CHAPTER 14

PROBABILISTIC REASONING OVER TIME

In which we try to interpret the present, understand the past, and perhaps predict the future, even when very little is crystal clear.

Agents in partially observable environments must be able to keep track of the current state, to the extent that their sensors allow. In Section 4.4 we showed a methodology for doing that: an agent maintains a **belief state** that represents which states of the world are currently possible. From the belief state and a **transition model**, the agent can predict how the world might evolve in the next time step. From the percepts observed and a **sensor model**, the agent can update the belief state. This is a pervasive idea: in Chapter 4 belief states were represented by explicitly enumerated sets of states, whereas in Chapters 7 and 11 they were represented by logical formulas. Those approaches defined belief states in terms of which world states were *possible*, but could say nothing about which states were *likely* or *unlikely*. In this chapter, we use probability theory to quantify the degree of belief in elements of the belief state.

As we show in Section 14.1, time itself is handled in the same way as in Chapter 7: a changing world is modeled using a variable for each aspect of the world state *at each point in time*. The transition and sensor models may be uncertain: the transition model describes the probability distribution of the variables at time t, given the state of the world at past times, while the sensor model describes the probability of each percept at time t, given the current state of the world. Section 14.2 defines the basic inference tasks and describes the general structure of inference algorithms for temporal models. Then we describe three specific kinds of models: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include hidden Markov models and Kalman filters as special cases).

14.1 Time and Uncertainty

We have developed our techniques for probabilistic reasoning in the context of *static* worlds, in which each random variable has a single fixed value. For example, when repairing a car, we assume that whatever is broken remains broken during the process of diagnosis; our job is to infer the state of the car from observed evidence, which also remains fixed.

Now consider a slightly different problem: treating a diabetic patient. As in the case of car repair, we have evidence such as recent insulin doses, food intake, blood sugar measurements, and other physical signs. The task is to assess the current state of the patient, including the actual blood sugar level and insulin level. Given this information, we can make a decision about the patient's food intake and insulin dose. Unlike the case of car repair, here the *dynamic* aspects of the problem are essential. Blood sugar levels and measurements thereof can change rapidly over time, depending on recent food intake and insulin doses, metabolic activity, the time of day, and so on. To assess the current state from the history of evidence and to predict the outcomes of treatment actions, we must model these changes.

The same considerations arise in many other contexts, such as tracking the location of a robot, tracking the economic activity of a nation, and making sense of a spoken or written sequence of words. How can dynamic situations like these be modeled?

14.1.1 States and observations

Discrete time

Time slice

This chapter discusses **discrete-time** models, in which the world is viewed as a series of snapshots or **time slices**.[1] We'll just number the time slices 0, 1, 2, and so on, rather than assigning specific times to them. Typically, the time interval Δ between slices is assumed to be the same for every interval. For any particular application, a specific value of Δ has to be chosen. Sometimes this is dictated by the sensor; for example, a video camera might supply images at intervals of 1/30 of a second. In other cases, the interval is dictated by the typical rates of change of the relevant variables; for example, in the case of blood glucose monitoring, things can change significantly in the course of ten minutes, so a one-minute interval might be appropriate. On the other hand, in modeling continental drift over geological time, an interval of a million years might be fine.

Each time slice in a discrete-time probability model contains a set of random variables, some observable and some not. For simplicity, we will assume that the same subset of variables is observable in each time slice (although this is not strictly necessary in anything that follows). We will use $\mathbf{X}_t$ to denote the set of state variables at time t, which are assumed to be unobservable, and $\mathbf{E}_t$ to denote the set of observable evidence variables. The observation at time t is $\mathbf{E}_t = \mathbf{e}_t$ for some set of values $\mathbf{e}_t$.

Consider the following example: You are the security guard stationed at a secret underground installation. You want to know whether it's raining today, but your only access to the outside world occurs each morning when you see the director coming in with, or without, an umbrella. For each day t, the set $\mathbf{E}_t$ thus contains a single evidence variable $Umbrella_t$ or U_t for short (whether the umbrella appears), and the set $\mathbf{X}_t$ contains a single state variable $Rain_t$ or R_t for short (whether it is raining). Other problems can involve larger sets of variables. In the diabetes example, the evidence variables might be $MeasuredBloodSugar_t$ and $PulseRate_t$ while the state variables might include $BloodSugar_t$ and $StomachContents_t$. (Notice that $BloodSugar_t$ and $MeasuredBloodSugar_t$ are not the same variable; this is how we deal with noisy measurements of actual quantities.)

We will assume that the state sequence starts at $t=0$ and evidence starts arriving at $t=1$. Hence, our umbrella world is represented by state variables $R_0, R_1, R_2, \ldots$ and evidence variables $U_1, U_2, \ldots$. We will use the notation $a{:}b$ to denote the sequence of integers from a to b inclusive and the notation $\mathbf{X}_{a:b}$ to denote the set of variables from $\mathbf{X}_a$ to $\mathbf{X}_b$ inclusive. For example, $U_{1:3}$ corresponds to U_1, U_2, U_3. (Note that this is different from the notation used in programming languages such as Python and Go, where `U[1:3]` would *not* include `U[3]`.)

[1] Uncertainty over *continuous* time can be modeled by **stochastic differential equations** (SDEs). The models studied in this chapter can be viewed as discrete-time approximations to SDEs.

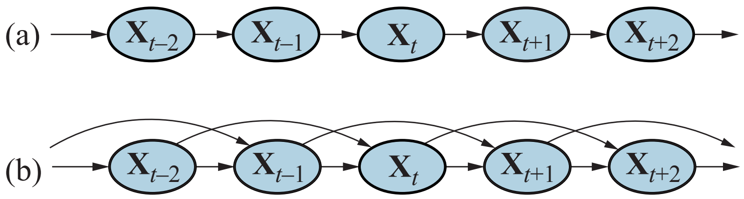


Figure 14.1 (a) Bayesian network structure corresponding to a first-order Markov process with state defined by the variables $\mathbf{X}_t$. (b) A second-order Markov process.

14.1.2 Transition and sensor models

With the set of state and evidence variables for a given problem decided on, the next step is to specify how the world evolves (the transition model) and how the evidence variables get their values (the sensor model).

The transition model specifies the probability distribution over the latest state variables, given the previous values, that is, $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1})$. Now we face a problem: the set $\mathbf{X}_{0:t-1}$ is unbounded in size as t increases. We solve the problem by making a **Markov assumption**—that the current state depends on only a *finite fixed number* of previous states. Processes satisfying this assumption were first studied in depth by the statistician Andrei Markov (1856–1922) and are called **Markov processes** or **Markov chains**. They come in various flavors; the simplest is the **first-order Markov process**, in which the current state depends only on the previous state and not on any earlier states. In other words, a state provides enough information to make the future conditionally independent of the past, and we have

Markov assumption

Markov process

First-order Markov process

$$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1}). \tag{14.1}$$

Hence, in a first-order Markov process, the transition model is the conditional distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$. The transition model for a second-order Markov process is the conditional distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-2}, \mathbf{X}_{t-1})$. Figure 14.1 shows the Bayesian network structures corresponding to first-order and second-order Markov processes.

Even with the Markov assumption there is still a problem: there are infinitely many possible values of t. Do we need to specify a different distribution for each time step? We avoid this problem by assuming that changes in the world state are caused by a **time-homogeneous** process—that is, a process of change that is governed by laws that do not themselves change over time. In the umbrella world, then, the conditional probability of rain, $\mathbf{P}(R_t \mid R_{t-1})$, is the same for all t, and we need specify only one conditional probability table.

Time-homogeneous

Now for the sensor model. The evidence variables $\mathbf{E}_t$ *could* depend on previous variables as well as the current state variables, but any state that's worth its salt should suffice to generate the current sensor values. Thus, we make a **sensor Markov assumption** as follows:

Sensor Markov assumption

$$\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_{0:t}, \mathbf{E}_{1:t-1}) = \mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t). \tag{14.2}$$

Thus, $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$ is our sensor model (sometimes called the **observation model**). Figure 14.2 shows both the transition model and the sensor model for the umbrella example. Notice the direction of the dependence between state and sensors: the arrows go from the actual state of the world to sensor values because the state of the world *causes* the sensors to take on particular values: the rain *causes* the umbrella to appear. (The inference process, of course,

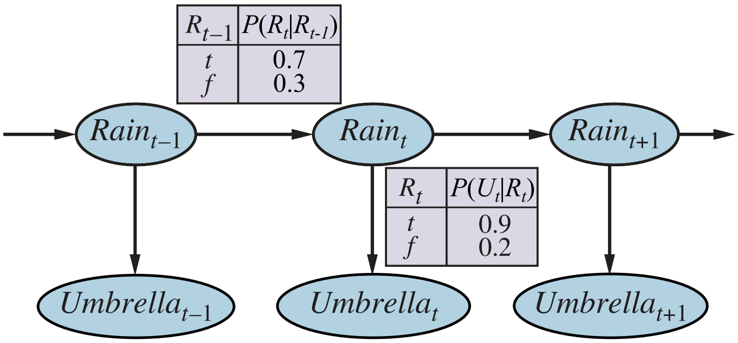


Figure 14.2 Bayesian network structure and conditional distributions describing the umbrella world. The transition model is $\mathbf{P}(Rain_t \mid Rain_{t-1})$ and the sensor model is $\mathbf{P}(Umbrella_t \mid Rain_t)$.

goes in the other direction; the distinction between the direction of modeled dependencies and the direction of inference is one of the principal advantages of Bayesian networks.)

In addition to specifying the transition and sensor models, we need to say how everything gets started—the prior probability distribution at time 0, $\mathbf{P}(\mathbf{X}_0)$. With that, we have a specification of the complete joint distribution over all the variables, using Equation (13.2). For any time step t,

$$\mathbf{P}(\mathbf{X}_{0:t}, \mathbf{E}_{1:t}) = \mathbf{P}(\mathbf{X}_0) \prod_{i=1}^{t} \mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1})\, \mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i)\,. \tag{14.3}$$

The three terms on the right-hand side are the initial state model $\mathbf{P}(\mathbf{X}_0)$, the transition model $\mathbf{P}(\mathbf{X}_i \mid \mathbf{X}_{i-1})$, and the sensor model $\mathbf{P}(\mathbf{E}_i \mid \mathbf{X}_i)$. This equation defines the semantics of the family of temporal models represented by the three terms. Notice that standard Bayesian networks cannot represent such models because they require a finite set of variables. The ability to handle an infinite set of variables comes from two things: first, defining the infinite set using integer indices; and second, the use of implicit universal quantification (see Section 8.2) to define the sensor and transition models for every time step.

The structure in Figure 14.2 is a first-order Markov process—the probability of rain is assumed to depend only on whether it rained the previous day. Whether such an assumption is reasonable depends on the domain itself. The first-order Markov assumption says that the state variables contain *all* the information needed to characterize the probability distribution for the next time slice. Sometimes the assumption is exactly true—for example, if a particle is executing a random walk along the x-axis, changing its position by ±1 at each time step, then using the x-coordinate as the state gives a first-order Markov process. Sometimes the assumption is only approximate, as in the case of predicting rain only on the basis of whether it rained the previous day. There are two ways to improve the accuracy of the approximation:

1. Increasing the order of the Markov process model. For example, we could make a second-order model by adding $Rain_{t-2}$ as a parent of $Rain_t$, which might give slightly more accurate predictions. For example, in Palo Alto, California, it very rarely rains more than two days in a row.

2. Increasing the set of state variables. For example, we could add $Season_t$ to allow us to incorporate historical records of rainy seasons, or we could add $Temperature_t$, $Humidity_t$, and $Pressure_t$ (perhaps at a range of locations) to allow us to use a physical model of rainy conditions.

Exercise 14.AUGM asks you to show that the first solution—increasing the order—can always be reformulated as an increase in the set of state variables, keeping the order fixed. Notice that adding state variables might improve the system's predictive power but also increases the prediction *requirements*: we now have to predict the new variables as well. Thus, we are looking for a "self-sufficient" set of variables, which really means that we have to understand the "physics" of the process being modeled. The requirement for accurate modeling of the process is obviously lessened if we can add new sensors (e.g., measurements of temperature and pressure) that provide information directly about the new state variables.

Consider, for example, the problem of tracking a robot wandering randomly on the X–Y plane. One might propose that the position and velocity are a sufficient set of state variables: one can simply use Newton's laws to calculate the new position, and the velocity may change unpredictably. If the robot is battery-powered, however, then battery exhaustion would tend to have a systematic effect on the change in velocity. Because this in turn depends on how much power was used by all previous maneuvers, the Markov property is violated.

We can restore the Markov property by including the charge level $Battery_t$ as one of the state variables that make up $\mathbf{X}_t$. This helps in predicting the motion of the robot, but in turn requires a model for predicting $Battery_t$ from $Battery_{t-1}$ and the velocity. In some cases, that can be done reliably, but more often we find that error accumulates over time. In that case, accuracy can be improved by *adding a new sensor* for the battery level. We will return to the battery example in Section 14.5.

14.2 Inference in Temporal Models

Having set up the structure of a generic temporal model, we can formulate the basic inference tasks that must be solved:

- **Filtering**[2] or **state estimation** is the task of computing the **belief state** $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$—the posterior distribution over the most recent state given all evidence to date. In the umbrella example, this would mean computing the probability of rain today, given all the umbrella observations made so far. Filtering is what a rational agent does to keep track of the current state so that rational decisions can be made. It turns out that an almost identical calculation provides the likelihood of the evidence sequence, $P(\mathbf{e}_{1:t})$.

Filtering

State estimation

Belief state

- **Prediction**: This is the task of computing the posterior distribution over the *future* state, given all evidence to date. That is, we wish to compute $\mathbf{P}(\mathbf{X}_{t+k} \mid \mathbf{e}_{1:t})$ for some $k > 0$. In the umbrella example, this might mean computing the probability of rain three days from now, given all the observations to date. Prediction is useful for evaluating possible courses of action based on their expected outcomes.

Prediction

- **Smoothing**: This is the task of computing the posterior distribution over a *past* state, given all evidence up to the present. That is, we wish to compute $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ for some k

Smoothing

[2] The term "filtering" refers to the roots of this problem in early work on signal processing, where the problem is to filter out the noise in a signal by estimating its underlying properties.

such that $0 \leq k < t$. In the umbrella example, it might mean computing the probability that it rained last Wednesday, given all the observations of the umbrella carrier made up to today. Smoothing provides a better estimate of the state at time k than was available at that time, because it incorporates more evidence.[3]

- **Most likely explanation**: Given a sequence of observations, we might wish to find the sequence of states that is most likely to have generated those observations. That is, we wish to compute $\operatorname{argmax}_{\mathbf{x}_{1:t}} P(\mathbf{x}_{1:t} \,|\, \mathbf{e}_{1:t})$. For example, if the umbrella appears on each of the first three days and is absent on the fourth, then the most likely explanation is that it rained on the first three days and did not rain on the fourth. Algorithms for this task are useful in many applications, including speech recognition—where the aim is to find the most likely sequence of words, given a series of sounds—and the reconstruction of bit strings transmitted over a noisy channel.

In addition to these inference tasks, we also have

- **Learning**: The transition and sensor models, if not yet known, can be learned from observations. Just as with static Bayesian networks, dynamic Bayes net learning can be done as a by-product of inference. Inference provides an estimate of what transitions actually occurred and of what states generated the sensor readings, and these estimates can be used to learn the models. The learning process can operate via an iterative update algorithm called expectation–maximization or EM, or it can result from Bayesian updating of the model parameters given the evidence. See Chapter 21 for more details.

The remainder of this section describes generic algorithms for the four inference tasks, independent of the particular kind of model employed. Improvements specific to each model are described in subsequent sections.

