# 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.

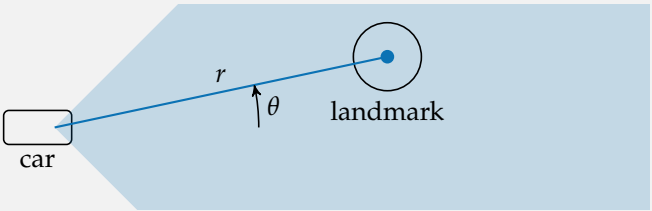
## 19.1 Belief Initialization

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 multivariate normal distribution. We will also discuss *nonparametric* representations, where the belief distribution is represented by particles, or points sampled from the state space. Associated with the different 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 belief can cause difficulties, especially for nonparametric representations of the belief, where the state space can be only very sparsely sampled. In some cases, it may be useful to wait to initialize our beliefs until an informative observation is made. 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 that 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 $\theta$ from its current pose:

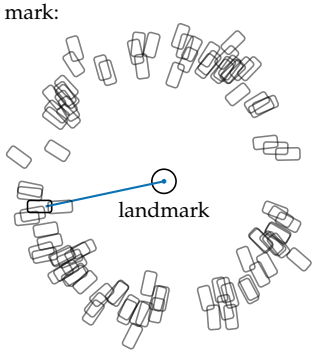



The range and bearing measurements have zero-mean Gaussian noise with variance $\nu_r$ and $\nu_\theta$, respectively, and the landmark is known to be at $(x, y)$. Given a measurement $r$ and $\theta$, we can produce a distribution over the car's position $(\hat{x}, \hat{y})$ and orientation $\hat{\psi}$:

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

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

Example 19.1. Generating an initial nonparametric 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 that the agent took action $a$ and transitioned to state $s'$, is given by $O(o \mid a, s')$. If $\mathcal{O}$

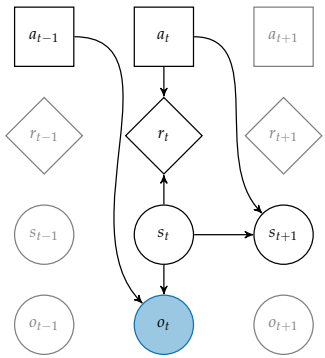



Figure 19.1. A dynamic decision network for the POMDP problem formulation. As with figure 7.1, informational edges into the action nodes are not shown.

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 comprehensive package for specifying and solving POMDPs is provided by M. Egorov, Z. N. Sunberg, E. Balaban, T. A. Wheeler, J. K. Gupta, and M. J. Kochenderfer, "POMDPs.jl: A Framework for Sequential Decision Making Under Uncertainty," *Journal of Machine Learning Research*, vol. 18, no. 26, pp. 1–5, 2017. In mathematical writing, POMDPs are sometimes defined in terms of a tuple consisting of the various components of the MDP, written as $(\mathcal{S}, \mathcal{A}, \mathcal{O}, T, R, O, \gamma)$.

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 observation. 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 nonnegative and must sum to 1:

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

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. A discrete POMDP problem is given in example 19.2.

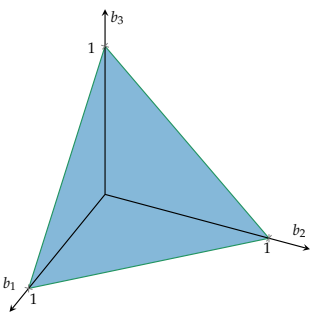
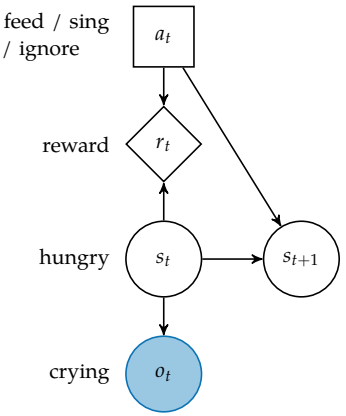



Figure 19.2. The set of valid belief vectors for problems with three states. Although the state space is discrete, the belief space is continuous.

