belief with particles that match his current observation. Thus, every particle has a weight of 1, and so both w_{fast} and w_{slow} are 1. It follows that $w_{\text{fast}} / w_{\text{slow}}$ is always 1, leading to no new particles.

Exercise 19.6. Why is the injection rate scalar ν in a particle filter with adaptive injection typically not set to a value less than 1?

Solution: Particle injection was designed to inject particles when the current observations have lower likelihood than a historic trend over the observation likelihood. Thus, injection typically only occurs when the short-term estimate of the mean particle weight w_{fast} is less than the long-term estimate of the mean particle weight w_{slow}. If $\nu < 1$, then particles can still be generated even if $w_{\text{fast}} \geq w_{\text{slow}}$, despite indicating that current observations have higher likelihood than the past average.

Exercise 19.7. Suppose we are performing in-flight monitoring of an aircraft. The aircraft is either in a state of normal operation s^0 or a state of malfunction s^1. We receive observations through the absence of a warning w^0 or the presence of a warning w^1. We can choose to allow the plane to continue to fly m^0 or send the plane in for maintenance m^1. We have the following transition and observation dynamics, where we assume the warnings are independent of the actions given the status of the plane:

$$\begin{aligned}
T(s^0 \mid s^0, m^0) &= 0.95 & O(w^0 \mid s^0) &= 0.99 \\
T(s^0 \mid s^0, m^1) &= 1 & O(w^1 \mid s^1) &= 0.7 \\
T(s^1 \mid s^1, m^0) &= 1 \\
T(s^0 \mid s^1, m^1) &= 0.98
\end{aligned}$$

Given the initial belief $\mathbf{b} = [0.95, 0.05]$, compute the updated belief $\mathbf{b}'$ given that we allow the plane to continue to fly and we observe a warning.

Solution: Using equation (19.7), we update the belief for s^0:

$$\begin{aligned}
b'(s^0) &\propto O(w^1 \mid s^0) \sum_s T(s^0 \mid s, m^0) b(s) \\
b'(s^0) &\propto O(w^1 \mid s^0)(T(s^0 \mid s^0, m^0) b(s^0) + T(s^0 \mid s^1, m^0) b(s^1)) \\
b'(s^0) &\propto (1 - 0.99)(0.95 \times 0.95 + (1 - 1) \times 0.05) = 0.009025
\end{aligned}$$

We repeat the update for s^1:

$$\begin{aligned}
b'(s^1) &\propto O(w^1 \mid s^1) \sum_s T(s^1 \mid s, m^0) b(s) \\
b'(s^1) &\propto O(w^1 \mid s^1)(T(s^1 \mid s^0, m^0) b(s^0) + T(s^1 \mid s^1, m^0) b(s^1)) \\
b'(s^1) &\propto 0.7((1 - 0.95) \times 0.95 + 1 \times 0.05) = 0.06825
\end{aligned}$$

After normalization, we obtain the updated belief:

$$\begin{aligned}
b'(s^0) &= \frac{b'(s^0)}{b'(s^0) + b'(s^1)} \approx 0.117 \\
b'(s^1) &= \frac{b'(s^1)}{b'(s^0) + b'(s^1)} \approx 0.883 \\
b' &\approx [0.117, 0.883]
\end{aligned}$$

Exercise 19.8. Consider a robot moving along a line with position x, velocity v, and acceleration a. At each timestep, we directly control the acceleration and we observe the velocity. The equations of motion for the robot are

$$\begin{aligned} x' &= x + v\Delta t + \tfrac{1}{2}a\Delta t^2 \\ v' &= v + a\Delta t \end{aligned}$$

Suppose we would like to implement a Kalman filter to update our belief of the robot's true state. The state vector is $\mathbf{s} = [x, v]$. Determine $\mathbf{T}_s$, $\mathbf{T}_a$, and $\mathbf{O}_s$.

