Chapter 15

Kalman Variations: Least Squares for Dynamical Systems

The targets that we track are physical objects that are subject to physical constraints and to the laws of physics. For example, passenger cars cannot travel at more than 200 km/h, so a high-confidence localization of a car at time t at $\hat{\ell}$ provides us with useful information about its location at time $t + 60$ s. One minute after t, the car is almost certainly within a circle of radius 3.3 km; it is almost certainly not at locations that are 5 or more kilometers away from $\hat{\ell}$.

This chapter describes a flexible technique, called *Kalman filtering and smoothing*, that exploits this insight to produce a wide range of useful estimates. The technique is widely used for location estimation, but also for a range of other applications. We start with a motivating example.

15.1 ▪ Where Will the Cannonball Land?

A cannon at coordinates $\begin{bmatrix}0 & 0 & 0\end{bmatrix}^T$ is pointed east. It shoots a ball at an elevation angle of 45 degrees. A sensor estimates the velocity of the ball coming out the barrel to be about 28.28 m/s (20 m/s in both the east and up directions). We assume for the sake of simplicity that there is no wind and that drag, the slowing-down force that movement through air generates, is negligible. We also assume that there is no wind. We want to estimate where the ball will fall.

We denote the east-west position of the ball by $x(t)$ and its height by $z(t)$. Mathematically, the initial conditions are

$$\begin{aligned} x(0) &= 0 && \text{initial horizontal position,} \\ z(0) &= 0 && \text{initial vertical position,} \\ \tfrac{dx}{dt}(0) = x'(0) &= 20 && \text{initial horizontal velocity, and} \\ \tfrac{dz}{dt}(0) = z'(0) &= 20 && \text{initial vertical velocity.} \end{aligned}$$

We can describe the forces that act on the ball using simple differential equations,

$$\begin{aligned} \tfrac{d^2x}{dt^2}(t) = x''(t) &= 0 && \text{no horizontal acceleration,} \\ \tfrac{d^2z}{dt^2}(t) = z''(t) &= -9.8 && \text{gravity causes vertical deceleration.} \end{aligned}$$

These particular equations can be solved analytically, but for more complex cases the numerical technique that we introduce next is more useful. We discretize time and represent the state of the ball at discrete times $t = 0, \Delta, 2\Delta, \ldots$ using vectors $u_i = \begin{bmatrix}x_i & z_i & x'_i & z'_i\end{bmatrix}^T$ that approximate the position and velocity of the ball at time $t = i\Delta$ for some small Δ.[10] We relate the position to the velocity using a finite-difference approximation to the derivatives,

$$x'((i-1)\Delta) \approx x'_{i-1} = \frac{x_i - x_{i-1}}{\Delta} \approx \frac{x(i\Delta) - x((i-1)\Delta)}{\Delta} ,$$

which we rearrange as $x_i \approx x_{i-1} + x'_{i-1}\Delta$. The discretized acceleration/deceleration equations are

$$-9.8 = \lim_{\Delta \to 0} \frac{z'_i - z'_{i-1}}{\Delta} \approx \frac{z'(i\delta) - z'((i-1)\delta)}{\Delta} .$$

This gives us four recurrence relations,

$$\begin{aligned}
\hat{x}_i &= \hat{x}_{i-1} + \Delta \hat{x}'_{i-1} , \\
\hat{z}_i &= \hat{z}_{i-1} + \Delta z'_{i-1} , \\
\hat{x}'_i &= \hat{x}'_{i-1} , \\
\hat{z}'_i &= \hat{z}'_{i-1} - \Delta 9.8 ,
\end{aligned}$$

which we can use to estimate the location and velocity of the ball given the initial conditions $\hat{u}_o = \begin{bmatrix} 0 & 0 & 20 & 20 \end{bmatrix}^T$; we can evolve u_i forward in time.

However, this type of model is severely restricted. The best way to obtain accurate estimates of the state vectors is to make more measurements. For example, we might be able to use a radar to measure the position of the cannonball a few times during its flight. These measurements are of the true position, not of the position that solves our recurrence relations. The measurements are also not of the solution to the differential equations, which uses simplified laws of motion that ignore wind, drag, and so on. It is better to define $\mathring{u}_i = \begin{bmatrix} \mathring{x}_i & \mathring{z}_i & \mathring{x}'_i & \mathring{z}'_i \end{bmatrix}$ as the true position and velocity at time $t = i\Delta$. This implies that our equations are not satisfied exactly, so we add error terms to compensate:

$$\begin{aligned}
\mathring{x}_i &= \mathring{x}_{i-1} + \Delta \mathring{x}'_{i-1} + \epsilon_{x,i} , \\
\mathring{z}_i &= \mathring{z}_{i-1} + \Delta \mathring{z}'_{i-1} + \epsilon_{z,i} , \\
\mathring{x}'_i &= \mathring{x}'_{i-1} + \epsilon_{x',i} , \\
\mathring{z}'_i &= \mathring{z}'_{i-1} - \Delta 9.8 + \epsilon_{z',i} .
\end{aligned}$$

Our measurements might also contain errors, so we add error terms to the initial conditions too, $\mathring{u}_o = \begin{bmatrix} 0 & 0 & 20 & 20 \end{bmatrix}^T + \epsilon_0$.

