DAN SIMON

Kalman Filtering

Originally developed for use in spacecraft navigation, the Kalman filter turns out to be useful for many applications. It is mainly used to estimate system states that can only be observed indirectly or inaccurately by the system itself.

Filtering is desirable in many situations in engineering and embedded systems. For example, radio communication signals are corrupted with noise. A good filtering algorithm can remove the noise from electromagnetic signals while retaining the useful information. Another example is power supply voltages. Uninterruptible power supplies are devices that filter line voltages in order to smooth out undesirable fluctuations that might otherwise shorten the lifespan of electrical devices such as computers and printers.

The Kalman filter is a tool that can estimate the variables of a wide range of processes. In mathematical terms we would say that a Kalman filter estimates the states of a linear system. The Kalman filter not only works well in practice, but it is theoretically attractive because it can be shown that of all possible filters, it is the one that minimizes the variance of the estimation error. Kalman filters are often implemented in embedded control systems because in order to control a process, you first need an accurate estimate of the process variables.

This article will tell you the basic concepts that you need to know to design and implement a Kalman filter. I will introduce the Kalman filter algorithm and we'll look at the use of this filter to solve a vehicle navigation problem. In order to control the position of an automated vehicle, we first must have a reliable estimate of the vehicle's present position. Kalman filtering provides a tool for obtaining that reliable estimate.

Linear systems

In order to use a Kalman filter to remove noise from a signal, the process that we are measuring must be able to be described by a linear system. Many physical processes, such as a vehicle driving along a road, a satellite orbiting the earth, a motor shaft driven by winding currents, or a sinusoidal

The Kalman filter not only works well in practice, but it is theoretically attractive because it can be shown that of all possible filters, it is the one that minimizes the variance of the estimation error.

radio-frequency carrier signal, can be approximated as linear systems. A linear system is simply a process that can be described by the following two equations:

State equation:

$$x_{k+1} = Ax_k + Bu_k + w_k$$

Output equation:

$$y_k = Cx_k + z_k$$

In the above equations *A*, *B*, and *C* are matrices; *k* is the time index; *x* is called the state of the system; *u* is a known input to the system; *y* is the measured output; and *w* and *z* are the noise. The variable *w* is called the process noise, and *z* is called the measurement noise. Each of these quantities are (in general) vectors and therefore contain more than one element. The vector *x* contains all of the information about the present state of the system, but we cannot measure *x* directly. Instead we measure *y*, which is a function of *x* that is corrupted by the noise *z*. We can use *y* to help us obtain an estimate of *x*, but we cannot necessarily take the information from *y* at face value because it is corrupted by noise. The measurement is like a politician. We can use the information that it presents to a certain extent, but we cannot afford to grant it our total trust.

For example, suppose we want to model a vehicle going in a straight line. We can say that the state consists of the vehicle position *p* and velocity *v*. The input *u* is the commanded acceleration and the output *y* is the measured position. Let's say that we are able to change the acceleration and measure the position every *T* seconds. In this case, elementary laws of physics say that the velocity *v* will be governed by the following equation:

$$v_{k+1} = v_k + Tu_k$$

That is, the velocity one time-step from now (*T* seconds from now) will be equal to the present velocity plus the commanded acceleration multiplied by *T*. But the previous equation does not give a precise value for v_{k+1}. Instead, the velocity will be perturbed by noise due to gusts of wind, potholes, and other unfortunate realities. The velocity noise is a random variable that changes with time. So a more realistic equation for *v* would be:

$$v_{k+1} = v_k + Tu_k + \tilde{v_k}$$

where $\tilde{v_k}$ is the velocity noise. A similar equation can be derived for the position *p*:

$$p_{k+1} = p_k + Tv_k + \frac{1}{2}T^2u_k + \tilde{p_k}$$

where $\tilde{p_k}$ is the position noise. Now we can define a state vector *x* that consists of position and velocity:

$$x_k = \begin{bmatrix} p_k \\ v_k \end{bmatrix}$$

Finally, knowing that the measured output is equal to the position, we can write our linear system equations as follows:

$$x_{k+1} = \begin{bmatrix} 1 & T \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} T^2/2 \\ T \end{bmatrix} u_k + w_k$$

$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} x_k + z_k$$

z_k is the measurement noise due to such things as instrumentation errors. If we want to control the vehicle with some sort of feedback system, we need an accurate estimate of the position *p* and the velocity *v*. In other words, we need a way to estimate the state *x*. This is where the Kalman filter comes in.