14.2.1 Filtering and prediction

As we pointed out in Section 7.7.3, a useful filtering algorithm needs to maintain a current state estimate and update it, rather than going back over the entire history of percepts for each update. (Otherwise, the cost of each update increases as time goes by.) In other words, given the result of filtering up to time t, the agent needs to compute the result for $t+1$ from the new evidence $\mathbf{e}_{t+1}$. So we have

$$\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t+1}) = f(\mathbf{e}_{t+1}, \mathbf{P}(\mathbf{X}_t \,|\, \mathbf{e}_{1:t}))$$

for some function f. This process is called **recursive estimation**. (See also Sections 4.4 and 7.7.3.) We can view the calculation as being composed of two parts: first, the current state distribution is projected forward from t to $t+1$; then it is updated using the new evidence $\mathbf{e}_{t+1}$. This two-part process emerges quite simply when the formula is rearranged:

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t+1}) &= \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t}, \mathbf{e}_{t+1}) \quad \text{(dividing up the evidence)} \\
&= \alpha\, \mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1}, \mathbf{e}_{1:t})\, \mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t}) \quad \text{(using Bayes' rule, given } \mathbf{e}_{1:t}\text{)} \\
&= \alpha\, \underbrace{\mathbf{P}(\mathbf{e}_{t+1} \,|\, \mathbf{X}_{t+1})}_{\text{update}}\, \underbrace{\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t})}_{\text{prediction}} \quad \text{(by the sensor Markov assumption).} \qquad (14.4)
\end{aligned}$$

Here and throughout this chapter, α is a normalizing constant used to make probabilities sum up to 1. Now we plug in an expression for the one-step prediction $\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{e}_{1:t})$, obtained by

[3] In particular, when tracking a moving object with inaccurate position observations, smoothing gives a smoother estimated trajectory than filtering—hence the name.

conditioning on the current state $\mathbf{X}_t$. The resulting equation for the new state estimate is the central result in this chapter:

$$\begin{aligned}\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) &= \alpha\, \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1}) \sum_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t, \mathbf{e}_{1:t}) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\ &= \alpha\, \underbrace{\mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1})}_{\text{sensor model}} \sum_{\mathbf{x}_t} \underbrace{\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)}_{\text{transition model}}\ \underbrace{P(\mathbf{x}_t \mid \mathbf{e}_{1:t})}_{\text{recursion}} \quad \text{(Markov assumption).} \end{aligned} \tag{14.5}$$

In this expression, all the terms come either from the model or from the previous state estimate. Hence, we have the desired recursive formulation. We can think of the filtered estimate $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ as a "message" $\mathbf{f}_{1:t}$ that is propagated forward along the sequence, modified by each transition and updated by each new observation. The process is given by

$$\mathbf{f}_{1:t+1} = \text{FORWARD}(\mathbf{f}_{1:t}, \mathbf{e}_{t+1}),$$

where FORWARD implements the update described in Equation (14.5) and the process begins with $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0)$. When all the state variables are discrete, the time for each update is constant (i.e., independent of t), and the space required is also constant. (The constants depend, of course, on the size of the state space and the specific type of the temporal model in question.) *The time and space requirements for updating must be constant if a finite agent is to keep track of the current state distribution indefinitely.*

Let us illustrate the filtering process for two steps in the basic umbrella example (Figure 14.2). That is, we will compute $\mathbf{P}(R_2 \mid u_{1:2})$ as follows:

- On day 0, we have no observations, only the security guard's prior beliefs; let's assume that consists of $\mathbf{P}(R_0) = \langle 0.5, 0.5 \rangle$.
- On day 1, the umbrella appears, so $U_1 = true$. The prediction from $t=0$ to $t=1$ is

  $$\begin{aligned}\mathbf{P}(R_1) &= \sum_{r_0} \mathbf{P}(R_1 \mid r_0) P(r_0) \\ &= \langle 0.7, 0.3 \rangle \times 0.5 + \langle 0.3, 0.7 \rangle \times 0.5 = \langle 0.5, 0.5 \rangle .\end{aligned}$$

  Then the update step simply multiplies by the probability of the evidence for $t=1$ and normalizes, as shown in Equation (14.4):

  $$\begin{aligned}\mathbf{P}(R_1 \mid u_1) &= \alpha\, \mathbf{P}(u_1 \mid R_1) \mathbf{P}(R_1) = \alpha \langle 0.9, 0.2 \rangle \langle 0.5, 0.5 \rangle \\ &= \alpha \langle 0.45, 0.1 \rangle \approx \langle 0.818, 0.182 \rangle .\end{aligned}$$

- On day 2, the umbrella appears, so $U_2 = true$. The prediction from $t=1$ to $t=2$ is

  $$\begin{aligned}\mathbf{P}(R_2 \mid u_1) &= \sum_{r_1} \mathbf{P}(R_2 \mid r_1) P(r_1 \mid u_1) \\ &= \langle 0.7, 0.3 \rangle \times 0.818 + \langle 0.3, 0.7 \rangle \times 0.182 \approx \langle 0.627, 0.373 \rangle ,\end{aligned}$$

  and updating it with the evidence for $t=2$ gives

  $$\begin{aligned}\mathbf{P}(R_2 \mid u_1, u_2) &= \alpha\, \mathbf{P}(u_2 \mid R_2) \mathbf{P}(R_2 \mid u_1) = \alpha \langle 0.9, 0.2 \rangle \langle 0.627, 0.373 \rangle \\ &= \alpha \langle 0.565, 0.075 \rangle \approx \langle 0.883, 0.117 \rangle .\end{aligned}$$

Intuitively, the probability of rain increases from day 1 to day 2 because rain persists. Exercise 14.CONV(a) asks you to investigate this tendency further.

The task of **prediction** can be seen simply as filtering without the addition of new evidence. In fact, the filtering process already incorporates a one-step prediction, and it is easy to derive the following recursive computation for predicting the state at $t+k+1$ from a prediction for $t+k$:

$$\mathbf{P}(\mathbf{X}_{t+k+1} \mid \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_{t+k}} \underbrace{\mathbf{P}(\mathbf{X}_{t+k+1} \mid \mathbf{x}_{t+k})}_{\text{transition model}} \underbrace{P(\mathbf{x}_{t+k} \mid \mathbf{e}_{1:t})}_{\text{recursion}} . \tag{14.6}$$

Naturally, this computation involves only the transition model and not the sensor model.

It is interesting to consider what happens as we try to predict further and further into the future. As Exercise 14.CONV(b) shows, the predicted distribution for rain converges to a fixed point $\langle 0.5, 0.5 \rangle$, after which it remains constant for all time.[4] This is the **stationary distribution** of the Markov process defined by the transition model. (See also page 462.) A great deal is known about the properties of such distributions and about the **mixing time**—roughly, the time taken to reach the fixed point. In practical terms, this dooms to failure any attempt to predict the *actual* state for a number of steps that is more than a small fraction of the mixing time, unless the stationary distribution itself is strongly peaked in a small area of the state space. The more uncertainty there is in the transition model, the shorter will be the mixing time and the more the future is obscured.

Mixing time

In addition to filtering and prediction, we can use a forward recursion to compute the **likelihood** of the evidence sequence, $P(\mathbf{e}_{1:t})$. This is a useful quantity if we want to compare different temporal models that might have produced the same evidence sequence (e.g., two different models for the persistence of rain). For this recursion, we use a likelihood message $\ell_{1:t}(\mathbf{X}_t) = \mathbf{P}(\mathbf{X}_t, \mathbf{e}_{1:t})$. It is easy to show (Exercise 14.LIKL) that the message calculation is identical to that for filtering:

$$\ell_{1:t+1} = \text{FORWARD}(\ell_{1:t}, \mathbf{e}_{t+1}) .$$

Having computed $\ell_{1:t}$, we obtain the actual likelihood by summing out $\mathbf{X}_t$:

$$L_{1:t} = P(\mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} \ell_{1:t}(\mathbf{x}_t) . \tag{14.7}$$

Notice that the likelihood message represents the probabilities of longer and longer evidence sequences as time goes by and so becomes numerically smaller and smaller, leading to underflow problems with floating-point arithmetic. This is an important problem in practice, but we shall not go into solutions here.

14.2.2 Smoothing

As we said earlier, smoothing is the process of computing the distribution over past states given evidence up to the present—that is, $\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t})$ for $0 \leq k < t$. (See Figure 14.3.) In anticipation of another recursive message-passing approach, we can split the computation into two parts—the evidence up to k and the evidence from $k+1$ to t,

$$\begin{aligned}
\mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:t}) &= \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}, \mathbf{e}_{k+1:t}) \\
&= \alpha \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k, \mathbf{e}_{1:k}) \quad \text{(using Bayes' rule, given } \mathbf{e}_{1:k}\text{)} \\
&= \alpha \mathbf{P}(\mathbf{X}_k \mid \mathbf{e}_{1:k}) \mathbf{P}(\mathbf{e}_{k+1:t} \mid \mathbf{X}_k) \quad \text{(using conditional independence)} \\
&= \alpha \mathbf{f}_{1:k} \times \mathbf{b}_{k+1:t} . \qquad (14.8)
\end{aligned}$$

where "×" represents pointwise multiplication of vectors. Here we have defined a "back-

[4] If one picks an arbitrary day to be $t = 0$, then it makes sense to choose the prior $\mathbf{P}(Rain_0)$ to match the stationary distribution, which is why we picked $\langle 0.5, 0.5 \rangle$ as the prior. Had we picked a different prior, the stationary distribution would still have worked out to $\langle 0.5, 0.5 \rangle$.

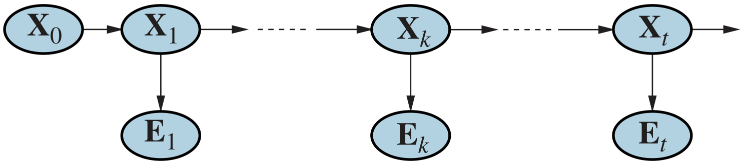


Figure 14.3 Smoothing computes $\mathbf{P}(\mathbf{X}_k \,|\, \mathbf{e}_{1:t})$, the posterior distribution of the state at some past time k given a complete sequence of observations from 1 to t.

ward" message $\mathbf{b}_{k+1:t} = \mathbf{P}(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_k)$, analogous to the forward message $\mathbf{f}_{1:k}$. The forward message $\mathbf{f}_{1:k}$ can be computed by filtering forward from 1 to k, as given by Equation (14.5). It turns out that the backward message $\mathbf{b}_{k+1:t}$ can be computed by a recursive process that runs *backward* from t:

$$\begin{aligned}
\mathbf{P}(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_k) &= \sum_{\mathbf{x}_{k+1}} \mathbf{P}(\mathbf{e}_{k+1:t} \,|\, \mathbf{X}_k, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \,|\, \mathbf{X}_k) \quad \text{(conditioning on } \mathbf{X}_{k+1}\text{)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1:t} \,|\, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \,|\, \mathbf{X}_k) \quad \text{(by conditional independence)} \\
&= \sum_{\mathbf{x}_{k+1}} P(\mathbf{e}_{k+1}, \mathbf{e}_{k+2:t} \,|\, \mathbf{x}_{k+1}) \mathbf{P}(\mathbf{x}_{k+1} \,|\, \mathbf{X}_k) \\
&= \sum_{\mathbf{x}_{k+1}} \underbrace{P(\mathbf{e}_{k+1} \,|\, \mathbf{x}_{k+1})}_{\text{sensor model}} \underbrace{P(\mathbf{e}_{k+2:t} \,|\, \mathbf{x}_{k+1})}_{\text{recursion}} \underbrace{\mathbf{P}(\mathbf{x}_{k+1} \,|\, \mathbf{X}_k)}_{\text{transition model}} ,
\end{aligned} \tag{14.9}$$

where the last step follows by the conditional independence of $\mathbf{e}_{k+1}$ and $\mathbf{e}_{k+2:t}$, given $\mathbf{x}_{k+1}$. In this expression, all the terms come either from the model or from the previous backward message. Hence, we have the desired recursive formulation. In message form, we have

$$\mathbf{b}_{k+1:t} = \text{BACKWARD}(\mathbf{b}_{k+2:t}, \mathbf{e}_{k+1}) ,$$

where BACKWARD implements the update described in Equation (14.9). As with the forward recursion, the time and space needed for each update are constant and thus independent of t.

We can now see that the two terms in Equation (14.8) can both be computed by recursions through time, one running forward from 1 to k and using the filtering equation (14.5) and the other running backward from t to $k+1$ and using Equation (14.9).

For the initialization of the backward phase, we have $\mathbf{b}_{t+1:t} = \mathbf{P}(\mathbf{e}_{t+1:t} \,|\, \mathbf{X}_t) = \mathbf{P}(\ \,|\, \mathbf{X}_t) = \mathbf{1}$, where $\mathbf{1}$ is a vector of 1s. The reason for this is that $\mathbf{e}_{t+1:t}$ is an empty sequence, so the probability of observing it is 1.

Let us now apply this algorithm to the umbrella example, computing the smoothed estimate for the probability of rain at time $k=1$, given the umbrella observations on days 1 and 2. From Equation (14.8), this is given by

$$\mathbf{P}(R_1 \,|\, u_1, u_2) = \alpha \, \mathbf{P}(R_1 \,|\, u_1) \mathbf{P}(u_2 \,|\, R_1) . \tag{14.10}$$

The first term we already know to be $\langle .818, .182 \rangle$, from the forward filtering process described earlier. The second term can be computed by applying the backward recursion in Equation (14.9):

$$\begin{aligned}
\mathbf{P}(u_2 \,|\, R_1) &= \sum_{r_2} P(u_2 \,|\, r_2) P(\ \,|\, r_2) \mathbf{P}(r_2 \,|\, R_1) \\
&= (0.9 \times 1 \times \langle 0.7, 0.3 \rangle) + (0.2 \times 1 \times \langle 0.3, 0.7 \rangle) = \langle 0.69, 0.41 \rangle .
\end{aligned}$$

```
function FORWARD-BACKWARD(ev, prior) returns a vector of probability distributions
  inputs: ev, a vector of evidence values for steps 1, ..., t
          prior, the prior distribution on the initial state, P(X_0)
  local variables: fv, a vector of forward messages for steps 0, ..., t
                   b, a representation of the backward message, initially all 1s
                   sv, a vector of smoothed estimates for steps 1, ..., t

  fv[0] ← prior
  for i = 1 to t do
      fv[i] ← FORWARD(fv[i − 1], ev[i])
  for i = t down to 1 do
      sv[i] ← NORMALIZE(fv[i] × b)
      b ← BACKWARD(b, ev[i])
  return sv
```

Figure 14.4 The forward–backward algorithm for smoothing: computing posterior probabilities of a sequence of states given a sequence of observations. The FORWARD and BACKWARD operators are defined by Equations (14.5) and (14.9), respectively.

Plugging this into Equation (14.10), we find that the smoothed estimate for rain on day 1 is

$$\mathbf{P}(R_1 \mid u_1, u_2) = \alpha \langle 0.818, 0.182 \rangle \times \langle 0.69, 0.41 \rangle \approx \langle 0.883, 0.117 \rangle .$$

Thus, the smoothed estimate for rain on day 1 is *higher* than the filtered estimate (0.818) in this case. This is because the umbrella on day 2 makes it more likely to have rained on day 2; in turn, because rain tends to persist, that makes it more likely to have rained on day 1.

Both the forward and backward recursions take a constant amount of time per step; hence, the time complexity of smoothing with respect to evidence $\mathbf{e}_{1:t}$ is $O(t)$. This is the complexity for smoothing at a particular time step k. If we want to smooth the whole sequence, one obvious method is simply to run the whole smoothing process once for each time step to be smoothed. This results in a time complexity of $O(t^2)$.

A better approach uses a simple application of dynamic programming to reduce the complexity to $O(t)$. A clue appears in the preceding analysis of the umbrella example, where we were able to reuse the results of the forward-filtering phase. The key to the linear-time algorithm is to *record the results* of forward filtering over the whole sequence. Then we run the backward recursion from t down to 1, computing the smoothed estimate at each step k from the computed backward message $\mathbf{b}_{k+1:t}$ and the stored forward message $\mathbf{f}_{1:k}$. The algorithm, aptly called the **forward–backward algorithm**, is shown in Figure 14.4.

Forward–backward algorithm

The alert reader will have spotted that the Bayesian network structure shown in Figure 14.3 is a *polytree* as defined on page 451. This means that a straightforward application of the clustering algorithm also yields a linear-time algorithm that computes smoothed estimates for the entire sequence. It is now understood that the forward–backward algorithm is in fact a special case of the polytree propagation algorithm used with clustering methods (although the two were developed independently).

The forward–backward algorithm forms the computational backbone for many applications that deal with sequences of noisy observations. As described so far, it has two practical drawbacks. The first is that its space complexity can be too high when the state space is large and the sequences are long. It uses $O(|\mathbf{f}|t)$ space where $|\mathbf{f}|$ is the size of the representation of the forward message. The space requirement can be reduced to $O(|\mathbf{f}|\log t)$ with a concomitant increase in the time complexity by a factor of $\log t$, as shown in Exercise 14.ISLE. In some cases (see Section 14.3), a constant-space algorithm can be used.