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

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




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 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:

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

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 an additional $-5$ reward for feeding the baby because of the effort required. Thus, feeding a hungry baby results in $-15$ reward. Singing to a baby takes extra effort, and incurs a further $-0.5$ reward. As baby caretakers, we seek the optimal infinite horizon policy with discount factor $\gamma = 0.9$.

in figure 19.1:

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

An instance 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 on having accurate observation and transition models. In cases where these models are not well known, it is generally advisable to use simplified models with more diffuse distributions to help prevent overconfidence, which leads to brittleness in the state estimates.

## 19.3 Kalman Filter

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

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

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 a *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]

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

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

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

[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.

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

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) as follows:

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

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 $[0.9890, 0.0110]$. A sated baby only becomes hungry 10 % of the time, and this percentage is further reduced by not observing any crying.

Algorithm 19.2. A method that updates a discrete belief based on equation (19.7), where `b` is a vector and $\mathcal{P}$ is the POMDP model. If the given observation has a zero likelihood, a uniform distribution is returned.

```
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
```

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

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

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

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

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
    Σpo = Σp*Os'
    K = Σpo/(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 `𝒫` contains the matrices that define linear Gaussian dynamics and observation model. This `𝒫` can be defined using a composite type or a named tuple.

Kalman filters are often applied to systems that do not actually have linear Gaussian dynamics and observations. A variety of 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:

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

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

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

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 we must compute the matrices $\mathbf{T}_s$ and $\mathbf{O}_s$ at every iteration based on the 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, $\boldsymbol{\mu}_p = \mathbf{f}_T(\boldsymbol{\mu}_b)$.

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.