The Kalman filter theory and algorithm

Suppose we have a linear system model as described previously. We want to use the available measurements *y* to estimate the state of the system *x*. We know how the system behaves according to the state equation, and we have measurements of the position, so how can we determine the best estimate of the state *x*? We want an estimator that gives an accurate estimate of the true state even though we cannot directly measure it. What criteria should our estimator satisfy? Two obvious requirements come to mind.

First, we want the average value of our state estimate to be equal to the average value of the true state. That is, we don't want our estimate to be biased one way or another. Mathematically, we would say that the expected value of the estimate should be equal to the expected value of the state.

Second, we want a state estimate that varies from the true state as little as possible. That is, not only do we want the average of the state estimate to be equal to the average of the true state, but we also want an estimator that results in the smallest possible variation of the state estimate. Mathematically, we would say that we want to find the estimator with the smallest possible error variance.

It so happens that the Kalman filter is the estimator that satisfies these two criteria. But the Kalman filter solution does not apply unless we can satisfy certain assumptions about the noise that affects our system. Remember from our system model that *w* is the process noise and *z* is the measurement noise. We have to assume that the average value of *w* is zero and the

HANNAH BLAIR

FIGURE 1 Vehicle position (true, measured, and estimated)

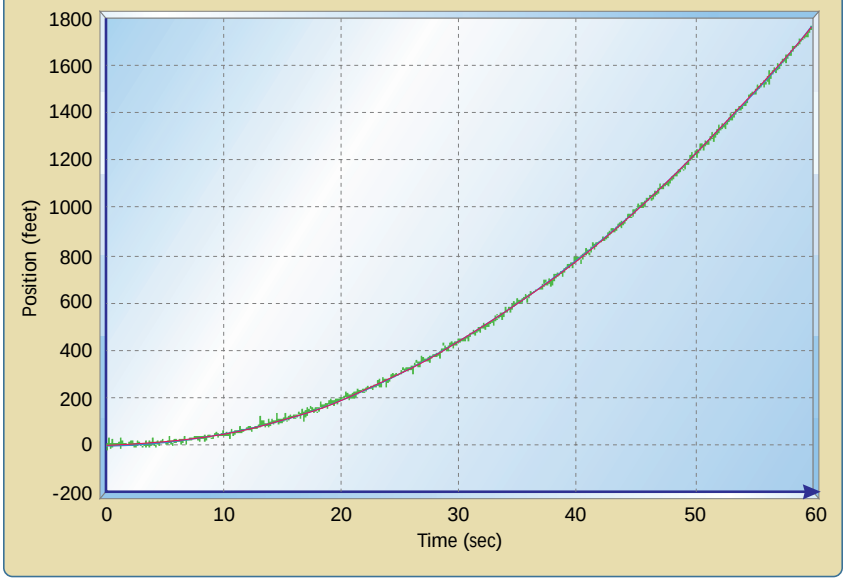
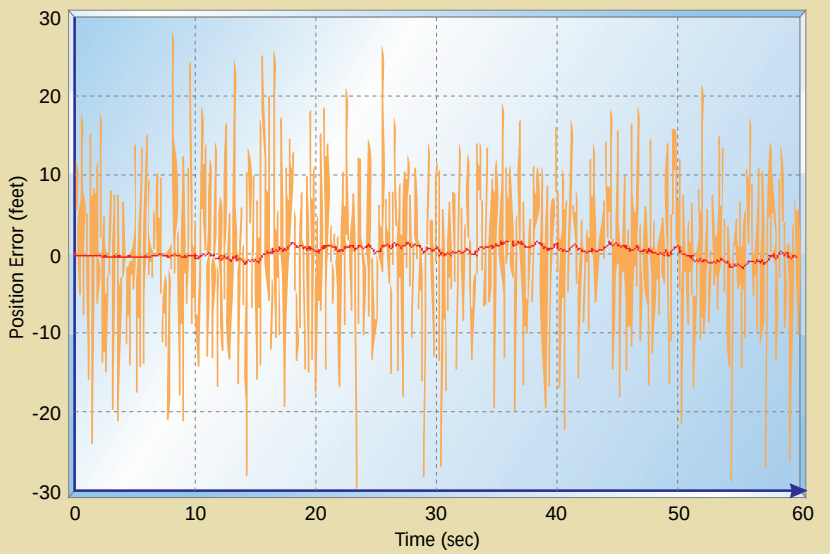