Fixed-lag smoothing

The second drawback of the basic algorithm is that it needs to be modified to work in an *online* setting where smoothed estimates must be computed for earlier time slices as new observations are continuously added to the end of the sequence. The most common requirement is for **fixed-lag smoothing**, which requires computing the smoothed estimate $\mathbf{P}(\mathbf{X}_{t-d} \,|\, \mathbf{e}_{1:t})$ for fixed d. That is, smoothing is done for the time slice d steps behind the current time t; as t increases, the smoothing has to keep up. Obviously, we can run the forward–backward algorithm over the d-step "window" as each new observation is added, but this seems inefficient. In Section 14.3, we will see that fixed-lag smoothing can, in some cases, be done in constant time per update, independent of the lag d.

14.2.3 Finding the most likely sequence

Suppose that $[\mathit{true}, \mathit{true}, \mathit{false}, \mathit{true}, \mathit{true}]$ is the observed umbrella sequence for the security guard's first five days on the job. What weather sequence is most likely to explain this? Does the absence of the umbrella on day 3 mean that it wasn't raining, or did the director forget to bring it? If it didn't rain on day 3, perhaps (because weather tends to persist) it didn't rain on day 4 either, but the director brought the umbrella just in case. In all, there are 2^5 possible weather sequences we could pick. Is there a way to find the most likely one, short of enumerating all of them and calculating their likelihoods?

We could try this linear-time procedure: use smoothing to find the posterior distribution for the weather at each time step; then construct the sequence, using at each step the weather that is most likely according to the posterior. Such an approach should set off alarm bells in the reader's head, because the posterior distributions computed by smoothing are distributions over *single* time steps, whereas to find the most likely *sequence* we must consider *joint* probabilities over all the time steps. The results can in fact be quite different. (See Exercise 14.VITE.)

There *is* a linear-time algorithm for finding the most likely sequence, but it requires more thought. It relies on the same Markov property that yielded efficient algorithms for filtering and smoothing. The idea is to view each sequence as a *path* through a graph whose nodes are the possible *states* at each time step. Such a graph is shown for the umbrella world in Figure 14.5(a). Now consider the task of finding the most likely path through this graph, where the likelihood of any path is the product of the transition probabilities along the path and the probabilities of the given observations at each state.

Let's focus in particular on paths that reach the state $\mathit{Rain}_5 = \mathit{true}$. Because of the Markov property, it follows that the most likely path to the state $\mathit{Rain}_5 = \mathit{true}$ consists of the most likely path to *some* state at time 4 followed by a transition to $\mathit{Rain}_5 = \mathit{true}$; and the state at time 4 that will become part of the path to $\mathit{Rain}_5 = \mathit{true}$ is whichever maximizes the likelihood of that path. In other words, *there is a recursive relationship between most likely paths to each state* $\mathbf{x}_{t+1}$ *and most likely paths to each state* $\mathbf{x}_t$.

We can use this property directly to construct a recursive algorithm for computing the most likely path given the evidence. We will use a recursively computed message $m_{1:t}$, like

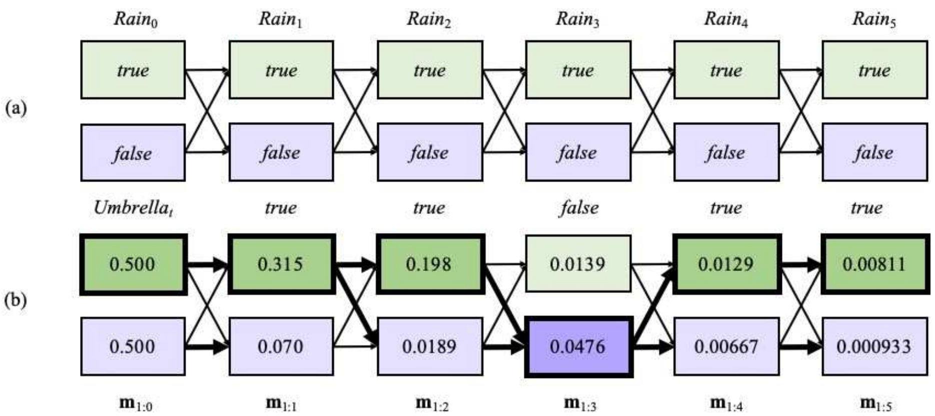


Figure 14.5 (a) Possible state sequences for $Rain_t$ can be viewed as paths through a graph of the possible states at each time step. (States are shown as rectangles to avoid confusion with nodes in a Bayes net.) (b) Operation of the Viterbi algorithm for the umbrella observation sequence $[true, true, false, true, true]$, where the evidence starts at time 1. For each t, we have shown the values of the message $\mathbf{m}_{1:t}$, which gives the probability of the best sequence reaching each state at time t. Also, for each state, the bold arrow leading into it indicates its best predecessor as measured by the product of the preceding sequence probability and the transition probability. Following the bold arrows back from the most likely state in $\mathbf{m}_{1:5}$ gives the most likely sequence, shown by the bold outlines and darker shading.

the forward message $\mathbf{f}_{1:t}$ in the filtering algorithm. The message is defined as follows:[5]

$$\mathbf{m}_{1:t} = \max_{\mathbf{x}_{1:t-1}} \mathbf{P}(\mathbf{x}_{1:t-1}, \mathbf{X}_t, \mathbf{e}_{1:t}) .$$

To obtain the recursive relationship between $\mathbf{m}_{1:t+1}$ and $\mathbf{m}_{1:t}$, we can repeat more or less the same steps that we used for Equation (14.5):

$$\begin{aligned}
\mathbf{m}_{1:t+1} &= \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t}, \mathbf{X}_{t+1}, \mathbf{e}_{1:t+1}) = \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{x}_{1:t}, \mathbf{X}_{t+1}, \mathbf{e}_{1:t}, e_{t+1}) \\
&= \max_{\mathbf{x}_{1:t}} \mathbf{P}(e_{t+1} \,|\, \mathbf{x}_{1:t}, \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \mathbf{P}(\mathbf{x}_{1:t}, \mathbf{X}_{t+1}, \mathbf{e}_{1:t}) \\
&= \mathbf{P}(e_{t+1} \,|\, \mathbf{X}_{t+1}) \max_{\mathbf{x}_{1:t}} \mathbf{P}(\mathbf{X}_{t+1}, \,|\, \mathbf{x}_t) P(\mathbf{x}_{1:t}, \mathbf{e}_{1:t}) \\
&= \mathbf{P}(e_{t+1} \,|\, \mathbf{X}_{t+1}) \max_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1}, \,|\, \mathbf{x}_t) \max_{\mathbf{x}_{1:t-1}} P(\mathbf{x}_{1:t-1}, \mathbf{x}_t, \mathbf{e}_{1:t}) \qquad (14.11)
\end{aligned}$$

where the final term $\max_{\mathbf{x}_{1:t-1}} P(\mathbf{x}_{1:t-1}, \mathbf{x}_t, \mathbf{e}_{1:t})$ is exactly the entry for the particular state $\mathbf{x}_t$ in the message vector $\mathbf{m}_{1:t}$. Equation (14.11) is essentially identical to the filtering equation (14.5) except that the summation over $\mathbf{x}_t$ in Equation (14.5) is replaced by the maximization over $\mathbf{x}_t$ in Equation (14.11), and there is no normalization constant α in Equation (14.11). Thus, the algorithm for computing the most likely sequence is similar to filtering: it starts at time 0 with the prior $\mathbf{m}_{1:0} = \mathbf{P}(\mathbf{X}_0)$ and then runs forward along the sequence, computing the

[5] Notice that these are not quite the probabilities of the most likely paths to reach the states $\mathbf{X}_t$ given the evidence, which would be the conditional probabilities $\max_{\mathbf{x}_{1:t-1}} \mathbf{P}(\mathbf{x}_{1:t-1}, \mathbf{X}_t \,|\, \mathbf{e}_{1:t})$; but the two vectors are related by a constant factor $P(\mathbf{e}_{1:t})$. The difference is immaterial because the max operator doesn't care about constant factors. We get a slightly simpler recursion with $\mathbf{m}_{1:t}$ defined this way.

m message at each time step using Equation (14.11). The progress of this computation is shown in Figure 14.5(b).

At the end of the observation sequence, $\mathbf{m}_{1:t}$ will contain the probability for the most likely sequence reaching *each* of the final states. One can thus easily select the final state of the most likely sequence overall (the state outlined in bold at step 5). In order to identify the actual sequence, as opposed to just computing its probability, the algorithm will also need to record, for each state, the best state that leads to it; these are indicated by the bold arrows in Figure 14.5(b). The optimal sequence is identified by following these bold arrows backwards from the best final state.

Viterbi algorithm

The algorithm we have just described is called the **Viterbi algorithm**, after its inventor, Andrew Viterbi. Like the filtering algorithm, its time complexity is linear in t, the length of the sequence. Unlike filtering, which uses constant space, its space requirement is also linear in t. This is because the Viterbi algorithm needs to keep the pointers that identify the best sequence leading to each state.

One final practical point: numerical underflow is a significant issue for the Viterbi algorithm. In Figure 14.5(b), the probabilities are getting smaller and smaller—and this is just a toy example. Real applications in DNA analysis or message decoding may have thousands or millions of steps. One possible solution is simply to normalize **m** at each step; this rescaling does not affect correctness because $\max(cx, cy) = c \cdot \max(x, y)$. A second solution is to use log probabilities everywhere and replace multiplication by addition. Again, correctness is unaffected because the log function is monotonic, so $\max(\log x, \log y) = \log \max(x, y)$.

14.3 Hidden Markov Models

The preceding section developed algorithms for temporal probabilistic reasoning using a general framework that was independent of the specific form of the transition and sensor models and independent of the nature of the state and evidence variables. In this and the next two sections, we discuss more concrete models and applications that illustrate the power of the basic algorithms and in some cases allow further improvements.

Hidden Markov model

We begin with the **hidden Markov model**, or **HMM**. An HMM is a temporal probabilistic model in which the state of the process is described by a *single*, *discrete* random variable. The possible values of the variable are the possible states of the world. The umbrella example described in the preceding section is therefore an HMM, since it has just one state variable: $Rain_t$. What happens if you have a model with two or more state variables? You can still fit it into the HMM framework by combining the variables into a single "megavariable" whose values are all possible tuples of values of the individual state variables. We will see that the restricted structure of HMMs allows for a simple and elegant matrix implementation of all the basic algorithms.[6]

Although HMMs require the *state* to be a single, discrete variable, there is no corresponding restriction on the *evidence* variables. This is because the evidence variables are always observed, which means that there is no need to keep track of any distribution over their values. (If a variable is not observed, it can simply be dropped from the model for that time step.) There can be many evidence variables, both discrete and continuous.

[6] The reader unfamiliar with basic operations on vectors and matrices might wish to consult Appendix A before proceeding with this section.

14.3.1 Simplified matrix algorithms

With a single, discrete state variable X_t, we can give concrete form to the representations of the transition model, the sensor model, and the forward and backward messages. Let the state variable X_t have values denoted by integers $1,\ldots,S$, where S is the number of possible states. The transition model $\mathbf{P}(X_t \mid X_{t-1})$ becomes an $S \times S$ matrix $\mathbf{T}$, where

$$\mathbf{T}_{ij} = P(X_t = j \mid X_{t-1} = i)\,.$$

That is, $\mathbf{T}_{ij}$ is the probability of a transition from state i to state j. For example, if we number the states $Rain = true$ and $Rain = false$ as 1 and 2, respectively, then the transition matrix for the umbrella world defined in Figure 14.2 is

$$\mathbf{T} = \mathbf{P}(X_t \mid X_{t-1}) = \begin{pmatrix} 0.7 & 0.3 \\ 0.3 & 0.7 \end{pmatrix}.$$

We also put the sensor model in matrix form. In this case, because the value of the evidence variable E_t is known at time t (call it e_t), we need only specify, for each state, how likely it is that the state causes e_t to appear: we need $P(e_t \mid X_t = i)$ for each state i. For mathematical convenience we place these values into an $S \times S$ diagonal **observation matrix**, $\mathbf{O}_t$, one for each time step. The ith diagonal entry of $\mathbf{O}_t$ is $P(e_t \mid X_t = i)$ and the other entries are 0. For example, on day 1 in the umbrella world of Figure 14.5, $U_1 = true$, and on day 3, $U_3 = false$, so we have

Observation matrix

$$\mathbf{O}_1 = \begin{pmatrix} 0.9 & 0 \\ 0 & 0.2 \end{pmatrix}; \qquad \mathbf{O}_3 = \begin{pmatrix} 0.1 & 0 \\ 0 & 0.8 \end{pmatrix}.$$

Now, if we use column vectors to represent the forward and backward messages, all the computations become simple matrix–vector operations. The forward equation (14.5) becomes

$$\mathbf{f}_{1:t+1} = \alpha\, \mathbf{O}_{t+1} \mathbf{T}^{\top} \mathbf{f}_{1:t} \tag{14.12}$$

and the backward equation (14.9) becomes

$$\mathbf{b}_{k+1:t} = \mathbf{T}\mathbf{O}_{k+1} \mathbf{b}_{k+2:t}\,. \tag{14.13}$$

From these equations, we can see that the time complexity of the forward–backward algorithm (Figure 14.4) applied to a sequence of length t is $O(S^2 t)$, because each step requires multiplying an S-element vector by an $S \times S$ matrix. The space requirement is $O(St)$, because the forward pass stores t vectors of size S.

Besides providing an elegant description of the filtering and smoothing algorithms for HMMs, the matrix formulation reveals opportunities for improved algorithms. The first is a simple variation on the forward–backward algorithm that allows smoothing to be carried out in *constant* space, independently of the length of the sequence. The idea is that smoothing for any particular time slice k requires the simultaneous presence of both the forward and backward messages, $\mathbf{f}_{1:k}$ and $\mathbf{b}_{k+1:t}$, according to Equation (14.8). The forward–backward algorithm achieves this by storing the **f**s computed on the forward pass so that they are available during the backward pass. Another way to achieve this is with a single pass that propagates both **f** and **b** in the same direction. For example, the "forward" message **f** can be propagated backward if we manipulate Equation (14.12) to work in the other direction:

$$\mathbf{f}_{1:t} = \alpha' (\mathbf{T}^{\top})^{-1} \mathbf{O}_{t+1}^{-1} \mathbf{f}_{1:t+1}\,.$$

The modified smoothing algorithm works by first running the standard forward pass to compute $\mathbf{f}_{t:t}$ (forgetting all the intermediate results) and then running the backward pass for both

```
function FIXED-LAG-SMOOTHING($e_t, hmm, d$) returns a distribution over $\mathbf{X}_{t-d}$
  inputs: $e_t$, the current evidence for time step $t$
          $hmm$, a hidden Markov model with $S \times S$ transition matrix $\mathbf{T}$
          $d$, the length of the lag for smoothing
  persistent: $t$, the current time, initially 1
              $\mathbf{f}$, the forward message $\mathbf{P}(X_t \mid e_{1:t})$, initially $hmm$.PRIOR
              $\mathbf{B}$, the $d$-step backward transformation matrix, initially the identity matrix
              $e_{t-d:t}$, double-ended list of evidence from $t-d$ to $t$, initially empty
  local variables: $\mathbf{O}_{t-d}, \mathbf{O}_t$, diagonal matrices containing the sensor model information

  add $e_t$ to the end of $e_{t-d:t}$
  $\mathbf{O}_t \leftarrow$ diagonal matrix containing $\mathbf{P}(e_t \mid X_t)$
  if $t > d$ then
     $\mathbf{f} \leftarrow$ FORWARD($\mathbf{f}, e_{t-d}$)
     remove $e_{t-d-1}$ from the beginning of $e_{t-d:t}$
     $\mathbf{O}_{t-d} \leftarrow$ diagonal matrix containing $\mathbf{P}(e_{t-d} \mid X_{t-d})$
     $\mathbf{B} \leftarrow \mathbf{O}_{t-d}^{-1}\mathbf{T}^{-1}\mathbf{B}\mathbf{T}\mathbf{O}_t$
  else $\mathbf{B} \leftarrow \mathbf{B}\mathbf{T}\mathbf{O}_t$
  $t \leftarrow t+1$
  if $t > d+1$ then return NORMALIZE($\mathbf{f} \times \mathbf{B1}$) else return null
```

Figure 14.6 An algorithm for smoothing with a fixed time lag of d steps, implemented as an online algorithm that outputs the new smoothed estimate given the observation for a new time step. Notice that the final output NORMALIZE($\mathbf{f} \times \mathbf{B1}$) is just $\alpha\,\mathbf{f} \times \mathbf{b}$, by Equation (14.14).

b and **f** together, using them to compute the smoothed estimate at each step. Since only one copy of each message is needed, the storage requirements are constant (i.e., independent of t, the length of the sequence). There are two significant restrictions on this algorithm: it requires that the transition matrix be invertible and that the sensor model have no zeroes—that is, that every observation be possible in every state.