15.2 ▪ Linear Discrete Dynamical Systems

We generalize the cannonball example as follows. We assume that the instantaneous state of the system at time $i\Delta$ is specified by an n-dimensional vector $\mathring{u}_i \in \mathbb{R}^n$ and that $\mathring{u}_i$ depends *linearly* on $\mathring{u}_{i-1}$ and possibly also on a known *control vector* w_i, but that this dependence is not exact. That is, we assume that $\mathring{u}_i$ is defined by an *evolution equation* of the form

$$\mathring{u}_i = F_i \mathring{u}_{i-1} + K_i w_i + \epsilon_i^{(e)} , \tag{15.1}$$

where F_i and K_i are matrices and $\epsilon_i^{(e)}$ is a noise or error vector. The matrix F_i specifies how

[10]In this chapter, subscripts denote a time step, so x_i denotes the quantity x (a coordinate) at time $i\Delta$. Similarly, E_i is a matrix associated with time step i.

$\mathring{u}_{i-1}$ influences $\mathring{u}_i$. The control vectors describe known forces that influence the system. They can be part of the environment, like gravity in the cannonball example, or actions that we apply, like pressing the acceleration pedal in a car. It is perfectly acceptable for the $K_i w_i$ term to be zero. Because we assume that we know both K_i and w_i, we replace their product from here on by a vector $b_i^{(e)} = K_i w_i$. The noise or error vector $\epsilon_i^{(e)}$ admits estimators $\hat{u}$ that do not satisfy the evolution equations exactly; we explain later the benefits of this approach.

Our goal is to estimate $\mathring{u}$, and to do that, we need observations that contain information about $\mathring{u}$; the known vectors $b_i^{(e)}$ contain no such information. We therefore assume that we have observations $b_i^{(o)}$ derived *linearly* from $\mathring{u}_i$,

$$b_i^{(o)} = G_i \mathring{u}_i + \epsilon_i^{(o)} \ , \tag{15.2}$$

where G_i is a matrix, and $\epsilon_i^{(o)}$ represents measurement noise. This *observation equation* might be missing in some time steps i. When it is present, the dimension of $b_i^{(o)}$ is typically n or smaller, but in principle it can be higher. In the analysis of the computational complexity of the algorithms we assume that the dimension of $b_i^{(o)}$ is $O(n)$.

The k evolution equations that relate $\mathring{u}_i$ to $\mathring{u}_{i-1}$ for $i = 1, \ldots, k$ consist of nk scalar equations; the number of scalar unknowns in all the $\mathring{u}_i$'s is $n(k+1)$. In principle we could insist that the estimator $\hat{u} = \begin{bmatrix} \hat{u}_1 & \cdots & \hat{u}_k \end{bmatrix}^T$ satisfy the evolution equations exactly and that errors would occur only in observation equations. But the (widely used) formulation that we study in this chapter allows errors in the evolution equations and solves for $\hat{u}$ in the generalized least-squares sense. There are two good reasons for this choice. First, solving for $\hat{u}$ in the least-squares sense is computationally cheaper than allowing errors only in observation equations. Second, the $\epsilon_i^{(e)}$ terms can compensate for simplified and approximate evolution relations. For example, in the cannonball example $\epsilon_i^{(e)}$ can compensate for the discretization errors (the discrepancy between the continuous differential equations and the discrete difference equations) and for physical effects that we did not include in the model, such as wind and drag. If we know that our evolution equations are fairly accurate, we can express this knowledge by setting $\operatorname{cov}(\epsilon_i^{(e)})$ small.

When we solve initial-value problems in which the only observations are functions of $\mathring{u}_0$, like the cannonball example, accurate evolution equations are important because errors accumulate. But Kalman filters and smoothers are used mostly when we have observations at many time steps. In such situations, allowing for errors in the evolution equations is a useful modeling tool.

We assume that the noise vectors $\epsilon_i^{(e)}$ and $\epsilon_i^{(o)}$ all have zero mean and known covariance matrices, and that different noise vectors are uncorrelated. That is, we assume that the constraints

$$\begin{aligned}
\mathrm{E}\left(\epsilon_i^{(e)}\right) &= 0 \ , \\
\operatorname{cov}\left(\epsilon_i^{(e)}\right) = \mathrm{E}\left(\epsilon_i^{(e)}\left(\epsilon_i^{(e)}\right)^T\right) &= C_i^{(e)} \ , \\
\mathrm{E}\left(\epsilon_i^{(e)}\left(\epsilon_j^{(e)}\right)^T\right) &= 0 \text{ for } i \neq j
\end{aligned}$$

hold for some known $C_i^{(e)}$, that the same conditions apply to the $\epsilon_i^{(o)}$'s, and that for any i, j,

$$\mathrm{E}\left(\epsilon_i^{(e)}\left(\epsilon_j^{(o)}\right)^T\right) = 0 \ .$$

Before moving on, you may want to write down the evolution and observation equations for the cannonball example.