FIGURE 2 Position measurement error and position estimation error



average value of z is zero. We have to further assume that no correlation exists between w and z. That is, at any time k, w_k, and z_k are independent random variables. Then the noise covariance matrices S_w and S_z are defined as:

Process noise covariance

$$S_w = E\left(w_k w_k^T\right)$$

Measurement noise covariance

$$S_z = E\left(z_k z_k^T\right)$$

where w^T and z^T indicate the transpose of the w and z random noise vectors, and E(·) means the expected value.

Now we are finally in a position to look at the Kalman filter equations. There are many alternative but equivalent ways to express the equations. One of the formulations is given as follows:

$$K_k = AP_kC^T\left(CP_kC^T + S_z\right)^{-1}$$
$$\hat{x}_{k+1} = \left(A\hat{x}_k + Bu_k\right) + K_k\left(y_{k+1} - C\hat{x}_k\right)$$
$$P_{k+1} = AP_kA^T + S_w - AP_kC^TS_z^{-1}CP_kA^T$$

That's the Kalman filter. It consists of three equations, each involving matrix manipulation. In the above equations, a *–1* superscript indicates matrix inversion and a *T* superscript indicates matrix transposition. The *K* matrix is called the Kalman gain, and the *P* matrix is called the estimation error covariance.

The state estimate ($\hat{x}$) equation is fairly intuitive. The first term used to derive the state estimate at time $k + 1$ is just A times the state estimate at time k, plus B times the known input at time k. This would be the state estimate if we didn't have a measurement. In other words, the state estimate would propagate in time just like the state vector in the system model. The second term in the $\hat{x}$ equation is called the *correction term* and it represents the amount by which to correct the propagated state estimate due to our measurement.

Inspection of the K equation shows that if the measurement noise is large, S_z will be large, so K will be small and we won't give much credibility to the measurement y when computing the next $\hat{x}$. On the other hand, if the measurement noise is small, S_z will be small, so K will be large and we will give a lot of credibility to the measurement when computing the next $\hat{x}$.

Vehicle navigation

Now consider the vehicle navigation problem that we looked at earlier. A vehicle is traveling along a road. The position is measured with an error of 10 feet (one standard deviation). The commanded acceleration is a constant 1 foot/sec^2. The acceleration noise is 0.2 feet/sec^2 (one standard deviation). The position is measured 10 times per second (T = 0.1). How can

we best estimate the position of the moving vehicle? In view of the large measurement noise, we can surely do better than just taking the measurements at face value.

Since $T = 0.1$, the linear model that represents our system can be derived from the system model presented earlier in this article as follows:

$$x_{k+1} = \begin{bmatrix} 1 & 0.1 \\ 0 & 1 \end{bmatrix} x_k + \begin{bmatrix} 0.005 \\ 0.1 \end{bmatrix} u_k + w_k$$

$$y_k = \begin{bmatrix} 1 & 0 \end{bmatrix} x_k + z_k$$

Because the standard deviation of the measurement noise is 10 feet, the S_z matrix is simply equal to 100.