A second area in which the matrix formulation reveals an improvement is in *online* smoothing with a fixed lag. The fact that smoothing can be done in constant space suggests that there should exist an efficient recursive algorithm for online smoothing—that is, an algorithm whose time complexity is independent of the length of the lag. Let us suppose that the lag is d; that is, we are smoothing at time slice $t-d$, where the current time is t. By Equation (14.8), we need to compute

$$\alpha\,\mathbf{f}_{1:t-d} \times \mathbf{b}_{t-d+1:t}$$

for slice $t-d$. Then, when a new observation arrives, we need to compute

$$\alpha\,\mathbf{f}_{1:t-d+1} \times \mathbf{b}_{t-d+2:t+1}$$

for slice $t-d+1$. How can this be done incrementally? First, we can compute $\mathbf{f}_{1:t-d+1}$ from $\mathbf{f}_{1:t-d}$, using the standard filtering process, Equation (14.5).

Computing the backward message incrementally is trickier, because there is no simple relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the new backward message $\mathbf{b}_{t-d+2:t+1}$. Instead, we will examine the relationship between the old backward message $\mathbf{b}_{t-d+1:t}$ and the backward message at the front of the sequence, $\mathbf{b}_{t+1:t}$. To do this, we apply

Equation (14.13) d times to get

$$\mathbf{b}_{t-d+1:t} = \left(\prod_{i=t-d+1}^{t} \mathbf{T}\mathbf{O}_i \right) \mathbf{b}_{t+1:t} = \mathbf{B}_{t-d+1:t}\mathbf{1}\,, \tag{14.14}$$

where the matrix $\mathbf{B}_{t-d+1:t}$ is the product of the sequence of $\mathbf{T}$ and $\mathbf{O}$ matrices and $\mathbf{1}$ is a vector of 1s. $\mathbf{B}$ can be thought of as a "transformation operator" that transforms a later backward message into an earlier one. A similar equation holds for the new backward messages *after* the next observation arrives:

$$\mathbf{b}_{t-d+2:t+1} = \left(\prod_{i=t-d+2}^{t+1} \mathbf{T}\mathbf{O}_i \right) \mathbf{b}_{t+2:t+1} = \mathbf{B}_{t-d+2:t+1}\mathbf{1}\,. \tag{14.15}$$

Examining the product expressions in Equations (14.14) and (14.15), we see that they have a simple relationship: to get the second product, "divide" the first product by the first element $\mathbf{T}\mathbf{O}_{t-d+1}$, and multiply by the new last element $\mathbf{T}\mathbf{O}_{t+1}$. In matrix language, then, there is a simple relationship between the old and new $\mathbf{B}$ matrices:

$$\mathbf{B}_{t-d+2:t+1} = \mathbf{O}_{t-d+1}^{-1}\mathbf{T}^{-1}\mathbf{B}_{t-d+1:t}\mathbf{T}\mathbf{O}_{t+1}\,. \tag{14.16}$$

This equation provides an incremental update for the $\mathbf{B}$ matrix, which in turn (through Equation (14.15)) allows us to compute the new backward message $\mathbf{b}_{t-d+2:t+1}$. The complete algorithm, which requires storing and updating $\mathbf{f}$ and $\mathbf{B}$, is shown in Figure 14.6.

14.3.2 Hidden Markov model example: Localization

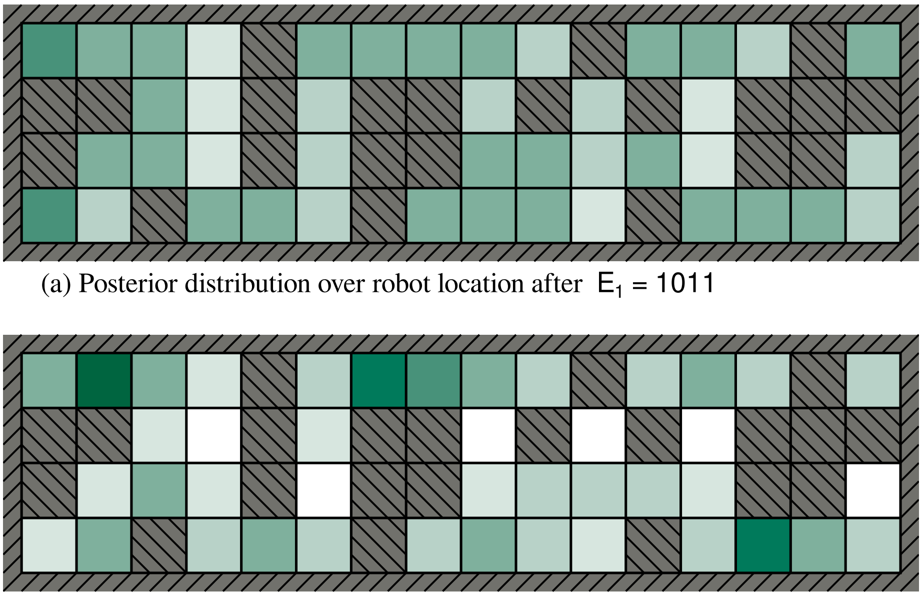
On page 151, we introduced a simple form of the **localization** problem for the vacuum world. In that version, the robot had a single nondeterministic *Move* action and its sensors reported perfectly whether or not obstacles lay immediately to the north, south, east, and west; the robot's belief state was the set of possible locations it could be in.

Here we make the problem slightly more realistic by allowing for noise in the sensors, and formalizing the idea that the robot moves randomly—it is equally likely to move to any adjacent empty square. The state variable X_t represents the location of the robot on the discrete grid; the domain of this variable is the set of empty squares, which we will label by the integers$\{1,\ldots,S\}$. Let $\textsc{Neighbors}(i)$ be the set of empty squares that are adjacent to i and let $N(i)$ be the size of that set. Then the transition model for the *Move* action says that the robot is equally likely to end up at any neighboring square:

$$P(X_{t+1}=j \,|\, X_t=i) = \mathbf{T}_{ij} = \begin{cases} 1/N(i) \text{ if } j \in \textsc{Neighbors}(i) \\ 0 \text{ otherwise.} \end{cases}$$

We don't know where the robot starts, so we will assume a uniform distribution over all the squares; that is, $P(X_0=i)=1/S$. For the particular environment we consider (Figure 14.7), $S=42$ and the transition matrix $\mathbf{T}$ has $42 \times 42 = 1764$ entries.

The sensor variable E_t has 16 possible values, each a four-bit sequence giving the presence or absence of an obstacle in each of the compass directions NESW. For example, 1010 means that the north and south sensors report an obstacle and the east and west do not. Suppose that each sensor's error rate is ϵ and that errors occur independently for the four sensor directions. In that case, the probability of getting all four bits right is $(1-\epsilon)^4$ and the probability of getting them all wrong is ϵ^4. Furthermore, if d_{it} is the discrepancy—the number of



(a) Posterior distribution over robot location after $E_1 = 1011$

(b) Posterior distribution over robot location after $E_1 = 1011$, $E_2 = 1010$

Figure 14.7 Posterior distribution over robot location: (a) after one observation $E_1 = 1011$ (i.e., obstacles to the north, south, and west); (b) after a random move to an adjacent location and a second observation $E_2 = 1010$ (i.e., obstacles to the north and south). The darkness of each square corresponds to the probability that the robot is at that location. The sensor error rate for each bit is $\epsilon = 0.2$.

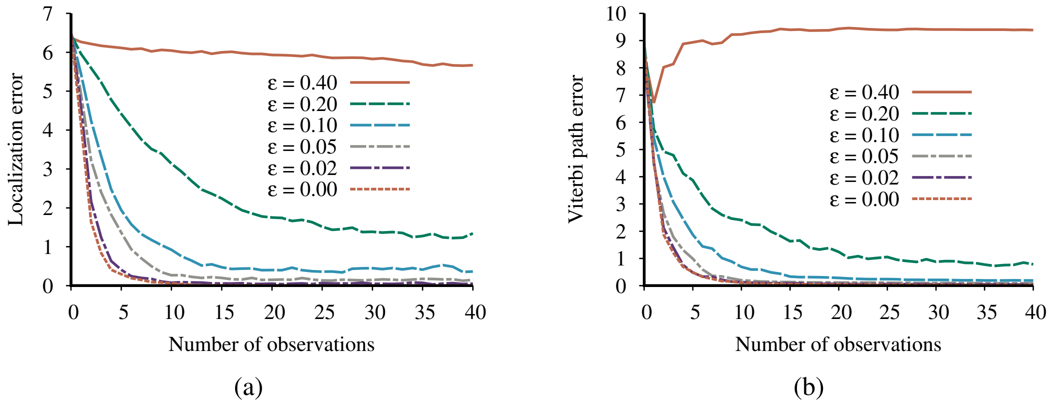
bits that are different—between the true values for square i and the actual reading e_t, then the probability that a robot in square i would receive a sensor reading e_t is

$$P(E_t = e_t \mid X_t = i) = (\mathbf{O}_t)_{ii} = (1-\epsilon)^{4-d_{it}}\epsilon^{d_{it}} .$$

For example, the probability that a square with obstacles to the north and south would produce a sensor reading 1110 is $(1-\epsilon)^3\epsilon^1$.

Given the matrices $\mathbf{T}$ and $\mathbf{O}_t$, the robot can use Equation (14.12) to compute the posterior distribution over locations—that is, to work out where it is. Figure 14.7 shows the distributions $\mathbf{P}(X_1 \mid E_1 = 1011)$ and $\mathbf{P}(X_2 \mid E_1 = 1011, E_2 = 1010)$. This is the same maze we saw before in Figure 4.18 (page 152), but there we used logical filtering to find the locations that were *possible*, assuming perfect sensing. Those same locations are still the most *likely* with noisy sensing, but now *every* location has some nonzero probability because any location could produce any sensor values.

In addition to filtering to estimate its current location, the robot can use smoothing (Equation (14.13)) to work out where it was at any given past time—for example, where it began at time 0—and it can use the Viterbi algorithm to work out the most likely path it has taken to get where it is now. Figure 14.8 shows the localization error and Viterbi path error for various values of the per-bit sensor error rate ϵ. Even when ϵ is 0.20—which means that the overall sensor reading is wrong 59% of the time—the robot is usually able to work out its location to within two squares after 20 observations. This is because of the algorithm's ability to integrate evidence over time and to take into account the probabilistic constraints imposed



(a) (b)

Figure 14.8 Performance of HMM localization as a function of the length of the observation sequence for various different values of the sensor error probability ϵ; data averaged over 400 runs. (a) The localization error, defined as the Manhattan distance from the true location. (b) The Viterbi path error, defined as the average Manhattan distance of states on the Viterbi path from corresponding states on the true path.

on the location sequence by the transition model. When ϵ is 0.10 or less, the robot needs only a few observations to work out where it is and to track its position accurately. When ϵ is 0.40, both the localization error and the Viterbi path error remain large; in other words, the robot is lost. This is because a sensor with an error probability of 0.40 provides too little information to counteract the loss of information about the robot's position that comes from the unpredictable random motion.

The state variable for the example we have considered in this section is a physical location in the world. Other problems can, of course, include other aspects of the world. Exercise 14.ROOM asks you to consider a version of the vacuum robot that has the policy of going straight for as long as it can; only when it encounters an obstacle does it change to a new heading. To model this robot, each state in the model consists of a *(location, heading)* pair. For the environment in Figure 14.7, which has 42 empty squares, this leads to 168 states and a transition matrix with $168^2 = 28,224$ entries—still a manageable number.

If we add the possibility of dirt in each of the 42 squares, the number of states is multiplied by 2^{42} and the transition matrix has more than 10^{29} entries—no longer a manageable number. In general, if the state is composed of n discrete variables with at most d values each, the corresponding HMM transition matrix will have size $O(d^{2n})$ and the per-update computation time will also be $O(d^{2n})$.

For these reasons, although HMMs have many uses in areas ranging from speech recognition to molecular biology, they are fundamentally limited in their ability to represent complex processes. In the terminology introduced in Chapter 2, HMMs are an atomic representation: states of the world have no internal structure and are simply labeled by integers. Section 14.5 shows how to use dynamic Bayesian networks—a factored representation—to model domains with many state variables. The next section shows how to handle domains with continuous state variables, which of course lead to an infinite state space.

14.4 Kalman Filters

Imagine watching a small bird flying through dense jungle foliage at dusk: you glimpse brief, intermittent flashes of motion; you try hard to guess where the bird is and where it will appear next so that you don't lose it. Or imagine that you are a World War II radar operator peering at a faint, wandering blip that appears once every 10 seconds on the screen. Or, going back further still, imagine you are Kepler trying to reconstruct the motions of the planets from a collection of highly inaccurate angular observations taken at irregular and imprecisely measured intervals.

In all these cases, you are doing filtering: estimating state variables (here, the position and velocity of a moving object) from noisy observations over time. If the variables were discrete, we could model the system with a hidden Markov model. This section examines methods for handling continuous variables, using an algorithm called **Kalman filtering**, after one of its inventors, Rudolf Kalman.

Kalman filtering

The bird's flight might be specified by six continuous variables at each time point; three for position (X_t, Y_t, Z_t) and three for velocity $(\dot{X}_t, \dot{Y}_t, \dot{Z}_t)$. We will need suitable conditional densities to represent the transition and sensor models; as in Chapter 13, we will use **linear–Gaussian** distributions. This means that the next state $\mathbf{X}_{t+1}$ must be a linear function of the current state $\mathbf{X}_t$, plus some Gaussian noise, a condition that turns out to be quite reasonable in practice. Consider, for example, the X-coordinate of the bird, ignoring the other coordinates for now. Let the time interval between observations be Δ, and assume constant velocity during the interval; then the position update is given by $X_{t+\Delta} = X_t + \dot{X}\,\Delta$. Adding Gaussian noise (to account for wind variation, etc.), we obtain a linear–Gaussian transition model:

$$P(X_{t+\Delta} = x_{t+\Delta} \mid X_t = x_t, \dot{X}_t = \dot{x}_t) = \mathcal{N}(x_{t+\Delta}; x_t + \dot{x}_t\,\Delta, \sigma^2)\,.$$

The Bayesian network structure for a system with position vector $\mathbf{X}_t$ and velocity $\dot{\mathbf{X}}_t$ is shown in Figure 14.9. Note that this is a very specific form of linear–Gaussian model; the general form will be described later in this section and covers a vast array of applications beyond the simple motion examples of the first paragraph. The reader might wish to consult Appendix A for some of the mathematical properties of Gaussian distributions; for our immediate purposes, the most important is that a **multivariate Gaussian** distribution for d variables is specified by a d-element mean μ and a $d \times d$ covariance matrix Σ.

14.4.1 Updating Gaussian distributions

In Chapter 13 on page 441, we alluded to a key property of the linear–Gaussian family of distributions: it remains closed under Bayesian updating. (That is, given any evidence, the posterior is still in the linear–Gaussian family.) Here we make this claim precise in the context of filtering in a temporal probability model. The required properties correspond to the two-step filtering calculation in Equation (14.5):

1. If the current distribution $\mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ is Gaussian and the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$ is linear–Gaussian, then the one-step predicted distribution given by

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) = \int_{\mathbf{x}_t} \mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t})\, d\mathbf{x}_t \qquad (14.17)$$

is also a Gaussian distribution.

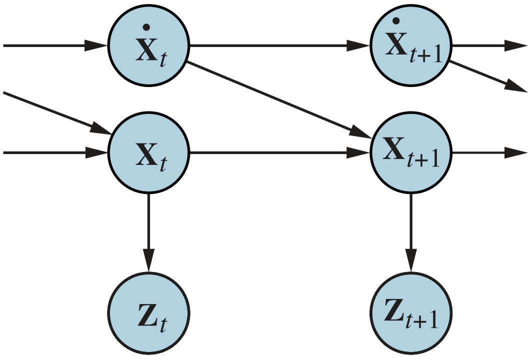


Figure 14.9 Bayesian network structure for a linear dynamical system with position $\mathbf{X}_t$, velocity $\dot{\mathbf{X}}_t$, and position measurement $\mathbf{Z}_t$.