15.3 ▪ A Least-Squares Formulation

We can assemble the evolution and observation equations into one large linear equation:

$$\begin{bmatrix} b_0^{(o)} \\ b_1^{(e)} \\ b_1^{(o)} \\ b_2^{(e)} \\ \vdots \\ \vdots \\ b_k^{(e)} \\ b_k^{(o)} \end{bmatrix} = \begin{bmatrix} G_0 & & & & & \\ -F_1 & I & & & & \\ & G_1 & & & & \\ & -F_2 & I & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -F_k & I \\ & & & & & G_k \end{bmatrix} \begin{bmatrix} \mathring{u}_0 \\ \mathring{u}_1 \\ \mathring{u}_2 \\ \vdots \\ \mathring{u}_{k-1} \\ \mathring{u}_k \end{bmatrix} + \begin{bmatrix} \epsilon_0^{(o)} \\ \epsilon_1^{(e)} \\ \epsilon_1^{(o)} \\ \epsilon_2^{(e)} \\ \vdots \\ \vdots \\ \epsilon_k^{(e)} \\ \epsilon_k^{(o)} \end{bmatrix} . \qquad (15.3)$$

We denote this system by $b = Au + \epsilon$, with ϵ having expectation zero and covariance

$$C = \begin{bmatrix} C_0^{(o)} & & & & & & \\ & C_1^{(e)} & & & & & \\ & & C_1^{(o)} & & & & \\ & & & C_2^{(e)} & & & \\ & & & & \ddots & & \\ & & & & & C_k^{(e)} & \\ & & & & & & C_k^{(o)} \end{bmatrix} .$$

Under these assumptions, the best (minimum-variance) unbiased linear estimate for $\mathring{u}$ is

$$\hat{u} = \left(A^T C^{-1} A\right)^{-1} A^T C^{-1} b = \arg\min_u \|W(Au - b)\|_2^2 \ ,$$

where $W^T W = C^{-1}$. If we assume that ϵ is Gaussian, then the same $\hat{u}$ is also the maximum-likelihood estimator of $\mathring{u}$.

15.4 ▪ Rank Considerations

Our task is to estimate $k + 1$ n-dimensional state vectors, concatenated into a single $\hat{u}$ vector of dimension $(k+1)n$. Therefore, we need at least $(k+1)n$ constraints. The evolution equations are always present; sometimes the E_i's are all the same and sometime they vary, but they are always there. This gives us kn constraints. At least n constraints must come from observations. We can assume that we know the initial conditions, so $G_0 = I$; this provides just enough constraints. But any n or more constraints that raise the rank to $(k + 1)n$ ensure a unique estimate $\hat{u}$; it is not necessary to assume that we know the initial conditions. Clearly, the more observations we have, the more accurate the estimate is likely to be. Beyond that, we do not require that there be observations at every time step, or that all the G_i's have the same number of rows. Some of them can be missing altogether, including perhaps G_0.

15.5 ▪ The Paige–Saunders Algorithm

For small k, we can estimate $\hat{u}$ using a generic generalized least-squares solver. But for large k, there is a much more efficient algorithm. We start by factoring the covariance blocks, $C_i^{(e)}$ and $C_i^{(o)}$,

$$\left(W_i^{(e)}\right)^T W_i^{(e)} = \left(C_i^{(e)}\right)^{-1} ,$$

and similarly for $C_i^{(o)}$. We represent $W_i^{(e)}$ and $W_i^{(o)}$ implicitly using the Cholesky factor of $C_i^{(e)}$ and $C_i^{(o)}$ if the covariance blocks are given, or explicitly if the inverses are given. We now decorrelate the constraints by evaluating Wb and WA.

The algorithm now computes the QR factorization of $WA = QR$ in a way that preserves the sparsity of WA in R. The algorithm applies a sequence of unitary transformations that together constitute Q^T to both WA and Wb, such that for any u,

$$\|WAu - Wb\|_2 = \left\|Q^T WAu - Q^T Wb\right\|_2 = \left\|R - Q^T Wb\right\|_2 .$$

The algorithm starts with

$$WA = \begin{bmatrix}
W_0^{(o)} G_0 & & & & & \\
-W_1^{(e)} F_1 & W_1^{(e)} & & & & \\
& W_1^{(o)} G_1 & & & & \\
& -W_2^{(e)} F_2 & W_2^{(e)} I & & & \\
& & \ddots & \ddots & & \\
& & & \ddots & \ddots & \\
& & & & -W_k^{(e)} F_k & W_2^{(e)} I \\
& & & & & W_k^{(o)} G_k
\end{bmatrix} .$$

If G_0 has more than n rows, we begin with a QR factorization of $W_0^{(o)} G_0 = \tilde{Q}_0 \tilde{R}_0$ to reduce WA to

$$\begin{bmatrix}
\tilde{R}_0 & & & & & \\
0 & & & & & \\
-W_1^{(e)} F_1 & W_1^{(e)} & & & & \\
& W_1^{(o)} G_1 & & & & \\
& -W_2^{(e)} F_2 & W_2^{(e)} I & & & \\
& & \ddots & \ddots & & \\
& & & \ddots & \ddots & \\
& & & & -W_k^{(e)} F_k & W_2^{(e)} I \\
& & & & & W_k^{(o)} G_k
\end{bmatrix} ,$$

and we permute the block of zero rows to the end,

$$\begin{bmatrix}
\tilde{R}_0 & & & & & \\
W_1^{(e)} F_1 & W_1^{(e)} & & & & \\
& W_1^{(o)} G_1 & & & & \\
& W_2^{(e)} F_2 & W_2^{(e)} I & & & \\
& & \ddots & \ddots & & \\
& & & \ddots & \ddots & \\
& & & & W_k^{(e)} F_k & W_2^{(e)} I \\
& & & & & W_k^{(o)} G_k \\
0 & & & & &
\end{bmatrix} .$$

