feature

DAN SIMON

From Here to Infinity

H_∞ filters can be used to estimate system states that cannot be observed directly. In this, they are like Kalman filters. However, only H_∞ filters are robust in the face of unpredictable noise sources.

Originally designed in the 1960s for aerospace applications, the Kalman filter ("Kalman Filtering," June 2001, p. 72) is an effective tool for estimating the states of a system. The widespread success of the Kalman filter in aerospace applications led to attempts to apply it to more common industrial applications. However, these attempts quickly made it clear that a serious mismatch existed between the underlying assumptions of Kalman filters and industrial state estimation problems.

Accurate system models are not as readily available for industrial problems. In addition, engineers rarely understand the statistical nature of the noise processes that impinge on such systems. After a decade or so of reappraising the nature and role of Kalman filters, engineers realized they needed a new filter that could handle system modeling errors and noise uncertainty. State estimators that can tolerate such uncertainty are called *robust*. The H_∞ filter was designed for robustness.

What's wrong with Kalman filters?

Recall that the Kalman filter estimates the states x of a linear dynamic system defined by the equations:[2]

$$x_{k+1} = Ax_k + Bu_k + w_k$$
$$y_k = Cx_k + z_k$$

where A, B, and C are known matrices; k is the time index; x is the state of the system (unavailable for measurement); u is the known input to the sys-

tem; y is the measured output; and w and z are noise. The two equations represent what is called a *discrete time system*, because the time variable k is defined only at discrete values (0, 1, 2, …). Suppose we want to estimate the state x of the above system. We cannot measure the state directly; we can only measure y directly. In this case we can use a Kalman filter to estimate the state. If we refer to our estimate as $\hat{x}$, the Kalman filter equations are given as follows:

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

where S_w and S_z are the covariance matrices of w and z, K is the Kalman gain, and P is the variance of the estimation error. The Kalman filter works well, but only under certain conditions.

First, the noise processes need to be zero mean. The average value of the process noise, w_k, must be zero, and the average value of the measurement noise, z_k, must also be zero. This zero mean property must hold not only across the entire time history of the process, but at each time instant, as well. That is, the expected value of w and z at each time instant must be equal to zero.

Second, we need to know the standard deviation of the noise processes. The Kalman filter uses the S_w and S_z matrices as design parameters (these are the covariance matrices of the noise processes). That means that if we do not know S_w and S_z, we cannot design an appropriate Kalman filter.

The attractiveness of the Kalman filter is that it results in the smallest possible standard deviation of the estimation error. In other words, the Kalman filter is the minimum variance estimator.

So what do we do if the Kalman filter assumptions are not satisfied? What should we do if the noise processes are not zero mean, or we don't have any information about the noise statistics? And what if we want to minimize the worst case estimation error rather than the variance of the estimation error?

Perhaps we could use the Kalman filter anyway, even though its assumptions are not satisfied, and hope for the best. Maybe if we ignore the problems we encounter they will go away. This is a common solution to our Kalman filter quandary and it works reasonably well in many cases. However, another option is available: the H_∞ filter, also called the minimax filter. The H_∞ filter does not make any assumptions about the noise, and it minimizes the worst case estimation error. Before we can attack the H_∞ filter problem, we need to cover a couple of simple mathematical preliminaries.

Eigenvalues

If we have a square matrix D, then the eigenvalues of D are defined as all values of λ that satisfy the equation $Dg=\lambda g$ for some vector g. It turns out that if D is an $n\times n$ matrix, then there are always n values of λ that satisfy this equation; that is, D has n eigenvalues. For instance, consider the following D matrix:

$$D = \begin{matrix} 2 & 1 \\ 1 & 2 \end{matrix}$$

The eigenvalues of this matrix are 1 and 3. In other words, the two solutions to the $Dg=\lambda g$ equation are as follows:[3]

$$\begin{matrix} 2 & 1 \\ 1 & 2 \end{matrix} \begin{matrix} 1 \\ 1 \end{matrix} = 1 \begin{matrix} 1 \\ 1 \end{matrix}$$

Eigenvalues are difficult to compute in general. But if you have a 2×2 matrix of the following form:

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

then it becomes much easier. The two eigenvalues of this matrix are:

$$= d_1 + d_3 \pm \sqrt{\left(d_1 + d_3\right)^2 + 4\left(d_2^2 \quad d_1d_3\right)} \quad / 2$$

In the example that we will consider in this article, we will need to compute the eigenvalues of a matrix that has the form of the D matrix above.

That's all there is to eigenvalues. It's a high-tech word that is conceptually simple.