2. If the prediction $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t})$ is Gaussian and the sensor model $\mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1})$ is linear–Gaussian, then, after conditioning on the new evidence, the updated distribution

$$\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t+1}) = \alpha\, \mathbf{P}(\mathbf{e}_{t+1} \mid \mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{e}_{1:t}) \qquad (14.18)$$

is also a Gaussian distribution.

Thus, the FORWARD operator for Kalman filtering takes a Gaussian forward message $\mathbf{f}_{1:t}$, specified by a mean μ_t and covariance Σ_t, and produces a new multivariate Gaussian forward message $\mathbf{f}_{1:t+1}$, specified by a mean μ_{t+1} and covariance Σ_{t+1}. So if we start with a Gaussian prior $\mathbf{f}_{1:0} = \mathbf{P}(\mathbf{X}_0) = \mathcal{N}(\mu_0, \Sigma_0)$, filtering with a linear–Gaussian model produces a Gaussian state distribution for all time.

This seems to be a nice, elegant result, but why is it so important? The reason is that except for a few special cases such as this, *filtering with continuous or hybrid (discrete and continuous) networks generates state distributions whose representation grows without bound over time.* This statement is not easy to prove in general, but Exercise 14.KFSW shows what happens for a simple example.

14.4.2 A simple one-dimensional example

We have said that the FORWARD operator for the Kalman filter maps a Gaussian into a new Gaussian. This translates into computing a new mean and covariance from the previous mean and covariance. Deriving the update rule in the general (multivariate) case requires rather a lot of linear algebra, so we will stick to a very simple univariate case for now, and later give the results for the general case. Even for the univariate case, the calculations are somewhat tedious, but we feel that they are worth seeing because the usefulness of the Kalman filter is tied so intimately to the mathematical properties of Gaussian distributions.

The temporal model we consider describes a **random walk** of a single continuous state variable X_t with a noisy observation Z_t. An example might be the "consumer confidence" index, which can be modeled as undergoing a random Gaussian-distributed change each month and is measured by a random consumer survey that also introduces Gaussian sampling noise. The prior distribution is assumed to be Gaussian with variance σ_0^2:

$$P(x_0) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)} .$$

(For simplicity, we use the same symbol α for all normalizing constants in this section.) The transition model adds a Gaussian perturbation of constant variance σ_x^2 to the current state:

$$P(x_{t+1}\,|\,x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_{t+1}-x_t)^2}{\sigma_x^2}\right)}.$$

The sensor model assumes Gaussian noise with variance σ_z^2:

$$P(z_t\,|\,x_t) = \alpha\, e^{-\frac{1}{2}\left(\frac{(z_t-x_t)^2}{\sigma_z^2}\right)}.$$

Now, given the prior $P(X_0)$, the one-step predicted distribution comes from Equation (14.17):

$$\begin{aligned} P(x_1) &= \int_{-\infty}^{\infty} P(x_1\,|\,x_0)P(x_0)\,dx_0 = \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{(x_1-x_0)^2}{\sigma_x^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_0-\mu_0)^2}{\sigma_0^2}\right)}\,dx_0 \\ &= \alpha \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(\frac{\sigma_0^2(x_1-x_0)^2+\sigma_x^2(x_0-\mu_0)^2}{\sigma_0^2\sigma_x^2}\right)}\,dx_0\,. \end{aligned}$$

This integral looks rather complicated. The key to progress is to notice that the exponent is the sum of two expressions that are *quadratic* in x_0 and hence is itself a quadratic in x_0. A simple trick known as **completing the square** allows the rewriting of any quadratic $ax_0^2+bx_0+c$ as the sum of a squared term $a(x_0-\frac{-b}{2a})^2$ and a residual term $c-\frac{b^2}{4a}$ that is independent of x_0. In this case, we have $a=(\sigma_0^2+\sigma_x^2)/(\sigma_0^2\sigma_x^2)$, $b=-2(\sigma_0^2x_1+\sigma_x^2\mu_0)/(\sigma_0^2\sigma_x^2)$, and $c=(\sigma_0^2x_1^2+\sigma_x^2\mu_0^2)/(\sigma_0^2\sigma_x^2)$. The residual term can be taken outside the integral, giving us

Completing the square

$$P(x_1) = \alpha\, e^{-\frac{1}{2}\left(c-\frac{b^2}{4a}\right)} \int_{-\infty}^{\infty} e^{-\frac{1}{2}\left(a(x_0-\frac{-b}{2a})^2\right)}\,dx_0\,.$$

Now the integral is just the integral of a Gaussian over its full range, which is simply 1. Thus, we are left with only the residual term from the quadratic. Plugging back in the expressions for a, b, and c and simplifying, we obtain

$$P(x_1) = \alpha\, e^{-\frac{1}{2}\left(\frac{(x_1-\mu_0)^2}{\sigma_0^2+\sigma_x^2}\right)}.$$

That is, the one-step predicted distribution is a Gaussian with the same mean μ_0 and a variance equal to the sum of the original variance σ_0^2 and the transition variance σ_x^2.

To complete the update step, we need to condition on the observation at the first time step, namely, z_1. From Equation (14.18), this is given by

$$\begin{aligned} P(x_1\,|\,z_1) &= \alpha P(z_1\,|\,x_1)P(x_1) \\ &= \alpha\, e^{-\frac{1}{2}\left(\frac{(z_1-x_1)^2}{\sigma_z^2}\right)} e^{-\frac{1}{2}\left(\frac{(x_1-\mu_0)^2}{\sigma_0^2+\sigma_x^2}\right)}. \end{aligned}$$

Once again, we combine the exponents and complete the square (Exercise 14.KALM), obtaining the following expression for the posterior:

$$P(x_1\,|\,z_1) = \alpha\, e^{-\frac{1}{2}\frac{\left(x_1-\frac{(\sigma_0^2+\sigma_x^2)z_1+\sigma_z^2\mu_0}{\sigma_0^2+\sigma_x^2+\sigma_z^2}\right)^2}{(\sigma_0^2+\sigma_x^2)\sigma_z^2/(\sigma_0^2+\sigma_x^2+\sigma_z^2)}}. \qquad (14.19)$$

Thus, after one update cycle, we have a new Gaussian distribution for the state variable.

From the Gaussian formula in Equation (14.19), we see that the new mean and standard deviation can be calculated from the old mean and standard deviation as follows:

$$\mu_{t+1} = \frac{(\sigma_t^2+\sigma_x^2)z_{t+1}+\sigma_z^2\mu_t}{\sigma_t^2+\sigma_x^2+\sigma_z^2} \qquad \text{and} \qquad \sigma_{t+1}^2 = \frac{(\sigma_t^2+\sigma_x^2)\sigma_z^2}{\sigma_t^2+\sigma_x^2+\sigma_z^2}. \qquad (14.20)$$

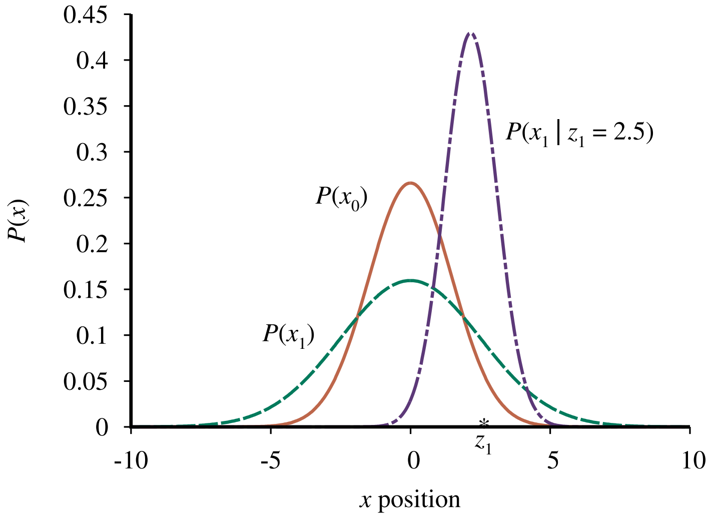


Figure 14.10 Stages in the Kalman filter update cycle for a random walk with a prior given by $\mu_0 = 0.0$ and $\sigma_0 = 1.5$, transition noise given by $\sigma_x = 2.0$, sensor noise given by $\sigma_z = 1.0$, and a first observation $z_1 = 2.5$ (marked on the x-axis). Notice how the prediction $P(x_1)$ is flattened out, relative to $P(x_0)$, by the transition noise. Notice also that the mean of the posterior distribution $P(x_1 \mid z_1)$ is slightly to the left of the observation z_1 because the mean is a weighted average of the prediction and the observation.

Figure 14.10 shows one update cycle of the Kalman filter in the one-dimensional case for particular values of the transition and sensor models.

Equation (14.20) plays exactly the same role as the general filtering equation (14.5) or the HMM filtering equation (14.12). Because of the special nature of Gaussian distributions, however, the equations have some interesting additional properties.

First, we can interpret the calculation for the new mean μ_{t+1} as a *weighted mean* of the new observation z_{t+1} and the old mean μ_t. If the observation is unreliable, then σ_z^2 is large and we pay more attention to the old mean; if the old mean is unreliable (σ_t^2 is large) or the process is highly unpredictable (σ_x^2 is large), then we pay more attention to the observation.

Second, notice that the update for the variance σ_{t+1}^2 is *independent of the observation*. We can therefore compute in advance what the sequence of variance values will be. Third, the sequence of variance values converges quickly to a fixed value that depends only on σ_x^2 and σ_z^2, thereby substantially simplifying the subsequent calculations. (See Exercise 14.VARI.)

14.4.3 The general case

The preceding derivation illustrates the key property of Gaussian distributions that allows Kalman filtering to work: the fact that the exponent is a quadratic form. This is true not just for the univariate case; the full multivariate Gaussian distribution has the form

$$\mathcal{N}(\mathbf{x};\mu,\Sigma) = \alpha\, e^{-\frac{1}{2}\left((\mathbf{x}-\mu)^{\top}\Sigma^{-1}(\mathbf{x}-\mu)\right)}.$$

Multiplying out the terms in the exponent, we see that the exponent is also a quadratic function of the values x_i in $\mathbf{x}$. Thus, filtering preserves the Gaussian nature of the state distribution.

Let us first define the general temporal model used with Kalman filtering. Both the transition model and the sensor model are required to be a *linear* transformation with additive Gaussian noise. Thus, we have

$$\begin{aligned} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) &= \mathcal{N}(\mathbf{x}_{t+1}; \mathbf{F}\mathbf{x}_t, \Sigma_x) \\ P(\mathbf{z}_t \mid \mathbf{x}_t) &= \mathcal{N}(\mathbf{z}_t; \mathbf{H}\mathbf{x}_t, \Sigma_z), \end{aligned} \tag{14.21}$$

where $\mathbf{F}$ and Σ_x are matrices describing the linear transition model and transition noise covariance, and $\mathbf{H}$ and Σ_z are the corresponding matrices for the sensor model. Now the update equations for the mean and covariance, in their full, hairy horribleness, are

$$\begin{aligned} \mu_{t+1} &= \mathbf{F}\mu_t + \mathbf{K}_{t+1}(\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\mu_t) \\ \Sigma_{t+1} &= (\mathbf{I} - \mathbf{K}_{t+1}\mathbf{H})(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x), \end{aligned} \tag{14.22}$$

where $\mathbf{K}_{t+1} = (\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top(\mathbf{H}(\mathbf{F}\Sigma_t\mathbf{F}^\top + \Sigma_x)\mathbf{H}^\top + \Sigma_z)^{-1}$ is the **Kalman gain matrix**. Believe it or not, these equations make some intuitive sense. For example, consider the update for the mean state estimate μ. The term $\mathbf{F}\mu_t$ is the *predicted* state at $t+1$, so $\mathbf{H}\mathbf{F}\mu_t$ is the *predicted* observation. Therefore, the term $\mathbf{z}_{t+1} - \mathbf{H}\mathbf{F}\mu_t$ represents the error in the predicted observation. This is multiplied by $\mathbf{K}_{t+1}$ to correct the predicted state; hence, $\mathbf{K}_{t+1}$ is a measure of *how seriously to take the new observation* relative to the prediction. As in Equation (14.20), we also have the property that the variance update is independent of the observations. The sequence of values for Σ_t and $\mathbf{K}_t$ can therefore be computed offline, and the actual calculations required during online tracking are quite modest.

Kalman gain matrix

To illustrate these equations at work, we have applied them to the problem of tracking an object moving on the X–Y plane. The state variables are $\mathbf{X} = (X, Y, \dot{X}, \dot{Y})^\top$, so $\mathbf{F}$, Σ_x, $\mathbf{H}$, and Σ_z are 4×4 matrices. Figure 14.11(a) shows the true trajectory, a series of noisy observations, and the trajectory estimated by Kalman filtering, along with the covariances indicated by the one-standard-deviation contours. The filtering process does a good job of tracking the actual motion, and, as expected, the variance quickly reaches a fixed point.

We can also derive equations for *smoothing* as well as filtering with linear–Gaussian models. The smoothing results are shown in Figure 14.11(b). Notice how the variance in the position estimate is sharply reduced, except at the ends of the trajectory (why?), and that the estimated trajectory is much smoother.

14.4.4 Applicability of Kalman filtering

The Kalman filter and its elaborations are used in a vast array of applications. The "classical" application is in radar tracking of aircraft and missiles. Related applications include acoustic tracking of submarines and ground vehicles and visual tracking of vehicles and people. In a slightly more esoteric vein, Kalman filters are used to reconstruct particle trajectories from bubble-chamber photographs and ocean currents from satellite surface measurements. The range of application is much larger than just the tracking of motion: any system characterized by continuous state variables and noisy measurements will do. Such systems include pulp mills, chemical plants, nuclear reactors, plant ecosystems, and national economies.

The fact that Kalman filtering can be applied to a system does not mean that the results will be valid or useful. The assumptions made—linear–Gaussian transition and sensor models—are very strong. The **extended Kalman filter (EKF)** attempts to overcome nonlinearities in the system being modeled. A system is **nonlinear** if the transition model cannot be described as a matrix multiplication of the state vector, as in Equation (14.21). The EKF

Extended Kalman filter (EKF)

Nonlinear

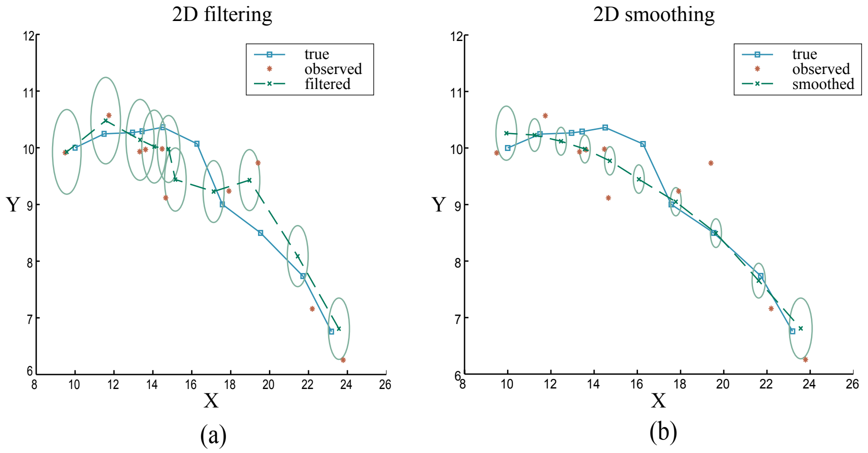


Figure 14.11 (a) Results of Kalman filtering for an object moving on the X–Y plane, showing the true trajectory (left to right), a series of noisy observations, and the trajectory estimated by Kalman filtering. Variance in the position estimate is indicated by the ovals. (b) The results of Kalman smoothing for the same observation sequence.

works by modeling the system as *locally* linear in $\mathbf{x}_t$ in the region of $\mathbf{x}_t = \mu_t$, the mean of the current state distribution. This works well for smooth, well-behaved systems and allows the tracker to maintain and update a Gaussian state distribution that is a reasonable approximation to the true posterior. A detailed example is given in Chapter 26.