The first block row consists of n individual rows. If G_0 had n or fewer rows, we start the algorithm here, denoting $\tilde{R}_0 = W_0^{(o)} G_0$. We now compute the QR factorization of the two nonzero block rows in the first block column, printed in red above, and we apply the same transformation to the entire two block rows to obtain

$$\begin{bmatrix} R_0 & R_{0,1} & & & & \\ 0 & \bar{R}_1 & & & & \\ & W_1^{(o)} G_1 & & & & \\ & -W_2^{(e)} F_2 & W_2^{(e)} I & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -W_k^{(e)} F_k & W_2^{(e)} I \\ & & & & & W_k^{(o)} G_k \\ 0 & & & & & \end{bmatrix} .$$

The blocks $R_{0,1}$ and $\bar{R}_1$ are the result of applying the transformation to a zero block and to $W_1^{(e)}$. We now repeat the same steps on the second and third columns. The $\bar{R}_1$ block has n columns and up to n individual rows (but possibly fewer). If G_1 has any rows (that is, if we have any observations derived from $\mathring{u}_1$), we compute the QR factorization of the two blocks printed in red above and permute the zero rows to the end to obtain

$$\begin{bmatrix} R_0 & R_{0,1} & & & & \\ 0 & \tilde{R}_1 & & & & \\ & -W_2^{(e)} F_2 & W_2^{(e)} I & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -W_k^{(e)} F_k & W_2^{(e)} I \\ & & & & & W_k^{(o)} G_k \\ 0 & & & & & \\ & 0 & & & & \end{bmatrix} .$$

Again, if G_1 did not have any rows, we skip the last transformation and simply denote $\tilde{R}_1 = \bar{R}_1$. We now compute the QR factorization of the blocks in red, transform the two block rows, and obtain

$$\begin{bmatrix} R_0 & R_{0,1} & & & & \\ 0 & R_1 & R_{1,2} & & & \\ & 0 & \bar{R}_2 & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -W_k^{(e)} F_k & W_k^{(e)} I \\ & & & & & W_k^{(o)} G_k \\ 0 & & & & & \\ & 0 & & & & \end{bmatrix} .$$

At the end of the process, R will be upper triangular with the following sparsity pattern:

$$\begin{bmatrix}
R_0 & R_{0,1} & & & & \\
0 & R_1 & R_{1,2} & & & \\
 & 0 & R_2 & R_{2,3} & & \\
 & & 0 & \ddots & \ddots & \\
 & & & \ddots & R_{k-1} & R_{k-1,k} \\
 & & & & 0 & \tilde{R}_k \\
0 & & & & & \\
 & 0 & & & & \\
 & & 0 & & & \\
 & & & \ddots & & \\
 & & & & 0 & \\
 & & & & & 0
\end{bmatrix} \tag{15.4}$$

(the number of zero block rows depends on the dimension of G_0 and on the number of additional G_i's, so it may be smaller than $k+1$). Solving for $\hat{u}$ is now easy by substitution.

Let's analyze the computational complexity of the algorithm. The factorization of the covariance blocks (or their inverses) and the application W to A costs $O(n^3)$ per block row, or $O(kn^3)$ for the entire matrix. Once we have WA, we perform one or two QR factorizations per block column. The factorizations are of matrices with n columns and $O(n)$ rows. We apply the transpose of one of the Q factors to another matrix of the same dimensions (two blocks in the next block column) and apply both Q factors to the vector of transformed observations. The total cost of each step of the overall QR factorization is therefore $\Theta(n^3)$, and the total cost of the entire algorithm is $\Theta(kn^3)$ operations. A generic QR factorization would perform $\Theta((kn)^3)$ operations, a factor of k^2 more.

Solving for $\hat{u}$ requires only $\Theta(kn^2)$ operations, because there are only $\Theta(kn^2)$ nonzero elements in R.

The savings in memory are also dramatic. At any given time, each block column contains at most three nonzero blocks, so the total amount of memory required is $\Theta(kn^2)$ words of memory; a naive representation of the matrices involved requires $\Theta((kn)^2)$ words. We will see below that in some important cases, it is possible to reduce the memory requirement even further to only $\Theta(n^2)$.

15.6 ▪ Smoothing, Interpolation, Filtering, and Prediction

There are several useful ways to use the Paige–Saunders QR factorization algorithm. In the simplest case, we have observations $b_0^{(o)}, b_1^{(o)}, \ldots, b_k^{(o)}$ for every time step and solve for the generalized least-squares estimator $\hat{u} = \begin{bmatrix}\hat{u}_0 & \hat{u}_1 & \cdots & \hat{u}_k\end{bmatrix}$. The observations may or may not include the expectation of $\mathring{u}_0$. This application of the technique is called *Kalman smoothing*. It performs two tasks. The first is to smooth the observations. The evolution equations integrate information from all the observations into the estimate of every $\hat{u}_i$; this tends to smooth the track and to eliminate some of the observation noise. This effect is useful even if we can estimate each $\mathring{u}_i$ from the observations $b_i^{(o)}$. The second task is to estimate *hidden* components of the state. For example, by Kalman smoothing a sequence of GPS location estimates we can obtain estimates of the velocity and acceleration of the target, even though we never measure them directly.