[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 $\partial f_i / \partial x_j$.

```
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
    Σpo = Σp*Os'
    K = Σpo/(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`. The problem `𝒫` specifies the nonlinear dynamics using the mean transition dynamics function `fT` and mean observation dynamics function `fO`. The Jacobians are obtained using the `ForwardDiff.jl` package.

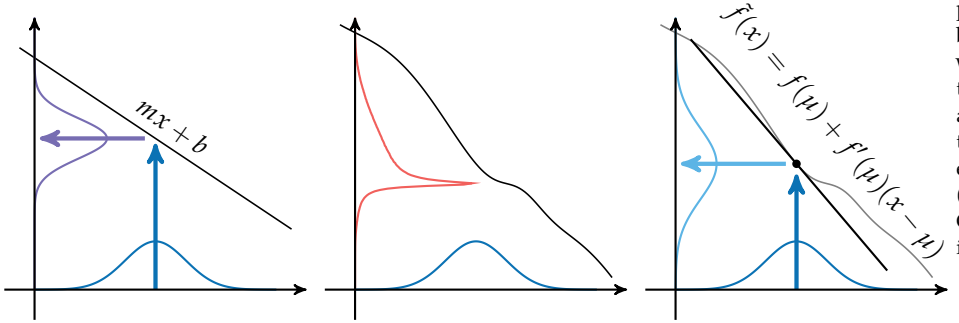



Figure 19.3. Updating a Gaussian belief with a linear transform (left) produces another Gaussian distribution. Updating a Gaussian belief with a nonlinear transform (center) does not in general produce a Gaussian distribution. The extended Kalman filter uses a linear approximation of the transform (right),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

## 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 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 transforming a distribution over $\mathbf{x}$ with a nonlinear function $\mathbf{f}(\mathbf{x})$, producing a distribution over $\mathbf{x}'$. 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]

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 original mean and covariance are constructed using the sigma points and a vector of weights $\mathbf{w}$:

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

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

[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 Jeffrey K. 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." IEEE History Center Staff, "Proceedings of the IEEE Through 100 Years: 2000–2009," *Proceedings of the IEEE*, vol. 100, no. 11, pp. 3131–3145, 2012.

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

where the $i$th sigma point $\mathbf{s}_i$ has weight $w_i$. These weights must sum to 1 in order to provide an unbiased estimate, but they need not all be positive.

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

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

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

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 $\boldsymbol{\Sigma}$:[9]

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

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

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

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* $\lambda$ determines how far the sigma points are spread from the mean.[10] Several sigma point sets for different values of $\lambda$ are shown in figure 19.4.

[9] 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.

[10] It is common to use $\lambda = 2$, which is optimal for matching the fourth moment of Gaussian distributions. Motivations for choosing sigma point sets of this form are provided in exercise 19.13 and exercise 19.14.

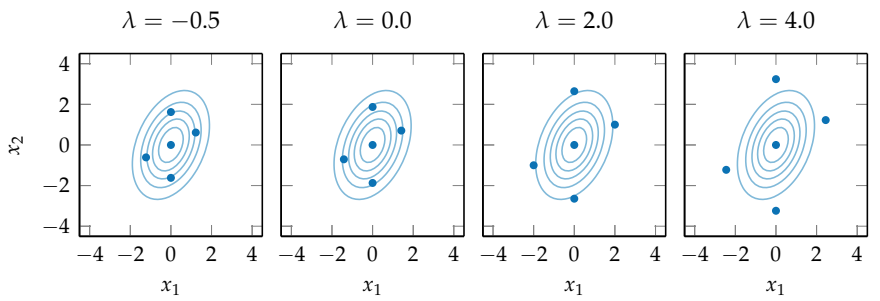



Figure 19.4. The effect of varying $\lambda$ on the sigma points from equation (19.23) generated for a Gaussian distribution with zero mean and covariance: `Σ = [1 1/2; 1/2 2]`.

The unscented Kalman filter performs two unscented transformations: one for the prediction step and one for the observation update. Algorithm 19.5 provides an implementation of this.

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

function unscented_transform(μ, Σ, f, λ, ws)
    n = length(μ)
    Δ = cholesky((n + λ) * Σ).L
    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 problem `𝒫` specifies the nonlinear dynamics using the mean transition dynamics function `fT` and mean observation dynamics function `fO`. The sigma points used in the unscented transforms are controlled by the spread parameter `λ`.

## 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.[11] Each state in the approximate belief is called a *particle*.

[11] 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 19.4 illustrates an application of a particle filter.

In problems with discrete observations, we can also perform particle belief updates with rejection. We repeat the following process $m$ times to generate the set of next state samples. First, we randomly select some state $s_i$ in the filter and then sample a next state $s_i'$ according to our transition model. Second, we generate a random observation $o_i$ according to our observation model. If $o_i$ does not equal the true observation $o$, it is rejected, and we generate a new $s_i'$ and $o_i$ until the observations match. This *particle filter with rejection* is implemented in algorithm 19.7.

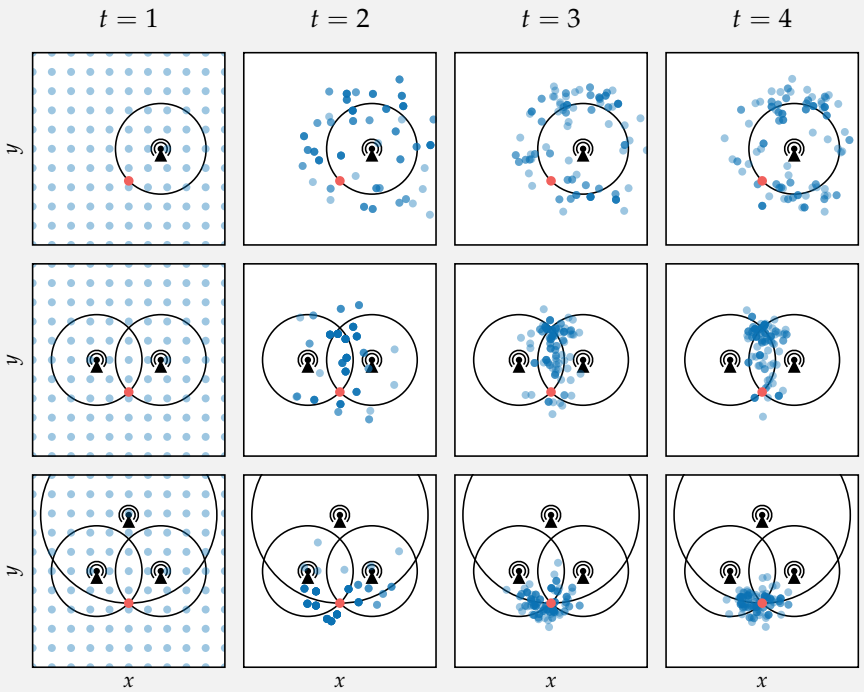
As the number of particles in a particle filter increases, the distribution represented by the particles approaches the true posterior distribution. Unfortunately, particle filters can fail in practice. Low particle coverage and the stochastic nature of the resampling procedure can cause there to be 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 19.5.

Example 19.4. A particle filter applied to different beacon configurations.

Suppose that 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 that we would expect to measure for each particle to the observed ranges.

We assume that 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 here show the evolution of the particle filter. The rows correspond to different numbers of beacons. The red dots indicate 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 identify our location accurately. 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`. Appendix G.5 provides an implementation of `SetCategorical` for defining distributions over discrete sets.

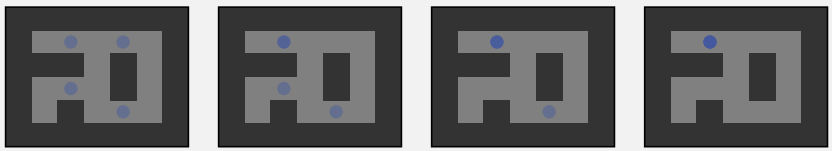
```
struct RejectionParticleFilter
    states # vector of state samples
end