What does it mean for a system to be "unsmooth" or "poorly behaved"? Technically, it means that there is significant nonlinearity in system response within the region that is "close" (according to the covariance Σ_t) to the current mean μ_t. To understand this idea in nontechnical terms, consider the example of trying to track a bird as it flies through the jungle. The bird appears to be heading at high speed straight for a tree trunk. The Kalman filter, whether regular or extended, can make only a Gaussian prediction of the location of the bird, and the mean of this Gaussian will be centered on the trunk, as shown in Figure 14.12(a). A reasonable model of the bird, on the other hand, would predict evasive action to one side or the other, as shown in Figure 14.12(b). Such a model is highly nonlinear, because the bird's decision varies sharply depending on its precise location relative to the trunk.

Switching Kalman filter

To handle examples like these, we clearly need a more expressive language for representing the behavior of the system being modeled. Within the control theory community, for which problems such as evasive maneuvering by aircraft raise the same kinds of difficulties, the standard solution is the **switching Kalman filter**. In this approach, multiple Kalman filters run in parallel, each using a different model of the system—for example, one for straight flight, one for sharp left turns, and one for sharp right turns. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data. We will see in the next section that this is simply a special case of the general dynamic Bayesian network model, obtained by adding a discrete "maneuver" state variable to the network shown in Figure 14.9. Switching Kalman filters are discussed further in Exercise 14.KFSW.

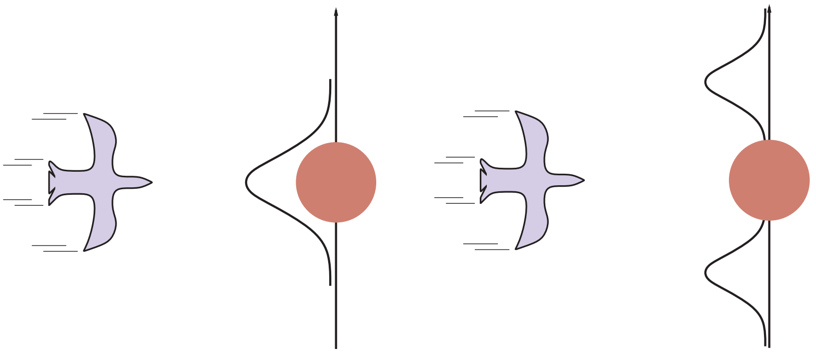


Figure 14.12 A bird flying toward a tree (top views). (a) A Kalman filter will predict the location of the bird using a single Gaussian centered on the obstacle. (b) A more realistic model allows for the bird's evasive action, predicting that it will fly to one side or the other.

14.5 Dynamic Bayesian Networks

Dynamic Bayesian network

Dynamic Bayesian networks, or **DBNs**, extend the semantics of standard Bayesian networks to handle temporal probability models of the kind described in Section 14.1. We have already seen examples of DBNs: the umbrella network in Figure 14.2 and the Kalman filter network in Figure 14.9. In general, each slice of a DBN can have any number of state variables $\mathbf{X}_t$ and evidence variables $\mathbf{E}_t$. For simplicity, we assume that the variables, their links, and their conditional distributions are exactly replicated from slice to slice and that the DBN represents a first-order Markov process, so that each variable can have parents only in its own slice or the immediately preceding slice. In this way, the DBN corresponds to a Bayesian network with infinitely many variables.

It should be clear that every hidden Markov model can be represented as a DBN with a single state variable and a single evidence variable. It is also the case that every discrete-variable DBN can be represented as an HMM; as explained in Section 14.3, we can combine all the state variables in the DBN into a single state variable whose values are all possible tuples of values of the individual state variables. Now, if every HMM is a DBN and every DBN can be translated into an HMM, what's the difference? The difference is that, *by decomposing the state of a complex system into its constituent variables, we can take advantage of* sparseness *in the temporal probability model.* ◀

To see what this means in practice, remember that in Section 14.3 we said that an HMM representation for a temporal process with n discrete variables, each with up to d values, needs a transition matrix of size $O(d^{2n})$. The DBN representation, on the other hand, has size $O(nd^k)$ if the number of parents of each variable is bounded by k. In other words, the DBN representation is linear rather than exponential in the number of variables. For the vacuum robot with 42 possibly dirty locations, the number of probabilities required is reduced from 5×10^{29} to a few thousand.

We have already explained that every Kalman filter model can be represented in a DBN with continuous variables and linear–Gaussian conditional distributions (Figure 14.9). It should be clear from the discussion at the end of the preceding section that *not* every DBN can be represented by a Kalman filter model. In a Kalman filter, the current state distribution

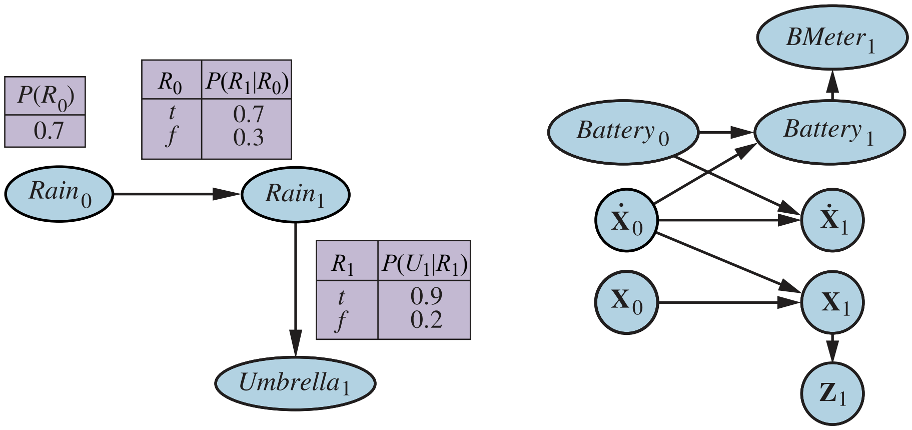


Figure 14.13 Left: Specification of the prior, transition model, and sensor model for the umbrella DBN. Subsequent slices are copies of slice 1. Right: A simple DBN for robot motion in the X–Y plane.

is always a single multivariate Gaussian distribution—that is, a single "bump" in a particular location. DBNs, on the other hand, can model arbitrary distributions.

For many real-world applications, this flexibility is essential. Consider, for example, the current location of my keys. They might be in my pocket, on the bedside table, on the kitchen counter, dangling from the front door, or locked in the car. A single Gaussian bump that included all these places would have to allocate significant probability to the keys being in mid-air above the front garden. Aspects of the real world such as purposive agents, obstacles, and pockets introduce "nonlinearities" that require combinations of discrete and continuous variables in order to get reasonable models.

14.5.1 Constructing DBNs

To construct a DBN, one must specify three kinds of information: the prior distribution over the state variables, $\mathbf{P}(\mathbf{X}_0)$; the transition model $\mathbf{P}(\mathbf{X}_{t+1} \,|\, \mathbf{X}_t)$; and the sensor model $\mathbf{P}(\mathbf{E}_t \,|\, \mathbf{X}_t)$. To specify the transition and sensor models, one must also specify the topology of the connections between successive slices and between the state and evidence variables. Because the transition and sensor models are assumed to be time-homogeneous—the same for all t—it is most convenient simply to specify them for the first slice. For example, the complete DBN specification for the umbrella world is given by the three-node network shown in Figure 14.13(a). From this specification, the complete DBN with an unbounded number of time slices can be constructed as needed by copying the first slice.

Let us now consider a more interesting example: monitoring a battery-powered robot moving in the X–Y plane, as introduced at the end of Section 14.1. First, we need state variables, which will include both $\mathbf{X}_t = (X_t, Y_t)$ for position and $\dot{\mathbf{X}}_t = (\dot{X}_t, \dot{Y}_t)$ for velocity. We assume some method of measuring position—perhaps a fixed camera or onboard GPS (Global Positioning System)—yielding measurements $\mathbf{Z}_t$. The position at the next time step depends on the current position and velocity, as in the standard Kalman filter model. The velocity at the next step depends on the current velocity and the state of the battery. We add $Battery_t$ to represent the actual battery charge level, which has as parents the previous battery level and the velocity, and we add $BMeter_t$, which measures the battery charge level. This gives us the basic model shown in Figure 14.13(b).

It is worth looking in more depth at the nature of the sensor model for $BMeter_t$. Let us suppose, for simplicity, that both $Battery_t$ and $BMeter_t$ can take on discrete values 0 through 5. (Exercise 14.BATT asks you to relate this discrete model to a corresponding continuous model.) If the meter is always accurate, then the CPT $\mathbf{P}(BMeter_t \mid Battery_t)$ should have probabilities of 1.0 "along the diagonal" and probabilities of 0.0 elsewhere. In reality, noise always creeps into measurements. For continuous measurements, a Gaussian distribution with a small variance might be used.[7] For our discrete variables, we can approximate a Gaussian using a distribution in which the probability of error drops off in the appropriate way, so that the probability of a large error is very small. We use the term **Gaussian error model** to cover both the continuous and discrete versions.

Gaussian error model

Anyone with hands-on experience of robotics, computerized process control, or other forms of automatic sensing will readily testify to the fact that small amounts of measurement noise are often the least of one's problems. Real sensors *fail*. When a sensor fails, it does not necessarily send a signal saying, "Oh, by the way, the data I'm about to send you is a load of nonsense." Instead, it simply sends the nonsense. The simplest kind of failure is called a **transient failure**, where the sensor occasionally decides to send some nonsense. For example, the battery level sensor might have a habit of sending a reading of 0 when someone bumps the robot, even if the battery is fully charged.

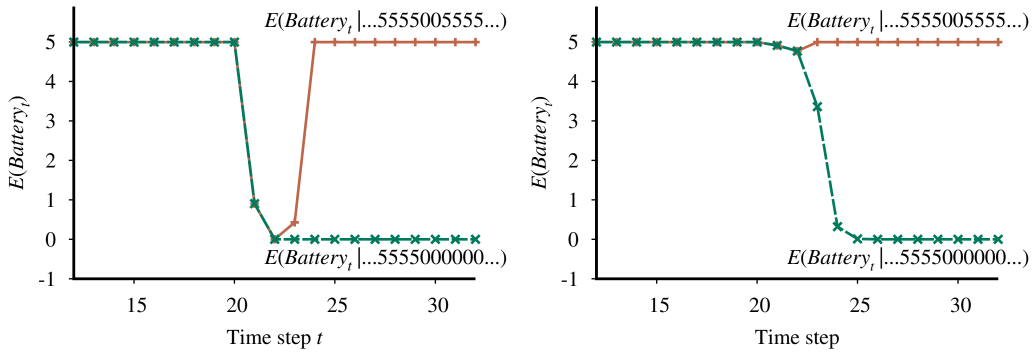
Transient failure

Let's see what happens when a transient failure occurs with a Gaussian error model that doesn't accommodate such failures. Suppose, for example, that the robot is sitting quietly and observes 20 consecutive battery readings of 5. Then the battery meter has a temporary seizure and the next reading is $BMeter_{21} = 0$. What will the simple Gaussian error model lead us to believe about $Battery_{21}$? According to Bayes' rule, the answer depends on both the sensor model $\mathbf{P}(BMeter_{21} = 0 \mid Battery_{21})$ and the prediction $\mathbf{P}(Battery_{21} \mid BMeter_{1:20})$. If the probability of a large sensor error is significantly less than the probability of a transition to $Battery_{21} = 0$, even if the latter is very unlikely, then the posterior distribution will assign a high probability to the battery's being empty.

A second reading of 0 at $t = 22$ will make this conclusion almost certain. If the transient failure then disappears and the reading returns to 5 from $t = 23$ onwards, the estimate for the battery level will quickly return to 5. (This does not mean the algorithm thinks the battery magically recharged itself, which may be physically impossible; instead, the algorithm now believes that the battery was never low and the extremely unlikely hypothesis that the battery meter had two consecutive huge errors must be the right explanation.) This course of events is illustrated in the upper curve of Figure 14.14(a), which shows the expected value (see Appendix A) of $Battery_t$ over time, using a discrete Gaussian error model.

Despite the recovery, there is a time ($t = 22$) when the robot is convinced that its battery is empty; presumably, then, it should send out a mayday signal and shut down. Alas, its oversimplified sensor model has led it astray. The moral of the story is simple: *for the system to handle sensor failure properly, the sensor model must include the possibility of failure.*

[7] Strictly speaking, a Gaussian distribution is problematic because it assigns nonzero probability to large negative charge levels. The **beta distribution** is sometimes a better choice for a variable whose range is restricted.



(a)

(b)

Figure 14.14 (a) Upper curve: trajectory of the expected value of $Battery_t$ for an observation sequence consisting of all 5s except for 0s at $t=21$ and $t=22$, using a simple Gaussian error model. Lower curve: trajectory when the observation remains at 0 from $t=21$ onwards. (b) The same experiment run with the transient failure model. The transient failure is handled well, but the persistent failure results in excessive pessimism about the battery charge.

The simplest kind of failure model for a sensor allows a certain probability that the sensor will return some completely incorrect value, regardless of the true state of the world. For example, if the battery meter fails by returning 0, we might say that

$$P(BMeter_t=0\,|\,Battery_t=5)=0.03\,,$$

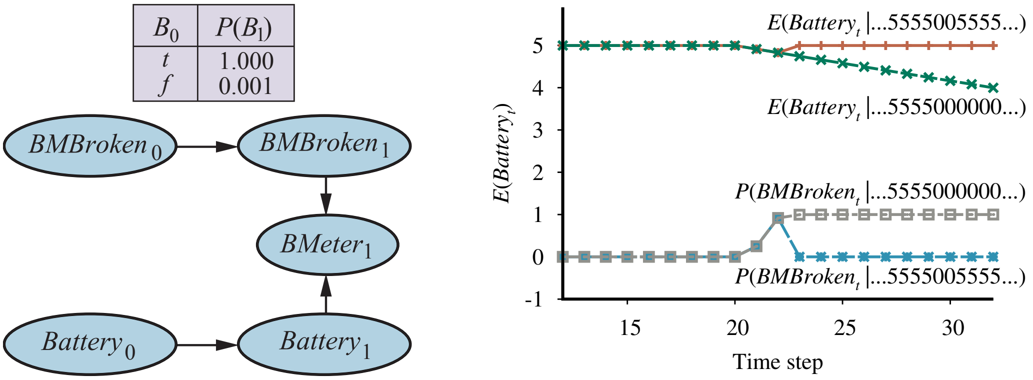
Transient failure model

which is presumably much larger than the probability assigned by the simple Gaussian error model. Let's call this the **transient failure model**. How does it help when we are faced with a reading of 0? Provided that the *predicted* probability of an empty battery, according to the readings so far, is much less than 0.03, then the best explanation of the observation $BMeter_{21}=0$ is that the sensor has temporarily failed. Intuitively, we can think of the belief about the battery level as having a certain amount of "inertia" that helps to overcome temporary blips in the meter reading. The upper curve in Figure 14.14(b) shows that the transient failure model can handle transient failures without a catastrophic change in beliefs.

So much for temporary blips. What about a persistent sensor failure? Sadly, failures of this kind are all too common. If the sensor returns 20 readings of 5 followed by 20 readings of 0, then the transient sensor failure model described in the preceding paragraph will result in the robot gradually coming to believe that its battery is empty when in fact it may be that the meter has failed. The lower curve in Figure 14.14(b) shows the belief "trajectory" for this case. By $t=25$—five readings of 0—the robot is convinced that its battery is empty. Obviously, we would prefer the robot to believe that its battery meter is broken—if indeed this is the more likely event.

Persistent failure model

Unsurprisingly, to handle persistent failure, we need a **persistent failure model** that describes how the sensor behaves under normal conditions and after failure. To do this, we need to augment the state of the system with an additional variable, say, *BMBroken*, that describes the status of the battery meter. The persistence of failure must be modeled by an



B_0	$P(B_1)$
t	1.000
f	0.001

(a) (b)

Figure 14.15 (a) A DBN fragment showing the sensor status variable required for modeling persistent failure of the battery sensor. (b) Upper curves: trajectories of the expected value of $Battery_t$ for the "transient failure" and "permanent failure" observations sequences. Lower curves: probability trajectories for *BMBroken* given the two observation sequences.

Persistence arc

arc linking $BMBroken_0$ to $BMBroken_1$. This **persistence arc** has a CPT that gives a small probability of failure in any given time step, say, 0.001, but specifies that the sensor stays broken once it breaks. When the sensor is OK, the sensor model for *BMeter* is identical to the transient failure model; when the sensor is broken, it says *BMeter* is always 0, regardless of the actual battery charge.