If there are no observations in some time steps ($b_i^{(o)}$ and G_i have no rows), the corresponding state estimates $\hat{u}_i$ are *interpolated.* The algorithm uses observations from previous and subsequent time steps together with the evolution equations to fill in the gaps. There are two common reasons for not having observations at all time steps. One is a transient failure, perhaps a sensor that malfunctioned or was turned off to save power. It is often useful to fill in these gaps and to output estimates $\hat{u}_i$ at regular time intervals; this tends to simplify additional processing of the track. Another situation in which there are no observations at many time steps is when the sensor produces an observation vector every $m\Delta$ seconds but we model the system every Δ seconds to better approximate the continuous dynamical system.

A more surprising way to use the algorithm is to maintain only two block columns of WA at any given time, as well as the corresponding entries of Wb. Suppose that we completed the factorization of WA up to step k, so that the reduced matrix is the $(k+1)n$-column R factor given in (15.4). We now wish to estimate $\hat{u}_{k+1}$; perhaps we already have $b_{k+1}^{(o)}$, and perhaps we do not. We extend the reduced WA by one block column and two block rows (only one if there are no observations for step $k+1$):

$$\begin{bmatrix}
R_0 & R_{0,1} & & & & & \\
0 & R_1 & R_{1,2} & & & & \\
 & 0 & R_2 & R_{2,3} & & & \\
 & & 0 & \ddots & \ddots & & \\
 & & & \ddots & R_{k-1} & R_{k-1,k} & \\
 & & & & 0 & \tilde{R}_k & \\
 & & & & & -W_{k+1}^{(e)}E_{k+1} & W_{k+1}^{(e)}I \\
 & & & & & & W_{k+1}^{(o)}G_{k+1} \\
0 & & & & & & \\
 & 0 & & & & & \\
 & & 0 & & & & \\
 & & & \ddots & & & \\
 & & & & 0 & & \\
 & & & & & 0 &
\end{bmatrix}.$$

We continue with the factorization and produce R_k, $R_{k,k+1}$, and $\tilde{R}_{k+1}$. To do that, we only need $\tilde{R}_k$; we don't need anything else from R. The same is true for the reduced observations; we only need the last n elements in that vector. This allows us to discard everything but the last blocks of R and of the reduced observations. Clearly, if we do not keep all the blocks of R and all the reduced observations, we cannot produce an estimate $\hat{u} = \begin{bmatrix}\hat{u}_0 & \hat{u}_1 & \cdots & \hat{u}_k & \hat{u}_{k+1}\end{bmatrix}$, because solving the triangular system for $\hat{u}$ is done by back substitution, from $\hat{u}_{k+1}$ to $\hat{u}_0$. But we can solve for $\hat{u}_{k+1}$. The overall algorithm produces a sequence of estimates of state vectors that are each based on past and present observations only, not on future observations. That is, the estimate for $\mathring{u}_i$, which we denote in this case by $\hat{u}_{i|i}$, is based on $b_0^{(o)}$ up to $b_i^{(o)}$, but not on $b_{i+1}^{(o)}$ or later. This use of the algorithm is called Kalman *filtering*, as opposed to *smoothing*.

Clearly, nothing prevents us from extending the system to $k+2$ before we have $b_{k+1}^{(o)}$; we simply treat G_{k+1} as having zero rows. This produces a *prediction* $\hat{u}_{k+1|k}$. When we get $b_{k+1}^{(o)}$ later, we eliminate the $W_{k+1}^{(o)}G_{k+1}$ to produce $\tilde{R}_{k+1}$, and we can produce $\hat{u}_{k+1|k+1}$ if we need it. We can *extrapolate* in this way as far as we need; the only overhead is to keep the part of the reduced matrix and the reduced observations that might change as new observations come in. Prediction is useful if we need to rendezvous with the target, or if we need to find control inputs that will bring it close to a desired state.

15.7 ▪ Computing the Variance of the Estimates

We know from (6.1) that the covariance of the generalized least-squares estimator $\hat{u}$ is

$$\operatorname{cov}(\hat{u}) = \left(A^T C^{-1} A\right)^{-1} \ .$$

We have $C^{-1} = W^T W$ and $WA = QR$, so by Equation (6.3) $\operatorname{cov}(\hat{u}) = R^{-1}R^{-T}$. In our case $R^T R$ is a sparse matrix (all but three of its block diagonals are zero, because R is block bidiagonal), but its inverse is not. This is fairly obvious. If a particular $\hat{u}_i$ deviates significantly from its mean, then $\hat{u}_{i+1}$ usually tends to deviate in the same direction, because the evolution equation keeps them close. Therefore, for large k, the full covariance matrix $\operatorname{cov}(\hat{u})$ is difficult to store and expensive to compute.