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`.

Example 19.5. 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 when there are fewer particles or when the particles are spread over a large state space.

Spelunker Joe is lost in a grid-based maze. He lost his lantern, so he can observe his surroundings only 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 that 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 figures below show his belief over time, with dots indicating belief particles in his particle filter corresponding to those locations in the maze.



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 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.

## 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.[12] While particle injection can help prevent particle deprivation, it also reduces the accuracy of the posterior belief represented by the particle filter.

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

```
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.

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 weights, 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 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.[13]

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.[14] If $w_{\text{mean}}$ is the current mean particle weight, the two moving averages are updated according to

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

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

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

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

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

The number of injected samples in a given iteration is obtained by comparing the fast and slow mean particle weights:[15]

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

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

[15] 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
```

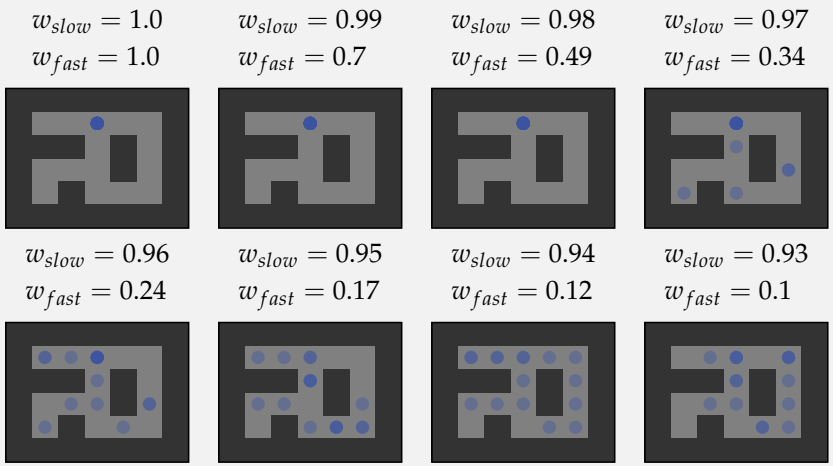
Algorithm 19.9. A particle filter with adaptive injection, which maintains fast and slow exponential moving averages `w_fast` and `w_slow` of the mean particle weight with smoothness factors `α_fast` and `α_slow`, respectively. Particles are injected only if the fast moving average of the mean particle 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`.

## 19.8 Summary

- Partially observable Markov decision processes (POMDPs) extend MDPs to include state uncertainty.
- The uncertainty requires agents in a POMDP to maintain a belief over their state.

Example 19.6. 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 iterations proceed left to right and top to bottom.

Spelunker Joe from example 19.6 now moves one tile to the east and moves all particles in his particle filter one tile east as well. He now senses walls only 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. Here, we see how his filter injects particles from a uniform random distribution, along with the values for the fast and slow filters:




Iterations proceed left to right and top to bottom. Each blue dot represents a particle in the particle filter, corresponding to a partial belief in being in that location of the grid.

- 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 they 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 under the assumption that 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.** What is the belief update for a discrete POMDP with no observation? What is the belief update for a POMDP with linear Gaussian dynamics with no observation?

*Solution:* If an agent in a POMDP without an observation with belief $b$ takes an action $a$, the new belief $b'$ can be calculated as follows:

$$b'(s') = P(s' \mid b, a) = \sum_s P(s' \mid a, b, s) P(s \mid b, a) = \sum_s T(s' \mid s, a) b(s)$$

This belief update is equivalent to having a uniform observation distribution. A POMDP with linear Gaussian dynamics that has no observation will update its belief using only the Kalman filter predict step in equation (19.12).