Now we need to derive the S_w matrix. Since the position is proportional to 0.005 times the acceleration, and the acceleration noise is 0.2 feet/sec^2, the variance of the position noise is $(0.005)^2 \cdot (0.2)^2 = 10^{-6}$. Similarly, since the velocity is proportional to 0.1 times the acceleration, the variance of the velocity noise is $(0.1)^2 \cdot (0.2)^2 = 4 \cdot 10^{-4}$. Finally, the covariance of the position noise and velocity noise is equal to the standard deviation of the position noise times the standard deviation of the velocity noise, which can be calculated as $(0.005 \cdot 0.2) \cdot (0.1 \cdot 0.2) = 2 \cdot 10^{-5}$. We can combine all of these calculations to obtain the following matrix for S_w:

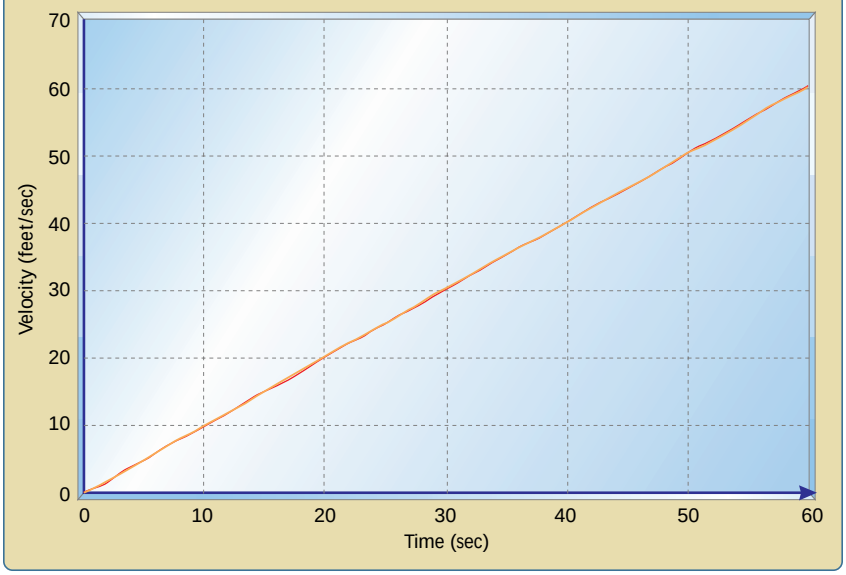
$$S_w = E\left(xx^T\right) = E\left(\begin{bmatrix} p \\ v \end{bmatrix} \begin{bmatrix} p & v \end{bmatrix}\right)$$

$$= E\begin{bmatrix} p^2 & pv \\ vp & v^2 \end{bmatrix} = \begin{bmatrix} 10^{-6} & 2\times10^{-5} \\ 2\times10^{-5} & 4\times10^{-4} \end{bmatrix}$$

Now we just initialize $\hat{x}_0$ as our best initial estimate of position and velocity, and we initialize P_0 as the uncertainty in our initial estimate. Then we execute the Kalman filter equations once per time step and we are off and running.

I simulated the Kalman filter for this problem using Matlab. The results are shown in the accompanying figures. Figure 1 shows the true position of the vehicle, the measured position, and the estimated position. The two smooth curves are the true position and the estimated position, and they

FIGURE 3 Velocity (true and estimated)