But it turns out that we can compute the covariance matrices of all the $\hat{u}_i$'s, which are the diagonal blocks of $\operatorname{cov}(\hat{u})$, without incurring significant costs in either memory or computation. That is, we will compute

$$\operatorname{cov}(\hat{u}_i) = \left(\left(R^T R\right)^{-1}\right)_{i,i} = \left(R^{-1}R^{-T}\right)_{i,i} \ ,$$

where the index i refers to a block, not to a single row and a single column. The technique is a bit tricky, but the case of $i = k$ (the last state) is easy and shows the way forward. The inverse R^{-1} is upper triangular (the inverse of any upper-triangular matrix is upper triangular), so the only nonzero block in block row k of R^{-1} is the k, k block. Therefore,

$$\begin{aligned}\operatorname{cov}(\hat{u}_k) &= \left(R^{-1}R^{-T}\right)_{k,k} \\ &= \sum_j \left(R^{-1}\right)_{k,j} \left(R^{-T}\right)_{j,k} \\ &= \left(R^{-1}\right)_{k,k} \left(R^{-T}\right)_{k,k} \ .\end{aligned}$$

Also, because $R^{-1}R = I$,

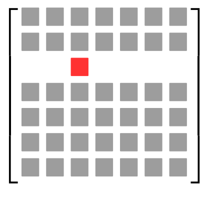
$$\begin{aligned}I &= \left(R^{-1}R\right)_{k,k} \\ &= \sum_j \left(R^{-1}\right)_{k,j} \left(R\right)_{j,k} \\ &= \left(R^{-1}\right)_{k,k} \left(R\right)_{k,k} \ .\end{aligned}$$

Therefore, $(R^{-1})_{k,k} = (R_{k,k})^{-1}$, so

$$\operatorname{cov}(\hat{u}_k) = \left(R^{-1}R^{-T}\right)_{k,k} = \left(R_{k.k}\right)^{-1} \left(R_{k,k}\right)^{-T} \ .$$

To evaluate $\operatorname{cov}(\hat{u}_i)$ for $i < k$, we need to generalize this result. We begin with a lemma that shows the structures of a diagonal block of the inverse when the corresponding row in the matrix has only one nonzero block, the diagonal block.

Lemma 15.1. *Let S be a square invertible block matrix in which in block row i, only the diagonal block contains nonzeros,*

$S =$  .

In the example above $i = 3$. Then $S_{i,i}$, depicted in red above, must be nonsingular. Furthermore, block row i in S^{-1} must have the same structure, zero outside block column i, and this nonzero block satisfies $(S^{-1})_{i,i} = (S_{i,i})^{-1}$.

Proof. The fact that $S_{i,i}$ is invertible is easy; otherwise S would be rank deficient, contradicting the assumption that it is invertible.

To prove the claims concerning the structure of block row i of S^{-1}, we observe that

$$\begin{aligned} I = \left(SS^{-1}\right)_{i,i} &= \sum_k S_{i,k}\left(S^{-1}\right)_{k,i} \\ &= S_{i,i}\left(S^{-1}\right)_{i,i} + \sum_{k\neq i} 0\left(S^{-1}\right)_{k,i} \\ &= S_{i,i}\left(S^{-1}\right)_{i,i} \ , \end{aligned}$$

so $(S^{-1})_{i,i} = (S_{i,i})^{-1}$. To prove that $(S^{-1})_{i,j}$ for $j \neq i$ observe that

$$\begin{aligned} 0 = \left(SS^{-1}\right)_{i,j} &= \sum_k S_{i,k}\left(S^{-1}\right)_{k,j} \\ &= S_{i,i}\left(S^{-1}\right)_{i,j} + \sum_{k\neq i} 0\left(S^{-1}\right)_{k,j} \\ &= S_{i,i}\left(S^{-1}\right)_{i,j} \ , \end{aligned}$$

so $(S^{-1})_{i,j} = (S_{i,i})^{-1}0 = 0$. $\square$

As in the block upper-triangular case, this structure of S^{-1} gives a simple expression for

$$\begin{aligned} \left(S^{-1}S^{-T}\right)_{i,i} &= \sum_k \left(S^{-1}\right)_{i,k}\left(S^{-T}\right)_{k,i} \\ &= \left(S^{-1}\right)_{i,i}\left(S^{-T}\right)_{i,i} + \sum_{k\neq i} 0\left(S^{-T}\right)_{k,i} \\ &= \left(S_{i,i}\right)^{-1}\left(S_{i,i}\right)^{-T} \ . \end{aligned}$$

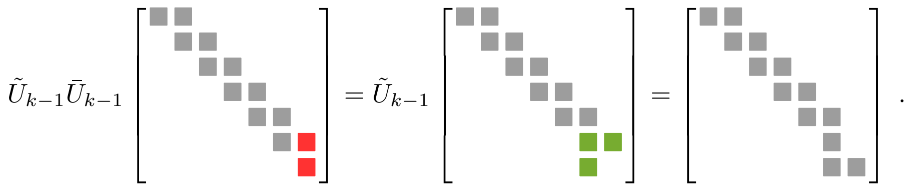
We now use this insight to develop an efficient algorithm that computes the covariance matrices of all the $\hat{u}_i$'s. The strategy is to transform R using a unitary matrix U so that $S = UR$ has the structure that we used in the lemma, with only the diagonal block in block row i being nonzero. The unitary transformation preserves the overall covariance matrix:

$$\begin{aligned} (UR)^{-1}(UR)^{-T} &= (UR)^{-1}\left((UR)^T\right)^{-1} \\ &= R^{-1}U^{-1}\left(R^TU^T\right)^{-1} \\ &= R^{-1}U^{-1}U^{-T}R^{-T} \\ &= R^{-1}\left(U^TU\right)^{-1}R^{-T} \\ &= R^{-1}R^{-T} \ . \end{aligned}$$