**Exercise 19.3.** 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:* 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 threshold.

**Exercise 19.4.** 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 $\lambda$:

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

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.[16]

[16] 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 19.5.** 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 19.6.** Explain why Spelunker Joe would not benefit from switching to a particle filter with adaptive injection with $\nu \geq 1$ in example 19.5.

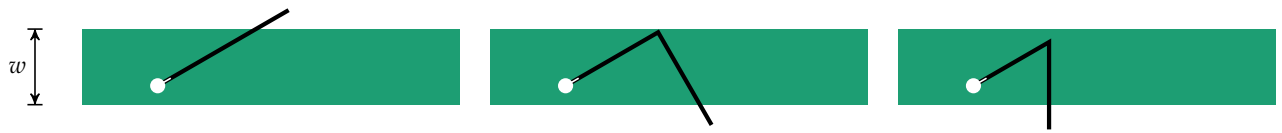
*Solution:* Adaptive injection injects new particles when $\nu w_{\text{fast}}/w_{\text{slow}} < 1$. Spelunker Joe assumes perfect observations and has a belief with particles that match his current observation. Thus, every particle has a weight of 1, and both $w_{\text{fast}}$ and $w_{\text{slow}}$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

**Exercise 19.7.** Why is the injection rate scalar $\nu$ 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 occurs only when the short-term estimate of the mean particle weight $w_{\text{fast}}$ is less than the long-term estimate of the mean particle weight $w_{\text{slow}}$. If $\nu < 1$, then particles can still be generated even if $w_{\text{fast}} \geq w_{\text{slow}}$, despite indicating that current observations have a higher likelihood than the past average.

**Exercise 19.8.** Suppose we are dropped into a rectangular forest at an initial location chosen uniformly at random. We do not know which direction we are facing. Fortunately, we do know the dimensions of the forest (it has width $w$ and length $\ell \gg w$).[17] We can move in a continuous path, continuously observing whether we are still in the forest. How can we apply belief updating to this problem? Here are three possible policies, each defining a different path. Which of these policies are guaranteed to escape the forest? Which policy is best?

[17] This problem was motivated by Richard Bellman's "Lost in a Forest Problem," in which we start at a random location and orientation in a forest with a known geometry and must find a policy that minimizes the average (or maximum) time to exit. R. Bellman, "Minimization Problem," *Bulletin of the American Mathematical Society*, vol. 62, no. 3, p. 270, 1956.




A straight path of length $2w$

Two perpendicular segments, each of length $\sqrt{2}w$

Two legs of an equilateral triangle, each of length $\frac{2\sqrt{3}}{3}w$

*Solution:* Our initial belief is a uniform distribution over all two-dimensional locations and orientations (states) in the forest. We can represent an updated belief using the path that we have traveled thus far. If we are still in the forest, our belief consists of all states that can be reached from a state within the forest by following our path while remaining entirely in the forest. As soon as we exit the forest, our belief consists of all states that reach the edge by following our path while remaining entirely in the forest.

Of the given policies, only the last two are guaranteed to escape the forest. The path formed by the two perpendicular segments and by the two sides of the equilateral triangle will always intersect with the forest's border. The straight segment, however, may not leave the forest. We prefer the shorter of the two escaping policies, which is the equilateral triangle.

**Exercise 19.9.** Algorithm 19.2 checks whether the updated belief is a zero vector. When can a belief update yield a zero vector? Why might this arise in real-world applications?

*Solution:* A zero belief vector can result from an observation $o$ that is considered impossible. This situation can arise after taking action $a$ from belief $b$ when $O(o \mid a, s') = 0$ for all possible next states $s'$ according to $b$ and our transition model. Algorithm 19.2 handles this case by returning a uniform belief. In practical applications, there may be a mismatch between the model and the real world. We generally want to be careful to avoid assigning zero probability to observations, just in case our belief, transition, or observations models are incorrect.