LISTING 1 Kalman filter simulation

```
function kalman(duration, dt)

% function kalman(duration, dt)
%
% Kalman filter simulation for a vehicle travelling along a road.
% INPUTS
%   duration = length of simulation (seconds)
%   dt = step size (seconds)

measnoise = 10; % position measurement noise (feet)
accelnoise = 0.2; % acceleration noise (feet/sec^2)

a = [1 dt; 0 1]; % transition matrix
b = [dt^2/2; dt]; % input matrix
c = [1 0]; % measurement matrix
x = [0; 0]; % initial state vector
xhat = x; % initial state estimate

Sz = measnoise^2; % measurement error covariance
Sw = accelnoise^2 * [dt^4/4 dt^3/2; dt^3/2 dt^2]; % process noise cov
P = Sw; % initial estimation covariance

% Initialize arrays for later plotting.
pos = []; % true position array
poshat = []; % estimated position array
posmeas = []; % measured position array
vel = []; % true velocity array
velhat = []; % estimated velocity array

for t = 0 : dt: duration,
     % Use a constant commanded acceleration of 1 foot/sec^2.
   u = 1;
   % Simulate the linear system.
   ProcessNoise = accelnoise * [(dt^2/2)*randn; dt*randn];
   x = a * x + b * u + ProcessNoise;
   % Simulate the noisy measurement
   MeasNoise = measnoise * randn;
   y = c * x + MeasNoise;
   % Extrapolate the most recent state estimate to the present time.
   xhat = a * xhat + b * u;
   % Form the Innovation vector.
   Inn = y - c * xhat;
   % Compute the covariance of the Innovation.
   s = c * P * c' + Sz;
   % Form the Kalman Gain matrix.
   K = a * P * c' * inv(s);
   % Update the state estimate.
   xhat = xhat + K * Inn;
   % Compute the covariance of the estimation error.
```

Listing 1 continued on p. 78.

are almost too close to distinguish from one another. The noisy-looking curve is the measured position.

Figure 2 shows the error between the true position and the measured position, and the error between the true position and the Kalman filter's estimated position. The measurement error has a standard deviation of about 10 feet, with occasional spikes up to 30 feet (3 sigma). The estimated position error stays within about two feet.

Figure 3 shows a bonus that we get from the Kalman filter. Since the vehicle velocity is part of the state *x*, we get a velocity estimate along with the position estimate. Figure 4 shows the error between the true velocity and the Kalman filter's estimated velocity.

The Matlab program that I used to generate these results is shown in Listing 1. Don't worry if you don't know Matlab—it's an easy-to-read language, almost like pseudocode, but with built-in matrix operations. If you use Matlab to run the program you will get different results every time because of the random noise that is simulated, but the results will be similar to the figures shown here.

Practical issues and extensions

The basic ideas of Kalman filtering are straightforward, but the filter equations rely heavily on matrix algebra. Listing 2 shows the Kalman filter update equations in C. The matrix algebra listings referenced in Listing 2 can be found at *www.embedded.com/code.html*.

These listings are very general and, if the problem is small enough, could probably be simplified considerably. For example, the inverse of the 2-by-2 matrix:

$$D = \begin{bmatrix} d_1 & d_2 \\ d_3 & d_4 \end{bmatrix}$$

is equal to:

$$D^{-1} = \frac{1}{d_1 d_4 - d_2 d_3} \begin{bmatrix} d_4 & -d_2 \\ -d_3 & d_1 \end{bmatrix}$$