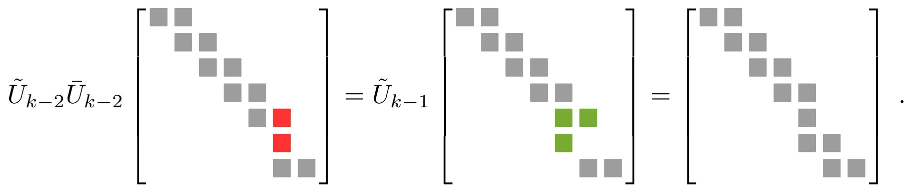
This allow us to extract $\operatorname{cov}(\hat{u}_i)$ from

$$\operatorname{cov}(\hat{u}_i) = \left(R^{-1}R^{-T}\right)_{i,i} = \left((UR)^{-1}(UR)^{-T}\right)_{i,i} = (S_{i,i})^{-1}(S_{i,i})^{-T} \ .$$

We now show how to construct a sequence of unitary matrices U that bring UR to a form in which the only nonzero in block row $k-1$ is the diagonal block, then row $k-2$ is brought into the required structure, then row $k-3$, and so on. Each step consists of two phases. The first phase computes the QR factorization of a two-block submatrix, marked in red below, and applies the orthonormal factor to two block rows of the entire matrix. The second phase swaps two block rows, marked in green:



This brought block row $k-1$ to the required form. The next steps are similar. To construct $(R^{-1}R^{-T})_{k-1,k-1}$ we again apply two orthonormal transformations:



To compute the covariance matrices of the $\hat{u}_i$'s, we need to store all the nonzero blocks of R, because this process is done backwards, just like solving for the $\hat{u}_i$. To compute only $\text{cov}(\hat{u}_k)$ we only need the last block of R. In general, we can compute all the $\text{cov}(\hat{u}_{i|i})$'s without storing all of R.

The cost to compute all the covariance matrices is again $\Theta(kn^3)$ operations.

15.8 ▪ Notes

The invention of the Kalman filter by Rudolph E. K’alm’an in the late 1950s [38] was a major breakthrough in tracking algorithms. Duncan and Horn showed that the filter (and smoother) can be interpreted as a linear least-squares problem [28]. The algorithm that we present in this book is a variant of an algorithm proposed by Paige and Saunders [50]; this algorithm is more numerically stable than most other implementations and also conceptually much simpler, at least for readers with a good understanding of the QR factorization. The original Paige–Saunders did not apply the QR factorization to entire blocks, but rather specified a sequence of Givens rotations. Our variant is much simpler to explain and to implement; it does perform a few more arithmetic operations, but due to the performance of today’s QR factorization codes, it is not necessarily slower than an implementation of the original algorithm.

The Kalman filter is presented in numerous books (always in a form that is less numerically stable than the one presented here, and arguably also more complicated). There are even entire books devoted entirely or mostly to Kalman filtering and smoothing [11, 32].

The Kalman filter can be extended to nonlinear problems in various ways. The most widely used extension is appropriately called the *extended Kalman filter*. It is intended for nonlinear observation equations. It works by using the evolution equations to predict u_i from $\hat{u}_{i-1}$, linearizing the observation equations at the prediction u_i, and setting $\hat{u}_i$ to be the minimizer of the resulting linear least-squares problem. In other words, it uses the evolution equations to predict u_i and then performs one step of the Gauss–Newton algorithm on the observations to improve that prediction.

15.9 ▪ Problems

15.9.1 ▪ Using a Kalman Filter and Smoother

The goal of this problem is to create and solve a Kalman filter model of the cannonball problem. You will use a MATLAB implementation of the Kalman filter and smoother. The implementation, called PaigeSaundersKalman, maintains and evolves an object that contains all the data associated with a filter or smoother. The programming interface of the object is as follows:

```
f = PaigeSaundersKalman;      % create an empty
                              % Kalman filter/smoother

% provide observations for this step in one of 4 ways
f.observe(Gi,bei,   Cei,'c'); % covariance matrix given
f.observe(Gi,bei,invCei,'i'); % or inv(Cei)
f.observe(Gi,bei,   Wei,'v'); % or chol(Cei)
f.observe(Gi,bei,invWei,'s'); % or inv(chol(Cei))
f.observe();                  % or indicate that there
                              % are no observations

% provide evolution data for this step
f.evolve (Fi,boi,   Coi,'c'); % or use 'i', 's', 'v' forms
```

The observe and evolve methods can accept $C_i^{(o)}$ and $C_i^{(e)}$ in four forms: explicitly (c); the inverse of the covariance matrices (i); W such that $W^TW = C^{-1}$ (v); or the inverse of the W factor (s). To use the implementation, you create the object, call observe to provide the observations of the initial state (if there are none, call observe with no arguments), and then call evolve once and observe once for every step.