**Exercise 19.10.** 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 that the warnings are independent of the actions, given the status of the plane:

$$T(s^0 \mid s^0, m^0) = 0.95 \qquad O(w^0 \mid s^0) = 0.99$$
$$T(s^0 \mid s^0, m^1) = 1 \qquad O(w^1 \mid s^1) = 0.7$$
$$T(s^1 \mid s^1, m^0) = 1$$
$$T(s^0 \mid s^1, m^1) = 0.98$$

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 $s^0$:

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

We repeat the update for $s^1$:

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

After normalization, we obtain the following updated belief:

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

**Exercise 19.11.** Consider a robot moving along a line with position $x$, velocity $v$, and acceleration $a$. At each time step, we directly control the acceleration and observe the velocity. The equations of motion for the robot are

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

where $\Delta t$ is the duration of each step. Suppose we would like to implement a Kalman filter to update our belief. 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:

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

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 19.12.** Consider a robot with a differential drive moving in two dimensions at a constant speed $v$. The robot's state is its position $(x, y)$ and its heading $\theta$. At each time step, we control the robot's turn rate $\omega$. The equations of motion for the robot are

$$x' = x + v\cos(\theta)\Delta t$$
$$y' = y + v\sin(\theta)\Delta t$$
$$\theta' = \theta + \omega\Delta t$$

This transition function is nonlinear. What is its linearization, $\mathbf{T}_\mathbf{s}$, as a function of the state $\mathbf{s} = [x, y, \theta]$?

*Solution:* The linearization is given by the Jacobian as follows:

$$\mathbf{T}_\mathbf{s} = \begin{bmatrix} \frac{\partial x'}{\partial x} & \frac{\partial x'}{\partial y} & \frac{\partial x'}{\partial \theta} \\ \frac{\partial y'}{\partial x} & \frac{\partial y'}{\partial y} & \frac{\partial y'}{\partial \theta} \\ \frac{\partial \theta'}{\partial x} & \frac{\partial \theta'}{\partial y} & \frac{\partial \theta'}{\partial \theta} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -v\sin(\theta)\Delta t \\ 0 & 1 & v\cos(\theta)\Delta t \\ 0 & 0 & 1 \end{bmatrix}$$

This linearization can be used in an extended Kalman filter to maintain a belief.

**Exercise 19.13.** Suppose we choose the following $2n$ sigma points for an $n$-dimensional distribution:

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

Show that we can reconstruct the mean and the covariance from these sigma points using the weights $w_i = 1/(2n)$.

*Solution:* 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

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

**Exercise 19.14.** Recall the $2n$ sigma points and weights from the previous problem that represent a mean $\boldsymbol{\mu}$ and covariance $\boldsymbol{\Sigma}$. We would like to parameterize the sigma points and weights in order to control the concentration of the points about the mean. Show that we can construct a new set of sigma points by uniformly down-weighting the original sigma points and then including the mean $\boldsymbol{\mu}$ as an additional sigma point. Show that this new set of $2n + 1$ sigma points matches the form in equation (19.23).

*Solution:* We can include the mean $\boldsymbol{\mu}$ in the sigma points from exercise 19.13 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 of the first sigma point. The weights of the remaining sigma points are uniformly reduced from $1/(2n)$ to $(1 - w_1)/(2n)$. 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. This results in a scaled set of sigma points with different higher-order moments, but it preserves the same mean and covariance.

We can match equation (19.23) by substituting $w_1 = \lambda/(n + \lambda)$. It follows that $(1 - w_1)/2n = 1/(2(n + \lambda))$ and $n/(1 - w_1) = n + \lambda$.

**Exercise 19.15.** 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 elementwise 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:

$$\mathbf{s}_1 = \begin{bmatrix}1\\2\end{bmatrix} \qquad w_1 = \frac{2}{2+2} = \frac{1}{2}$$

$$\mathbf{s}_2 = \begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}4\\0\end{bmatrix} = \begin{bmatrix}5\\2\end{bmatrix} \qquad w_2 = \frac{1}{2(2+2)} = \frac{1}{8}$$

$$\mathbf{s}_3 = \begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}4\\0\end{bmatrix} = \begin{bmatrix}-3\\2\end{bmatrix} \qquad w_3 = \frac{1}{2(2+2)} = \frac{1}{8}$$

$$\mathbf{s}_4 = \begin{bmatrix}1\\2\end{bmatrix} + \begin{bmatrix}0\\3\end{bmatrix} = \begin{bmatrix}1\\5\end{bmatrix} \qquad w_4 = \frac{1}{2(2+2)} = \frac{1}{8}$$

$$\mathbf{s}_5 = \begin{bmatrix}1\\2\end{bmatrix} - \begin{bmatrix}0\\3\end{bmatrix} = \begin{bmatrix}1\\-1\end{bmatrix} \qquad w_5 = \frac{1}{2(2+2)} = \frac{1}{8}$$