Solution: The transition and observation dynamics can be written in linear form as follows

$$\begin{aligned} \begin{bmatrix} x' \\ v' \end{bmatrix} &= \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} + \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} a \\ o &= \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ v \end{bmatrix} \end{aligned}$$

Through these equations, we can identify $\mathbf{T}_s$, $\mathbf{T}_a$, and $\mathbf{O}_s$

$$\mathbf{T}_s = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \qquad \mathbf{T}_a = \begin{bmatrix} \frac{1}{2}\Delta t^2 \\ \Delta t \end{bmatrix} \qquad \mathbf{O}_s = \begin{bmatrix} 0 & 1 \end{bmatrix}$$

Exercise 19.9. Compute the set of sigma points and weights with $\lambda = 2$ for a multivariate Gaussian distribution with

$$\boldsymbol{\mu} = \begin{bmatrix} 1 \\ 2 \end{bmatrix} \qquad \boldsymbol{\Sigma} = \begin{bmatrix} 4 & 0 \\ 0 & 2.25 \end{bmatrix}$$

Solution: Since we have a two-dimensional Gaussian distribution and we are given $\lambda = 2$, we need to compute $2n + 1 = 5$ sigma points. We need to compute the square-root matrix $\mathbf{B} = \sqrt{(n+\lambda)\boldsymbol{\Sigma}}$, such that $\mathbf{B}\mathbf{B}^\top = (n+\lambda)\boldsymbol{\Sigma}$. Since the scaled covariance matrix is diagonal, the square-root matrix is simply the element-wise square root of $(n+\lambda)\boldsymbol{\Sigma}$

$$\sqrt{(n+\lambda)\boldsymbol{\Sigma}} = \sqrt{(2+2)\begin{bmatrix} 4 & 0 \\ 0 & 2.25 \end{bmatrix}} = \begin{bmatrix} 4 & 0 \\ 0 & 3 \end{bmatrix}$$

Now, we can compute the sigma points and weights:

$$\begin{aligned}
\mathbf{s}_1 &= \begin{bmatrix}1\\2\end{bmatrix} & w_1 &= \frac{2}{2+2} = \frac{1}{2}\\
\mathbf{s}_2 &= \begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}4\\0\end{bmatrix} = \begin{bmatrix}5\\2\end{bmatrix} & w_2 &= \frac{1}{2(2+2)} = \frac{1}{8}\\
\mathbf{s}_3 &= \begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}4\\0\end{bmatrix} = \begin{bmatrix}-3\\2\end{bmatrix} & w_3 &= \frac{1}{2(2+2)} = \frac{1}{8}\\
\mathbf{s}_4 &= \begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0\\3\end{bmatrix} = \begin{bmatrix}1\\5\end{bmatrix} & w_4 &= \frac{1}{2(2+2)} = \frac{1}{8}\\
\mathbf{s}_5 &= \begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}0\\3\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix} & w_5 &= \frac{1}{2(2+2)} = \frac{1}{8}
\end{aligned}$$

Exercise 19.10. Using the sigma points and weights from the previous exercise, compute the updated mean and covariance given by the unscented transform through $\mathbf{f}(\mathbf{x}) = [2x_1, x_1x_2]$.

Solution: The transformed sigma points are

$$\mathbf{f}(\mathbf{s}_1) = \begin{bmatrix}2\\2\end{bmatrix} \qquad \mathbf{f}(\mathbf{s}_2) = \begin{bmatrix}10\\10\end{bmatrix} \qquad \mathbf{f}(\mathbf{s}_3) = \begin{bmatrix}-6\\-6\end{bmatrix} \qquad \mathbf{f}(\mathbf{s}_4) = \begin{bmatrix}2\\5\end{bmatrix} \qquad \mathbf{f}(\mathbf{s}_5) = \begin{bmatrix}2\\-1\end{bmatrix}$$

We can reconstruct the mean as the weighted sum of transformed sigma points

$$\begin{aligned}
\boldsymbol{\mu}' &= \sum_i w_i \mathbf{f}(\mathbf{s}_i)\\
\boldsymbol{\mu}' &= \frac{1}{2}\begin{bmatrix}2\\2\end{bmatrix} + \frac{1}{8}\begin{bmatrix}10\\10\end{bmatrix} + \frac{1}{8}\begin{bmatrix}-6\\-6\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\5\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\-1\end{bmatrix} = \begin{bmatrix}2\\2\end{bmatrix}
\end{aligned}$$

The covariance matrix can be reconstructed from the weighted sum of pointwise covariance matrices

$$\begin{aligned}
\boldsymbol{\Sigma}' &= \sum_i w_i \left(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\right)\left(\mathbf{f}(\mathbf{s}_i) - \boldsymbol{\mu}'\right)^\top\\
\boldsymbol{\Sigma}' &= \frac{1}{2}\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} = \begin{bmatrix}16 & 16\\16 & 18.25\end{bmatrix}
\end{aligned}$$