The method smooth (with no arguments) performs the backward substitution and returns the vector of smoothed states $\hat{u}$ (as a single vector with all the states concatenated). The function [covs,stddevs]=f.covariances() returns two cell arrays: one with $\mathrm{cov}(\hat{u}_i)$ and the other with vectors of the standard deviations of $\hat{u}_i$.

1. Simulate the cannonball example using state vectors $u_i = \begin{bmatrix} x_i & z_i & x_i' & z_i' \end{bmatrix}^T$ that model the horizontal distance, height, horizontal velocity, and vertical velocity. Use time steps of $\Delta = 0.1$ s and specify a standard deviation of 10^{-6} for observations of all the components of $\mathring{u}_o = \begin{bmatrix} 0 & 0 & 20 & 20 \end{bmatrix}^T$. The standard deviations of the evolution equations should be $\begin{bmatrix} 10^{-6} & 10^{-6} & 0.1 & 0.1 \end{bmatrix}^T$ (most of the uncertainty concerns the velocity). Plot the trajectory and the evolution of both the standard deviations and the velocity. When and where will the ball hit the $z = 0$ plane?

2. Simulate the cannonball again, but this time without an observation of the initial state. Instead, specify the position (but not the velocity) at time steps 4, 5, and 6; these might be obtained from a radar, for example. Take the positions from the corresponding steps of the simulation in part 1 of the problem. Specify standard deviations of 0.1 for these observations.

15.9.2 ▪ Accurate Event Timing Using a Pulse-per-Second Signal

A computer uses an internal clock to time various events (e.g., the arrival time of signals from transmitters in a location-estimation system). The clock is not accurate, but it is stable. That is, it may run a little slower or a little faster than real time, and it may have an offset relative to real time, but its rate changes very slowly.

One of the events that the computer observes is a periodic pulse that a GNSS receiver emits every second (most GNSS receivers emit such a signal, called a pulse-per-second or PPS signal). The timing of the rising edge of the ith pulse in a day is a random variable with expectation i and standard deviation of 100 ns.

We wish to use these pulses to create a data structure that allows us to correct time stamps of other events. That is, we want to estimate the precise absolute time t at which the computer observed some event, given that its internal clock showed $\bar{t}$ when the event occurred. We will use a Kalman filter to model the state of the internal clock. We denote by $\bar{t_i}$ the time shown by the internal clock at the ith rising edge of the PPS signal.

1. We define the state $\mathring{u}_i = \begin{bmatrix} \mathring{o}_i & \mathring{r}_i \end{bmatrix}^T$ of the internal clock in terms of its offset at second i and its rate during the second starting at $i-1$ and ending at i. We consider the rate to be the actual advance in the value showed by the clock during one second. Write the evolution and observation equations for this model and suggest how to set the covariance matrices associated with them.

2. Explain how to set the covariance matrix of error in the observation equation.

3. We did not give enough information to set the covariance matrix of the evolution equations. Suggest a way to decide experimentally whether you set it too high or too low.

4. Suppose that we observed the PPS at seconds $0, 1, 2, \ldots, k$. Explain how to process the data in order to convert a time stamp $\bar{t}$ of the internal clock to an estimate of the absolute time at which the event occurred.

15.9.3 ▪ Kalman Filters for Vehicular Navigation

Navigation systems in cars need estimates of the current location in order to decide on the next action required to reach the destination. In this problem we explore the use of Kalman filters to estimate the current location. Our basic observations are GNSS location estimates, at a rate of one per second (when the sky is not obscured by a tunnel or another structure). The navigation system can use the raw GNSS locations, but we can do better.

1. Develop evolution and observation equations for a model in which the state had four components: two Cartesian metric coordinates of the current location, and two components of the velocity (speed). We assume that the x axis points east and that the y axis points north. Assume that the GNSS does not produce speed or direction estimates, so the evolution equations should express the assumption that the car travels at constant speed. Also assume that the standard deviation of the GNSS coordinate estimates is 10 m. Propose a way to set the covariance matrices of the error or noise terms in the equation.

2. Extend your model to also include acceleration. This adds two parameters to each state vector: acceleration in the x and y directions.

3. Suppose that we also have access to the speedometer of the car, so we can observe its speed. Write the observation equation and explain whether it makes more sense to add it to the four-dimensional state model or to the six-dimensional state model.

4. This observation equation is not linear. Suggest a way to address this difficulty (there are at least two different ways).

15.9.4 ▪ Second- and Higher-Order Recurrences

The evolution equation used in this chapter is a first-order recurrence in the sense that $\mathring{u}_i$ depends on $\mathring{u}_{i-1}$ but not on earlier state vectors. We also assumed that the observations $b_i^{(e)}$ depend on $\mathring{u}_i$ but not on earlier state vectors. However, it is possible to extend the algorithms to second-order recurrences or higher.

1. Show how to include in A blocks that describe explicitly second-order relations. Specify the second-order evolution and observation equations, and show how to incorporate them into the global matrix A.

2. Show how to compute the R factor of a second-order A efficiently, and show the sparsity structure of the R factor.