The persistent failure model for the battery sensor is shown in Figure 14.15(a). Its performance on the two data sequences (temporary blip and persistent failure) is shown in Figure 14.15(b). There are several things to notice about these curves. First, in the case of the temporary blip, the probability that the sensor is broken rises significantly after the second 0 reading, but immediately drops back to zero once a 5 is observed. Second, in the case of persistent failure, the probability that the sensor is broken rises quickly to almost 1 and stays there. Finally, once the sensor is known to be broken, the robot can only assume that its battery discharges at the "normal" rate. This is shown by the gradually descending level of $E(Battery_t \,|\, \ldots)$.

So far, we have merely scratched the surface of the problem of representing complex processes. The variety of transition models is huge, encompassing topics as disparate as modeling the human endocrine system and modeling multiple vehicles driving on a freeway. Sensor modeling is also a vast subfield in itself. But dynamic Bayesian networks can model even subtle phenomena, such as sensor drift, sudden decalibration, and the effects of exogenous conditions (such as weather) on sensor readings.

14.5.2 Exact inference in DBNs

Having sketched some ideas for representing complex processes as DBNs, we now turn to the question of inference. In a sense, this question has already been answered: dynamic Bayesian networks *are* Bayesian networks, and we already have algorithms for inference in Bayesian networks. Given a sequence of observations, one can construct the full Bayesian network rep-

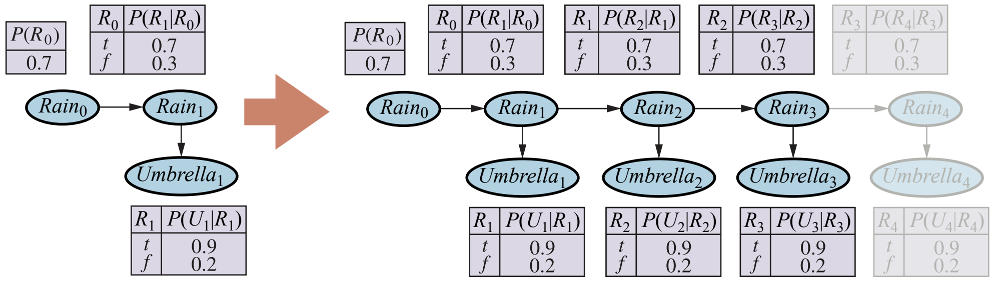

$P(R_0)$ 0.7
R_0 $P(R_1|R_0)$ t 0.7 f 0.3
$Rain_0$
$Rain_1$
$Umbrella_1$
R_1 $P(U_1|R_1)$ t 0.9 f 0.2
$P(R_0)$ 0.7
R_0 $P(R_1|R_0)$ t 0.7 f 0.3
R_1 $P(R_2|R_1)$ t 0.7 f 0.3
R_2 $P(R_3|R_2)$ t 0.7 f 0.3
R_3 $P(R_4|R_3)$ t 0.7 f 0.3
$Rain_0$
$Rain_1$
$Rain_2$
$Rain_3$
$Rain_4$
$Umbrella_1$
$Umbrella_2$
$Umbrella_3$
$Umbrella_4$
R_1 $P(U_1|R_1)$ t 0.9 f 0.2
R_2 $P(U_2|R_2)$ t 0.9 f 0.2
R_3 $P(U_3|R_3)$ t 0.9 f 0.2
R_4 $P(U_4|R_4)$ t 0.9 f 0.2

Figure 14.16 Unrolling a dynamic Bayesian network: slices are replicated to accommodate the observation sequence $Umbrella_{1:3}$. Further slices have no effect on inferences within the observation period.

resentation of a DBN by replicating slices until the network is large enough to accommodate the observations, as in Figure 14.16. This technique is called **unrolling**. (Technically, the DBN is equivalent to the semi-infinite network obtained by unrolling forever. Slices added beyond the last observation have no effect on inferences within the observation period and can be omitted.) Once the DBN is unrolled, one can use any of the inference algorithms—variable elimination, clustering methods, and so on—described in Chapter 13.

Unfortunately, a naive application of unrolling would not be particularly efficient. If we want to perform filtering or smoothing with a long sequence of observations $\mathbf{e}_{1:t}$, the unrolled network would require $O(t)$ space and would thus grow without bound as more observations were added. Moreover, if we simply run the inference algorithm anew each time an observation is added, the inference time per update will also increase as $O(t)$.

Looking back to Section 14.2.1, we see that constant time and space per filtering update can be achieved if the computation can be done recursively. Essentially, the filtering update in Equation (14.5) works by *summing out* the state variables of the previous time step to get the distribution for the new time step. Summing out variables is exactly what the **variable elimination** (Figure 13.13) algorithm does, and it turns out that running variable elimination with the variables in temporal order exactly mimics the operation of the recursive filtering update in Equation (14.5). The modified algorithm keeps at most two slices in memory at any one time: starting with slice 0, we add slice 1, then sum out slice 0, then add slice 2, then sum out slice 1, and so on. In this way, we can achieve constant space and time per filtering update. (The same performance can be achieved by suitable modifications to the clustering algorithm.) Exercise 14.DBNE asks you to verify this fact for the umbrella network.

So much for the good news; now for the bad news: It turns out that the "constant" for the per-update time and space complexity is, in almost all cases, exponential in the number of state variables. What happens is that, as the variable elimination proceeds, the factors grow to include all the state variables (or, more precisely, all those state variables that have parents in the previous time slice). The maximum factor size is $O(d^{n+k})$ and the total update cost per step is $O(nd^{n+k})$, where d is the domain size of the variables and k is the maximum number of parents of any state variable.

Of course, this is much less than the cost of HMM updating, which is $O(d^{2n})$, but it is still infeasible for large numbers of variables. This grim fact means is that *even though we can use DBNs to* represent *very complex temporal processes with many sparsely connected variables,* *we cannot* reason *efficiently and exactly about those processes.* The DBN model itself, which represents the prior joint distribution over all the variables, is factorable into its constituent CPTs, but the posterior joint distribution conditioned on an observation sequence—that is, the forward message—is generally *not* factorable. The problem is intractable in general, so we must fall back on approximate methods.

14.5.3 Approximate inference in DBNs

Section 13.4 described two approximation algorithms: likelihood weighting (Figure 13.18) and Markov chain Monte Carlo (MCMC, Figure 13.20). Of the two, the former is most easily adapted to the DBN context. (An MCMC filtering algorithm is described briefly in the notes at the end of this chapter.) We will see, however, that several improvements are required over the standard likelihood weighting algorithm before a practical method emerges.

Recall that likelihood weighting works by sampling the nonevidence nodes of the network in topological order, weighting each sample by the likelihood it accords to the observed evidence variables. As with the exact algorithms, we could apply likelihood weighting directly to an unrolled DBN, but this would suffer from the same problems of increasing time and space requirements per update as the observation sequence grows. The problem is that the standard algorithm runs each sample in turn, all the way through the network.

Instead, we can simply run all N samples together through the DBN, one slice at a time. The modified algorithm fits the general pattern of filtering algorithms, with the set of N samples as the forward message. The first key innovation, then, is to *use the samples themselves as an approximate representation of the current state distribution.* This meets the requirement of a "constant" time per update, although the constant depends on the number of samples required to maintain an accurate approximation. There is also no need to unroll the DBN, because we need to have in memory only the current slice and the next slice. This approach is called **sequential importance sampling** or SIS.

Sequential importance sampling

In our discussion of likelihood weighting in Chapter 13, we pointed out that the algorithm's accuracy suffers if the evidence variables are "downstream" from the variables being sampled, because in that case the samples are generated without any influence from the evidence and will nearly all have very low weights.

Now if we look at the typical structure of a DBN—say, the umbrella DBN in Figure 14.16—we see that indeed the early state variables will be sampled without the benefit of the later evidence. In fact, looking more carefully, we see that *none* of the state variables have *any* evidence variables among its ancestors! Hence, although the weight of each sample will depend on the evidence, the actual set of samples generated will be *completely independent* of the evidence. For example, even if the boss brings in the umbrella every day, the sampling process could still hallucinate endless days of sunshine.

What this means in practice is that the fraction of samples that remain reasonably close to the actual series of events (and therefore have non-negligible weights) drops exponentially with t, the length of the sequence. In other words, to maintain a given level of accuracy, we need to increase the number of samples exponentially with t. Given that a real-time filtering algorithm can use only a bounded number of samples, what happens in practice is that the error blows up after a very small number of update steps. Figure 14.19 on page 512 shows this effect for SIS applied to the grid-world localization problem from Section 14.3: even with 100,000 samples, the SIS approximation fails completely after about 20 steps.

```
function PARTICLE-FILTERING($\mathbf{e}, N, dbn$) returns a set of samples for the next time step
  inputs: $\mathbf{e}$, the new incoming evidence
          $N$, the number of samples to be maintained
          $dbn$, a DBN defined by $\mathbf{P}(\mathbf{X}_0)$, $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0)$, and $\mathbf{P}(\mathbf{E}_1 \mid \mathbf{X}_1)$
  persistent: $S$, a vector of samples of size $N$, initially generated from $\mathbf{P}(\mathbf{X}_0)$
  local variables: $W$, a vector of weights of size $N$

  for $i$ = 1 to $N$ do
     $S[i] \leftarrow$ sample from $\mathbf{P}(\mathbf{X}_1 \mid \mathbf{X}_0 = S[i])$        // step 1
     $W[i] \leftarrow \mathbf{P}(\mathbf{e} \mid \mathbf{X}_1 = S[i])$                     // step 2
  $S \leftarrow$ WEIGHTED-SAMPLE-WITH-REPLACEMENT($N, S, W$)              // step 3
  return $S$
```

Figure 14.17 The particle filtering algorithm implemented as a recursive update operation with state (the set of samples). Each of the sampling operations involves sampling the relevant slice variables in topological order, much as in PRIOR-SAMPLE. The WEIGHTED-SAMPLE-WITH-REPLACEMENT operation can be implemented to run in $O(N)$ expected time. The step numbers refer to the description in the text.

Clearly, we need a better solution. The second key innovation is to *focus the set of samples on the high-probability regions of the state space.* This can be done by throwing away samples that have very low weight, according to the observations, while replicating those that have high weight. In that way, the population of samples will stay reasonably close to reality. If we think of samples as a resource for modeling the posterior distribution, then it makes sense to use more samples in regions of the state space where the posterior is higher.

Particle filtering

A family of algorithms called **particle filtering** is designed to do just that. (Another early name was **sequential importance sampling with resampling**, but for some reason it failed on catch on.) Particle filtering works as follows: First, we generate a population of N samples from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. Then the update cycle is repeated for each time step:

1. Each sample is propagated forward by sampling the next state value $\mathbf{x}_{t+1}$ given the current value $\mathbf{x}_t$ for the sample, based on the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid \mathbf{x}_t)$.
2. Each sample is weighted by the likelihood it assigns to the new evidence, $P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1})$.
3. The population is *resampled* to generate a new population of N samples. Each new sample is selected from the current population; the probability that a particular sample is selected is proportional to its weight. The new samples are unweighted.

The algorithm is shown in detail in Figure 14.17, and its operation for the umbrella DBN is illustrated in Figure 14.18.

We can show that this algorithm is consistent—gives the correct probabilities as N tends to infinity—by examining the operations in one update cycle. We assume that the sample population starts with a correct representation of the forward message—that is, $\mathbf{f}_{1:t} = \mathbf{P}(\mathbf{X}_t \mid \mathbf{e}_{1:t})$ at time t. Writing $N(\mathbf{x}_t \mid \mathbf{e}_{1:t})$ for the number of samples occupying state $\mathbf{x}_t$ after observations $\mathbf{e}_{1:t}$ have been processed, we therefore have

$$N(\mathbf{x}_t \mid \mathbf{e}_{1:t})/N = P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \qquad (14.23)$$

for large N. Now we propagate each sample forward by sampling the state variables at $t+1$,

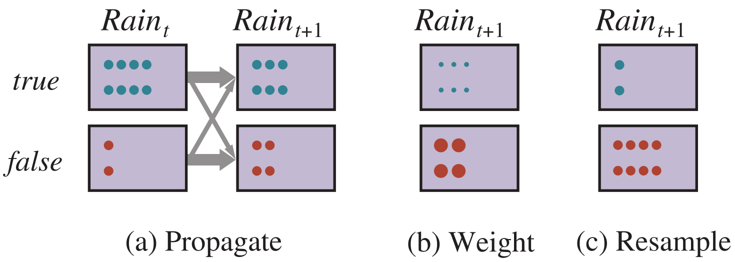


Figure 14.18 The particle filtering update cycle for the umbrella DBN with $N=10$, showing the sample populations of each state. (a) At time t, 8 samples indicate *rain* and 2 indicate $\neg$*rain*. Each is propagated forward by sampling the next state through the transition model. At time $t+1$, 6 samples indicate *rain* and 4 indicate $\neg$*rain*. (b) $\neg$*umbrella* is observed at $t+1$. Each sample is weighted by its likelihood for the observation, as indicated by the size of the circles. (c) A new set of 10 samples is generated by weighted random selection from the current set, resulting in 2 samples that indicate *rain* and 8 that indicate $\neg$*rain*.

given the values for the sample at t. The number of samples reaching state $\mathbf{x}_{t+1}$ from each $\mathbf{x}_t$ is the transition probability times the population of $\mathbf{x}_t$; hence, the total number of samples reaching $\mathbf{x}_{t+1}$ is

$$N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t}) = \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) N(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \,.$$

Now we weight each sample by its likelihood for the evidence at $t+1$. A sample in state $\mathbf{x}_{t+1}$ receives weight $P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1})$. The total weight of the samples in $\mathbf{x}_{t+1}$ after seeing $\mathbf{e}_{t+1}$ is therefore

$$W(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) = P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t}) \,.$$

Now for the resampling step. Since each sample is replicated with probability proportional to its weight, the number of samples in state $\mathbf{x}_{t+1}$ after resampling is proportional to the total weight in $\mathbf{x}_{t+1}$ before resampling:

$$\begin{aligned}
N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1})/N &= \alpha W(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) \\
&= \alpha P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) N(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t}) \\
&= \alpha P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) N(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\
&= \alpha N P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \quad \text{(by 14.23)} \\
&= \alpha' P(\mathbf{e}_{t+1} \mid \mathbf{x}_{t+1}) \sum_{\mathbf{x}_t} P(\mathbf{x}_{t+1} \mid \mathbf{x}_t) P(\mathbf{x}_t \mid \mathbf{e}_{1:t}) \\
&= P(\mathbf{x}_{t+1} \mid \mathbf{e}_{1:t+1}) \quad \text{(by 14.5).}
\end{aligned}$$

Therefore the sample population after one update cycle correctly represents the forward message at time $t+1$.

Particle filtering is *consistent*, therefore, but is it *efficient*? For many practical cases, it seems that the answer is yes: particle filtering seems to maintain a good approximation to the true posterior using a constant number of samples. Figure 14.19 shows that particle filtering does a good job on the grid-world localization problem with only a thousand samples. It also

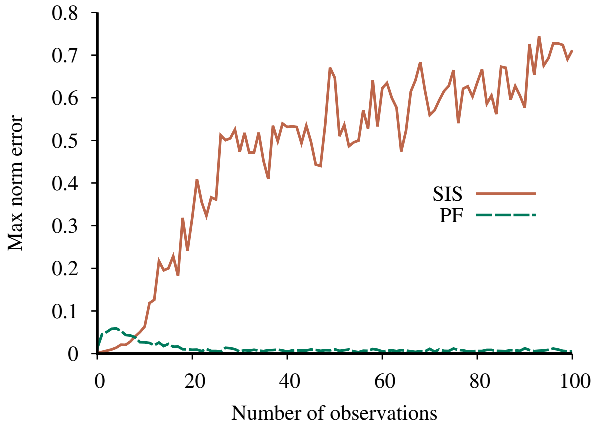


Figure 14.19 Max norm error in the grid-world location estimate (compared to exact inference) for likelihood weighting (sequential importance sampling) with 100,000 samples and particle filtering with 1,000 samples; data averaged over 50 runs.

works on real-world problems: the algorithm supports thousands of applications in science and engineering. (Some references are given at the end of the chapter.) It handles combinations of discrete and continuous variables as well as nonlinear and non-Gaussian models for continuous variables. Under certain assumptions—in particular, that the probabilities in the transition and sensor models are bounded away from 0 and 1—it is also possible to prove that the approximation maintains bounded error with high probability, as the figure suggests.