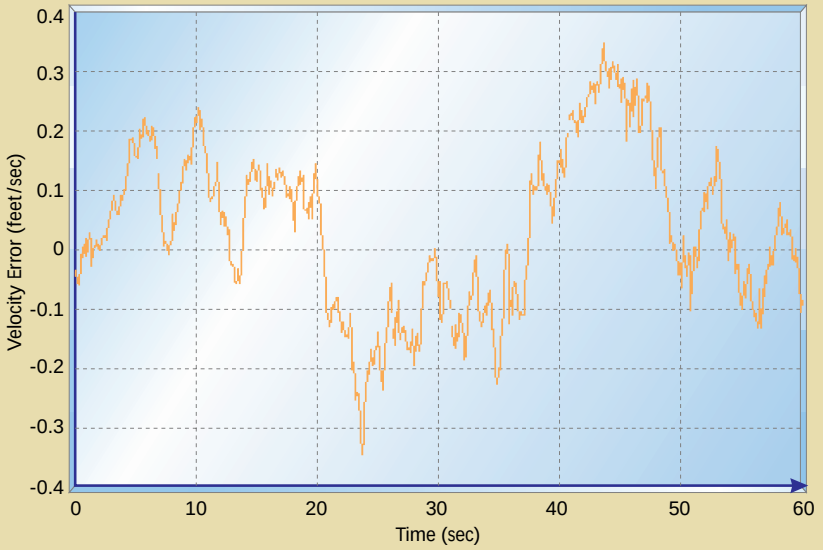
So if you need to invert a 2-by-2 matrix you can use the above equation. Some additional C code for matrix manipulation and Kalman filtering can be found at *http://wad.www.media.mit.edu/people/wad/mas864/proj_src.html*.

Systems with more than three states could exceed your budget for program size and computational effort. The computational effort associated with matrix inversion is proportional to n^3 (where n is the size of the matrix). This means that if the number of states in the Kalman filter doubles, the computational effort increases by a factor of eight. It's not too hard to see how you could run out of throughput pretty quickly for a moderately sized Kalman filter. But never fear! The so-called "steady state Kalman filter" can greatly reduce the computational expense while still giving good estimation performance. In the steady state Kalman filter the matrices K_k and P_k are constant, so they can be hard-coded as constants, and the only Kalman filter equation that needs to be implemented in real time is the $\hat{x}$ equation, which consists of simple multiplies and addition steps (or multiply and accumulates if you're using a DSP).

We have discussed state estimation for linear systems. But what if we want to estimate the states of a nonlinear system? As a matter of fact, almost all real engineering processes are nonlinear. Some can be approximated by linear systems but some cannot. This was recognized early in the history of Kalman filters and led to the development of the "extended Kalman filter," which is simply an extension of linear Kalman filter theory to nonlinear systems.

Up to this point we have talked about estimating the state one step at a time as we obtain measurements. But what if we want to estimate the state as a function of time after we already have the entire time-history of measurements? For example, what if we want to reconstruct the trajectory of our vehicle after the fact? Then it seems that we could do better than the Kalman filter because to estimate the state at time k we could use measurements not only up to and including time k, but also after time k. The Kalman filter can be modified for this problem to obtain the so-called Kalman smoother.

FIGURE 4 Velocity estimation error



LISTING 1, continued Kalman filter simulation

```
    P = a * P * a' - a * P * c' * inv(s) * c * P * a' + Sw;
    % Save some parameters for plotting later.
    pos = [pos; x(1)];
    posmeas = [posmeas; y];
    poshat = [poshat; xhat(1)];
    vel = [vel; x(2)];
    velhat = [velhat; xhat(2)];
end

% Plot the results
close all;
t = 0 : dt : duration;

figure;
plot(t,pos, t,posmeas, t,poshat);
grid;
xlabel('Time (sec)');
ylabel('Position (feet)');
title('Figure 1 - Vehicle Position (True, Measured, and Estimated)')

figure;
plot(t,pos-posmeas, t,pos-poshat);
grid;
xlabel('Time (sec)');
ylabel('Position Error (feet)');
title('Figure 2 - Position Measurement Error and Position Estimation Error');

figure;
plot(t,vel, t,velhat);
grid;
xlabel('Time (sec)');
ylabel('Velocity (feet/sec)');
title('Figure 3 - Velocity (True and Estimated)');

figure;
plot(t,vel-velhat);
grid;
xlabel('Time (sec)');
ylabel('Velocity Error (feet/sec)');
title('Figure 4 - Velocity Estimation Error');
```

The Kalman filter not only works well but is theoretically attractive. It can be shown that the Kalman filter minimizes the variance of the estimation error. But what if we have a problem where we are more concerned with the worst case estimation error? What if we want to minimize the "worst" estimation error rather than the "average" estimation error? This problem is solved by the H_∞ filter. The H_∞ filter (pronounced "H infinity" and sometimes written as H^∞) is an alternative to Kalman filtering that was developed in the 1980s. It is less widely known and less commonly applied than the Kalman filter, but it has advantages that make it more effective in certain situations.