**Exercise 19.16.** 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:

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

$$\boldsymbol{\mu}' = \frac{1}{2}\begin{bmatrix}2\\2\end{bmatrix} + \frac{1}{8}\begin{bmatrix}10\\10\end{bmatrix} + \frac{1}{8}\begin{bmatrix}-6\\-6\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\5\end{bmatrix} + \frac{1}{8}\begin{bmatrix}2\\-1\end{bmatrix} = \begin{bmatrix}2\\2\end{bmatrix}$$

The covariance matrix can be reconstructed from the weighted sum of point-wise covariance matrices:

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

$$\boldsymbol{\Sigma}' = \frac{1}{2}\begin{bmatrix}0 & 0\\0 & 0\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}64 & 64\\64 & 64\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} + \frac{1}{8}\begin{bmatrix}0 & 0\\0 & 9\end{bmatrix} = \begin{bmatrix}16 & 16\\16 & 18.25\end{bmatrix}$$

**Exercise 19.17.** Both the Kalman filter and the extended Kalman filter compute the cross-covariance matrix $\boldsymbol{\Sigma}_{po}$ using the observation covariance $\mathbf{O}_s$. The unscented Kalman filter does not directly compute this observation matrix, but instead computes $\boldsymbol{\Sigma}_{po}$ directly. Show that the covariance update for the unscented Kalman filter, $\boldsymbol{\Sigma}_{b'} \leftarrow \boldsymbol{\Sigma}_p - \mathbf{K}\boldsymbol{\Sigma}_o\mathbf{K}^\top$, matches the covariance update for the Kalman filter and extended Kalman filter, $\boldsymbol{\Sigma}_{b'} \leftarrow (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\boldsymbol{\Sigma}_p$.

*Solution:* We can use the relations $\mathbf{K} = \mathbf{\Sigma}_{po}\mathbf{\Sigma}_o^{-1}$ and $\mathbf{\Sigma}_{po} = \mathbf{\Sigma}_p\mathbf{O}_s^\top$ to show that the two updates are equivalent. Note also that a symmetric matrix is its own transpose, and that covariance matrices are symmetric.

$$\begin{aligned}
\mathbf{\Sigma}_{b'} &= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{\Sigma}_o\mathbf{K}^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{\Sigma}_o\left(\mathbf{\Sigma}_{po}\mathbf{\Sigma}_o^{-1}\right)^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{\Sigma}_o\left(\mathbf{\Sigma}_o^{-1}\right)^\top\mathbf{\Sigma}_{po}^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{\Sigma}_{po}^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\left(\Sigma_p\mathbf{O}_s^\top\right)^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{O}_s\Sigma_p^\top \\
&= \mathbf{\Sigma}_p - \mathbf{K}\mathbf{O}_s\Sigma_p \\
&= (\mathbf{I} - \mathbf{K}\mathbf{O}_s)\Sigma_p
\end{aligned}$$

**Exercise 19.18.** What are some advantages and disadvantages of using a particle filter instead of a Kalman filter?

*Solution:* A Kalman filter can provide an exact belief update when the system is linear Gaussian. Particle filters can work better when the system is nonlinear and the uncertainty is multimodal. Particle filters are generally more computationally expensive and may suffer from particle deprivation.

**Exercise 19.19.** Consider using a particle filter to maintain a belief in a problem where observations are very reliable, with observations having either high or low likelihood. For example, in the Spelunker Joe problem, we can reliably determine which of the four walls are present, allowing us to immediately discount any states that do not match the observation. Why might a particle filter with rejection be a better match than a traditional particle filter for such problems?

*Solution:* A traditional particle filter produces a set of particles and assigns weights to them according to their observation likelihoods. In problems like the one with Spelunker Joe, many particles may end up with little to no weight. Having many particles with low weight makes the belief vulnerable to particle deprivation. A particle filter with rejection ensures that each particle's successor state is compatible with the observation, thus mitigating the issue of particle deprivation.