The particle filtering algorithm does have weaknesses, however. Let's see how it performs for the vacuum world with dirt added. Recall from Section 14.3.2 that this increases the state space size by a factor of 2^{42}, making exact HMM inference infeasible. We want the robot to wander around and build a map of where the dirt is located. (This is a simple example of **simultaneous localization and mapping** or **SLAM**, which we cover in more depth in Chapter 26.) Let $Dirt_{i,t}$ mean that square i is dirty at time t and let $DirtSensor_t$ be true if and only if the robot detects dirt at time t. We'll assume that, in any given square, dirt persists with probability p, whereas a clean square becomes dirty with probability $1-p$ (which means that each square is dirty half the time, on average). The robot has a dirt sensor for its current location; the sensor is accurate with probability 0.9. Figure 14.20 shows the DBN.

For simplicity, we'll start by assuming that the robot has a perfect location sensor, rather than the noisy wall sensor. The algorithm's performance is shown in Figure 14.21(a), where its estimates for dirt are compared to the results of exact inference. (We'll see shortly how exact inference is possible.) For low values of the dirt persistence p, the error remains small—but this is no great achievement, because for every square the true posterior for dirt is close to 0.5 if the robot hasn't visited that square recently. For higher values of p, the dirt stays around longer, so visiting a square yields more useful information that is valid over a longer period. Perhaps surprisingly, particle filtering does *worse* for higher values of p. It fails completely when $p=1$, even though that seems like the easiest case: the dirt arrives at time 0 and stays put forever, so after a few tours of the world, the robot should have a close-to-perfect dirt map. Why does particle filtering fail in this case?

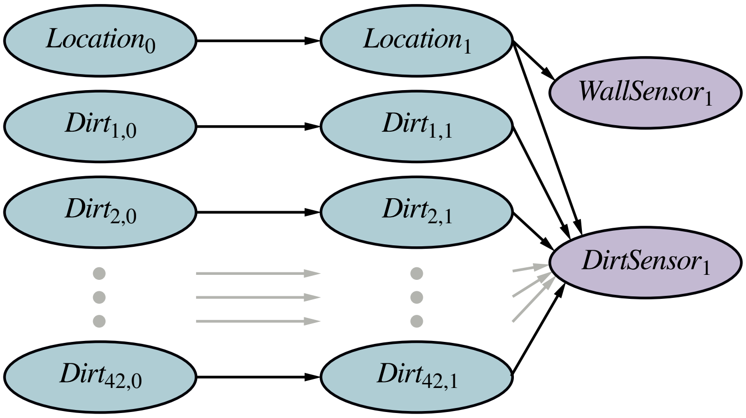


Figure 14.20 A dynamic Bayes net for simultaneous localization and mapping in the stochastic-dirt vacuum world. Dirty squares persist with probability p, and clean squares become dirty with probability $1-p$. The local dirt sensor is 90% accurate, for the square in which the robot is currently located.

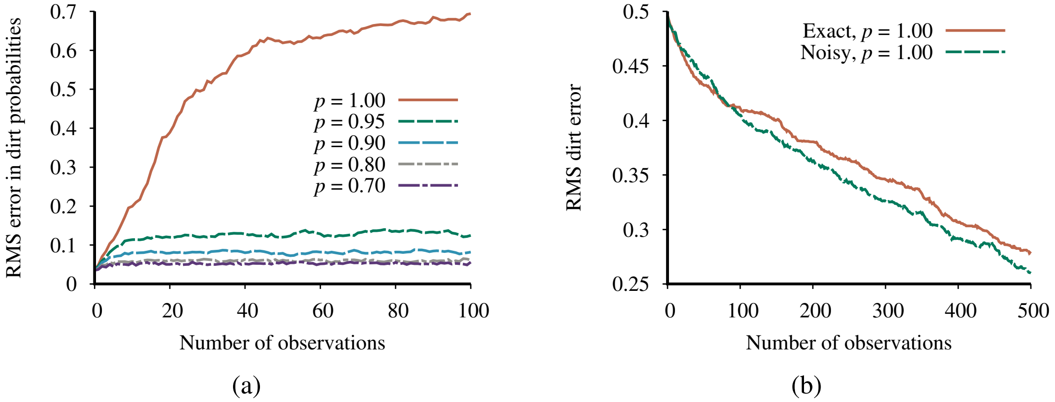


Figure 14.21 (a) Performance of the standard particle filtering algorithm with 1,000 particles, showing RMS error in marginal dirt probabilities compared to exact inference for different values of the dirt persistence p. (b) Performance of Rao-Blackwellized particle filtering (100 particles) compared to ground truth, for both exact location sensing and noisy wall sensing and with deterministic dirt. Data averaged over 20 runs.

It turns out that the theoretical condition requiring that "the probabilities in the transition and sensor models are strictly greater than 0 and less than 1" is more than mere mathematical pedantry. What happens is first each particle initially contains 42 guesses from $\mathbf{P}(\mathbf{X}_0)$ about which squares have dirt and which do not. Then, the state for each particle is projected forward in time according to the transition model. Unfortunately, the transition model for deterministic dirt is deterministic: the dirt stays exactly where it is. Thus, the initial guesses in each particle are never updated by the evidence.

The chance that the initial guesses are all correct is 2^{-42} or about 2×10^{-13}, so it is vanishingly unlikely that a thousand particles (or even a million particles) will include one with the correct dirt map. Typically, the best particle out of a thousand will get about 32 right and 10 wrong, and usually there will be only one such particle, or perhaps a handful. One of those best particles will come to dominate the total likelihood as time progresses and the diversity of the population of particles will collapse. Then, because all the particles agree on a single, incorrect map, the algorithm becomes convinced that that map is correct and never changes its mind.

Fortunately, the problem of simultaneous localization and mapping has a special structure: conditioned on the sequence of robot locations, the dirt statuses of the individual squares are independent (Exercise 14.RBPF). More specifically,

$$\begin{aligned}&\mathbf{P}(Dirt_{1,0:t},\ldots,Dirt_{42,0:t}\,|\,DirtSensor_{1:t},WallSensor_{1:t},Location_{1:t})\\ &= \prod_i \mathbf{P}(Dirt_{i,0:t}\,|\,DirtSensor_{1:t},Location_{1:t})\,. \qquad (14.24)\end{aligned}$$

Rao-Blackwellization

This means it is useful to apply a statistical trick called **Rao-Blackwellization**, which is based on the simple idea that exact inference is always more accurate than sampling, even if it's only for a subset of the variables. (See Exercise 14.RAOB.) For the SLAM problem, we run particle filtering on the robot location and then, for each particle, we run exact HMM inference for each dirt square independently, conditioned on the location sequence in that particle. Each particle therefore contains a sampled location plus 42 exact marginal posteriors for the 42 squares—exact, that is, assuming that the hypothesized location trajectory followed by that particle is correct. This approach, called the **Rao-Blackwellized particle filter**, handles the case of deterministic dirt with no difficulty, gradually building an exact dirt map with either exact location sensing or noisy wall sensing, as shown in Figure 14.21(b).

Rao-Blackwellized particle filter

In cases that do not satisfy the kind of conditional independence structure exemplified by Equation (14.24), Rao-Blackwellization is not applicable. The notes at the end of the chapter mention a number of algorithms that have been proposed to handle the general problem of filtering with static variables. None has the elegance and broad applicability of the particle filter, but several are effective in practice on certain classes of problems.

Summary

This chapter has addressed the general problem of representing and reasoning about probabilistic temporal processes. The main points are as follows:

- The changing state of the world is handled by using a set of random variables to represent the state at each point in time.
- Representations can be designed to (roughly) satisfy the **Markov property**, so that the future is independent of the past given the present. Combined with the assumption that the process is **time-homogeneous**, this greatly simplifies the representation.
- A temporal probability model can be thought of as containing a **transition model** describing the state evolution and a **sensor model** describing the observation process.
- The principal inference tasks in temporal models are **filtering** (**state estimation**), **prediction**, **smoothing**, and computing the **most likely explanation**. Each of these tasks

can be achieved using simple, recursive algorithms whose run time is linear in the length of the sequence.

- Three families of temporal models were studied in more depth: **hidden Markov models**, **Kalman filters**, and **dynamic Bayesian networks** (which include the other two as special cases).
- Unless special assumptions are made, as in Kalman filters, exact inference with many state variables is intractable. In practice, the **particle filtering** algorithm and its descendants are an effective family of approximation algorithms.

Bibliographical and Historical Notes

Many of the basic ideas for estimating the state of dynamical systems came from the mathematician C. F. Gauss (1809), who formulated a deterministic least-squares algorithm for the problem of estimating orbits from astronomical observations. A. A. Markov (1913) developed what was later called the **Markov assumption** in his analysis of stochastic processes; he estimated a first-order Markov chain on letters from the text of *Eugene Onegin*. The general theory of Markov chains and their mixing times is covered by Levin *et al.* (2008).

Significant classified work on filtering was done during World War II by Wiener (1942) for continuous-time processes and by Kolmogorov (1941) for discrete-time processes. Although this work led to important technological developments over the next 20 years, its use of a frequency-domain representation made many calculations quite cumbersome. Direct state-space modeling of the stochastic process turned out to be simpler, as shown by Peter Swerling (1959) and Rudolf Kalman (1960). The latter paper described what is now known as the Kalman filter for forward inference in linear systems with Gaussian noise; Kalman's results had, however, been obtained previously by the Danish astronomer Thorvold Thiele (1880) and by the Russian physicist Ruslan Stratonovich (1959). After a visit to NASA Ames Research Center in 1960, Kalman saw the applicability of the method to the tracking of rocket trajectories, and the filter was later implemented for the Apollo missions.

Key results on smoothing were derived by Rauch *et al.* (1965), and the impressively named Rauch–Tung–Striebel smoother is still a standard technique today. Many early results are gathered in Gelb (1974). Bar-Shalom and Fortmann (1988) give a more modern treatment with a Bayesian flavor, as well as many references to the vast literature on the subject. Chatfield (1989) and Box *et al.* (2016) cover the control theory approach to time series analysis.

The hidden Markov model and associated algorithms for inference and learning, including the forward–backward algorithm, were developed by Baum and Petrie (1966). The Viterbi algorithm first appeared in (Viterbi, 1967). Similar ideas also appeared independently in the Kalman filtering community (Rauch *et al.*, 1965).

The forward–backward algorithm was one of the main precursors of the general formulation of the EM algorithm (Dempster *et al.*, 1977); see also Chapter 21. Constant-space smoothing appears in Binder *et al.* (1997b), as does the divide-and-conquer algorithm developed in Exercise 14.ISLE. Constant-time fixed-lag smoothing for HMMs first appeared in Russell and Norvig (2003).

HMMs have found many applications in language processing (Charniak, 1993), speech recognition (Rabiner and Juang, 1993), machine translation (Och and Ney, 2003), computational biology (Krogh *et al.*, 1994; Baldi *et al.*, 1994), financial economics (Bhar and Hamori, 2004) and other fields. There have been several extensions to the basic HMM model: for example, the Hierarchical HMM (Fine *et al.*, 1998) and Layered HMM (Oliver *et al.*, 2004) introduce structure back into the model, replacing the single state variable of HMMs.

Dynamic Bayesian networks (DBNs) can be viewed as a sparse encoding of a Markov process and were first used in AI by Dean and Kanazawa (1989b), Nicholson and Brady (1992), and Kjaerulff (1992). The last work extends the HUGIN Bayes net system to accommodate dynamic Bayesian networks. The book by Dean and Wellman (1991) helped popularize DBNs and the probabilistic approach to planning and control within AI. Murphy (2002) provides a thorough analysis of DBNs.

Dynamic Bayesian networks have become popular for modeling a variety of complex motion processes in computer vision (Huang *et al.*, 1994; Intille and Bobick, 1999). Like HMMs, they have found applications in speech recognition (Zweig and Russell, 1998; Livescu *et al.*, 2003), robot localization (Theocharous *et al.*, 2004), and genomics (Murphy and Mian, 1999; Li *et al.*, 2011). Other application areas include gesture analysis (Suk *et al.*, 2010), driver fatigue detection (Yang *et al.*, 2010), and urban traffic modeling (Hofleitner *et al.*, 2012).

The link between HMMs and DBNs, and between the forward–backward algorithm and Bayesian network propagation, was explicated by Smyth *et al.* (1997). A further unification with Kalman filters (and other statistical models) appears in Roweis and Ghahramani (1999). Procedures exist for learning the parameters (Binder *et al.*, 1997a; Ghahramani, 1998) and structures (Friedman *et al.*, 1998) of DBNs. **Continuous-time Bayesian networks** (Nodelman *et al.*, 2002) are the discrete-state, continuous-time analog of DBNs, avoiding the need to choose a particular duration for time steps.

The first sampling algorithms for filtering (also called sequential Monte Carlo methods) were developed in the control theory community by Handschin and Mayne (1969), and the resampling idea that is the core of particle filtering appeared in a Russian control journal (Zaritskii *et al.*, 1975). It was later reinvented in statistics as **sequential importance sampling with resampling**, or **SIR** (Rubin, 1988; Liu and Chen, 1998), in control theory as particle filtering (Gordon *et al.*, 1993; Gordon, 1994), in AI as **survival of the fittest** (Kanazawa *et al.*, 1995), and in computer vision as **condensation** (Isard and Blake, 1996).

Evidence reversal

The paper by Kanazawa *et al.* (1995) includes an improvement called **evidence reversal** whereby the state at time $t+1$ is sampled conditional on both the state at time t and the evidence at time $t+1$. This allows the evidence to influence sample generation directly and was proved by Doucet (1997) and Liu and Chen (1998) to reduce the approximation error.

Particle filtering has been applied in many areas, including tracking complex motion patterns in video (Isard and Blake, 1996), predicting the stock market (de Freitas *et al.*, 2000), and diagnosing faults on planetary rovers (Verma *et al.*, 2004). Since its invention, tens of thousands of papers have been published on applications and variants of the algorithm. Scalable implementations on parallel hardware have become important; although one might think it straightforward to distribute N particles across up to N processor threads, the basic algorithm requires synchronized communication among threads for the resampling step (Hendeby *et al.*, 2010). The **particle cascade algorithm** (Paige *et al.*, 2015) removes the synchronization requirement, resulting in much faster parallel computation.

The **Rao-Blackwellized particle filter** is due to Doucet *et al.* (2000) and Murphy and Russell (2001); its application to practical localization and mapping problems in robotics is described in Chapter 26. Many other algorithms have been proposed to handle more general filtering problems with static or nearly-static variables, including the resample–move algorithm (Gilks and Berzuini, 2001), the Liu–West algorithm (Liu and West, 2001), the Storvik filter (Storvik, 2002), the extended parameter filter (Erol *et al.*, 2013), and the assumed parameter filter (Erol *et al.*, 2017). The latter is a hybrid of particle filtering with a much older idea called **assumed-density filter**. An assumed-density filter assumes that the posterior distribution over states at time t belongs to a particular finitely parameterized family; if the projection and update steps take it outside this family, the distribution is projected back to give the best approximation within the family. For DBNs, the Boyen–Koller algorithm (Boyen *et al.*, 1999) and the **factored frontier** algorithm (Murphy and Weiss, 2001) assume that the posterior distribution can be approximated well by a product of small factors.

Assumed-density filter

Factored frontier

MCMC methods (see Section 13.4.2) can be applied to the filtering problem; for example, Gibbs sampling can be applied directly to an unrolled DBN. The **particle MCMC** family of algorithms (Andrieu *et al.*, 2010; Lindsten *et al.*, 2014) combines MCMC on the unrolled temporal model with particle filtering to generate the MCMC proposals; although it provably converges to the correct posterior distribution in the general case (i.e., with both static and dynamic variables), it is an offline algorithm. To avoid the problem of increasing update times as the unrolled network grows, the **decayed MCMC** filter (Marthi *et al.*, 2002) prefers to sample more recent state variables, with a probability that decreases for variables further in the past.

Particle MCMC

Decayed MCMC

The book by Doucet *et al.* (2001) collects many important papers on **sequential Monte Carlo** (SMC) algorithms, of which particle filtering is the most important instance. There are useful tutorials by Arulampalam *et al.* (2002) and Doucet and Johansen (2011). There are also several theoretical results concerning conditions under which SMC methods retain a bounded error indefinitely compared to the true posterior (Crisan and Doucet, 2002; Del Moral, 2004; Del Moral *et al.*, 2006).

Sequential Monte Carlo