Kalman filter theory assumes that the process noise w and the measurement noise z are independent from each other. What if we have a system where these two noise processes are not independent? This is the correlated noise problem, and the Kalman filter can be modified to handle this case. In addition, the Kalman filter requires that the noise covariances S_w and S_z be known. What if they are not known? Then how can we best estimate the state? Again, this is the problem solved by the H_∞ filter.

Kalman filtering is a huge field whose depths we cannot hope to begin to plumb in these few pages. Thousands of papers and dozens of textbooks have been written on this subject since its inception in 1960.

Historical perspective

The Kalman filter was developed by Rudolph Kalman, although Peter Swerling developed a very similar algorithm in 1958. The filter is named after Kalman because he published his results in a more prestigious journal and his work was more general and complete. Sometimes the filter is referred to as the Kalman-Bucy filter because of Richard Bucy's early work on the topic, conducted jointly with Kalman.

The roots of the algorithm can be traced all the way back to the 18-year-old Karl Gauss's method of least squares in 1795. Like many new technologies, the Kalman filter was developed to solve a specific problem, in this case, spacecraft navigation for the Apollo space program. Since then, the Kalman filter has found applications in hundreds of diverse areas, including all forms of navigation (aerospace, land, and marine), nuclear power plant instrumentation, demographic modeling, manufacturing, the detection of underground radioactivity, and fuzzy logic and neural network training. esp

Dan Simon is a professor in the electrical and computer engineering department at

Cleveland State University and a consultant to industry. His teaching and research interests include filtering, control theory, embedded systems, fuzzy logic, and neural networks. He is presently trying to implement a DSP-based motor controller using a Kalman filter. You can contact him at d.j.simon@csuohio.edu.

For further reading

Gelb, A. *Applied Optimal Estimation*. Cambridge, MA: MIT Press, 1974. This is what you call an "oldie but goodie." And don't worry that it's published by MIT Press; it's a simple and straightforward book that starts with the basics and is heavy on practical issues.

Anderson, B. and J. Moore. *Optimal Filtering*. Englewood Cliffs, NJ: Prentice-Hall, 1979. This is very mathematical and difficult to read, but I have relied heavily on it for obtaining a fundamental theoretical understanding of Kalman filtering and related issues.

Grewal, M. and A. Andrews. *Kalman Filtering Theory and Practice*. Englewood Cliffs, NJ: Prentice-Hall, 1993. This is a happy medium between the first two references, a nice balance between theory and practice. One good feature of this book is that it includes Kalman filtering source code on a floppy disk. One not-so-nice feature is that the source code is written in Fortran.

Sorenson, H. *Kalman Filtering: Theory and Application*. Los Alamitos, CA: IEEE Press, 1985. This is a collection of some of the classic papers on Kalman filtering, starting with Kalman's original paper in 1960. The papers are academically oriented, but someone who likes theory will obtain an interesting historical perspective from this book.

http://ourworld.compuserve.com/homepages/PDJoseph/—This is Peter Joseph's Web site, and a useful resource on the topic of Kalman filtering. Dr. Joseph has worked with Kalman filters since their inception in 1960, and coauthored perhaps the earliest text on the subject (in 1968). His Web page includes lessons for the beginning, intermediate, and advanced student.

LISTING 2 Kalman filter equations

```
// The following code snippet assumes that the linear system has n states, m
// inputs, and r outputs.  Therefore, the following variables are assumed to
// already be defined.
// A is an n by n matrix
// B is an n by m matrix
// C is an r by n matrix
// xhat is an n by 1 vector
// y is an r by 1 vector
// u is an m by 1 vector
// Sz is an r by r matrix
// Sw is an n by n matrix
// P is an n by n matrix

float AP[n][n];                  // This is the matrix A*P
float CT[n][r];                  // This is the matrix C^T
float APCT[n][r];                // This is the matrix A*P*C^T
float CP[r][n];                  // This is the matrix C*P
float CPCT[r][r];                // This is the matrix C*P*C^T
float CPCTSz[r][r];              // This is the matrix C*P*C^T+Sz
float CPCTSzInv[r][r];           // This is the matrix (C*P*C^T+Sz)^-1
float K[n][r];                   // This is the Kalman gain.
float Cxhat[r][1];               // This is the vector C*xhat
float yCxhat[r][1];              // This is the vector y-C*xhat
float KyCxhat[n][1];             // This is the vector K*(y-C*xhat)
float Axhat[n][1];               // This is the vector A*xhat
float Bu[n][1];                  // This is the vector B*u
float AxhatBu[n][1];             // This is the vector A*xhat+B*u
float AT[n][n];                  // This is the matrix A^T
float APAT[n][n];                // This is the matrix A*P*A^T
float APATSw[n][n];              // This is the matrix A*P*A^T+Sw
float CPAT[r][n];                // This is the matrix C*P*A^T
float SzInv[r][r];               // This is the matrix Sz^-1
float APCTSzInv[n][r];           // This is the matrix A*P*C^T*Sz^-1
float APCTSzInvCPAT[n][n];       // This is the matrix A*P*C^T*Sz^-1*C*P*A^T

// The following sequence of function calls computes the K matrix.
MatrixMultiply((float*)A, (float*)P, n, n, n, (float*)AP);
MatrixTranspose((float*)C, r, n, (float*)CT);
MatrixMultiply((float*)AP, (float*)CT, n, n, r, (float*)APCT);
MatrixMultiply((float*)C, (float*)P, r, n, n, (float*)CP);
MatrixMultiply((float*)CP, (float*)CT, r, n, r, (float*)CPCT);
MatrixAddition((float*)CPCT, (float*)Sz, r, r, (float*)CPCTSz);
MatrixInversion((float*)CPCTSz, r, (float*)CPCTSzInv);
MatrixMultiply((float*)APCT, (float*)CPCTSzInv, n, r, r, (float*)K);

// The following sequence of function calls updates the xhat vector.
MatrixMultiply((float*)C, (float*)xhat, r, n, 1, (float*)Cxhat);
MatrixSubtraction((float*)y, (float*)Cxhat, r, 1, (float*)yCxhat);
MatrixMultiply((float*)K, (float*)yCxhat, n, r, 1, (float*)KyCxhat);
MatrixMultiply((float*)A, (float*)xhat, n, n, 1, (float*)Axhat);
MatrixMultiply((float*)B, (float*)u, n, r, 1, (float*)Bu);
MatrixAddition((float*)Axhat, (float*)Bu, n, 1, (float*)AxhatBu);
MatrixAddition((float*)AxhatBu, (float*)KyCxhat, n, 1, (float*)xhat);

// The following sequence of function calls updates the P matrix.
MatrixTranspose((float*)A, n, n, (float*)AT);
MatrixMultiply((float*)AP, (float*)AT, n, n, n, (float*)APAT);
MatrixAddition((float*)APAT, (float*)Sw, n, n, (float*)APATSw);
MatrixTranspose((float*)APCT, n, r, (float*)CPAT);
MatrixInversion((float*)Sz, r, (float*)SzInv);
MatrixMultiply((float*)APCT, (float*)SzInv, n, r, r, (float*)APCTSzInv);
MatrixMultiply((float*)APCTSzInv, (float*)CPAT, n, r, n,
(float*)APCTSzInvCPAT);
MatrixSubtraction((float*)APATSw, (float*)APCTSzInvCPAT, n, n, (float*)P);
```