Vector norms

Before we can look at the H_∞ filter problem we need to define the norm of a vector. A vector norm is defined as follows:[4]

$$\|x\| = \sqrt{x^Tx}$$

In order to avoid the square root sign in the rest of this article, we will use the square of the norm:

$$\|x\|^2 = x^Tx$$

So, for example, if the vector x is given as:

$$x = \begin{matrix} 1 \\ 2 \\ 3 \end{matrix}$$

then the norm squared of x is derived as follows:

FIGURE 1 Velocity estimation error

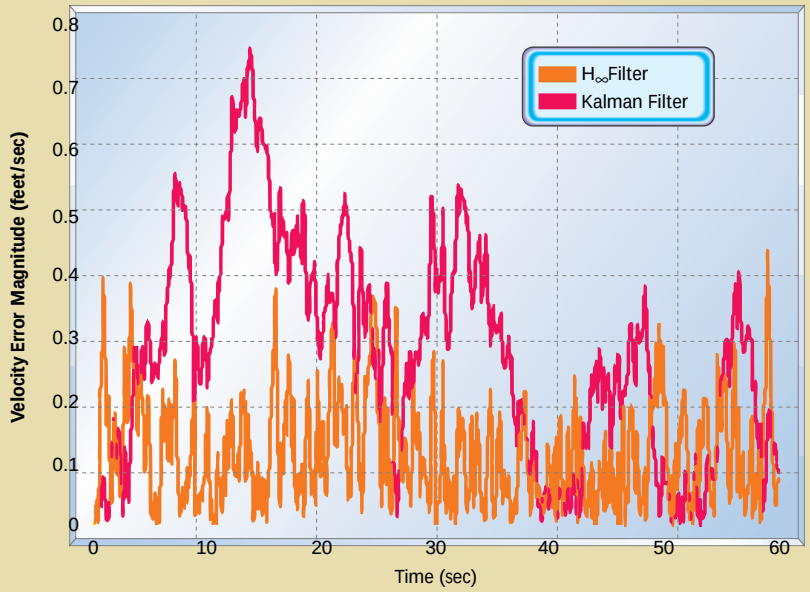
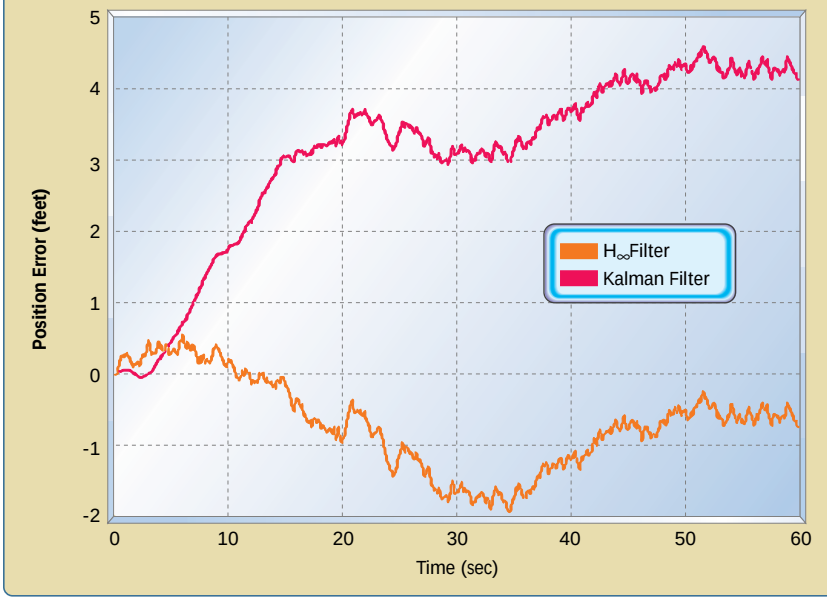


FIGURE 2 Position estimation error



$$\|x\|^2 = \begin{bmatrix}1 & 2 & 3\end{bmatrix}\begin{matrix}1\\2\\3\end{matrix} = 1 + 2 + 3 = 14$$

A weighted vector norm is slightly more complicated, and is defined as:

$$\|x\|_Q^2 = x^T Q x$$

where Q is any matrix with compatible dimensions. So, for example, suppose we have the vector x given previously, and the matrix Q is given as:

$$Q = \begin{matrix}2 & 0 & 0\\0 & 4 & 0\\0 & 0 & 5\end{matrix}$$

This Q matrix is called a diagonal matrix. That means that only the entries along the main diagonal are nonzero. In this case, the Q-weighted norm of x is derived as:

$$\|x\|^2 = \begin{bmatrix}1 & 2 & 3\end{bmatrix}\begin{matrix}2 & 0 & 0\\0 & 4 & 0\\0 & 0 & 5\end{matrix}\begin{matrix}1\\2\\3\end{matrix} = 2 + 8 + 15 = 63$$

We see that the elements of Q serve to weight the elements of x when the norm is computed; hence the term weighted norm. In general, Q could be a nondiagonal matrix, but in this article we can assume that Q is diagonal.

H_∞ filtering

Now that we know what a weighted vector norm is, we can state the H_∞ filter problem and solution. First of all, suppose we have a linear dynamic system as defined at the beginning of this article. Then suppose we want to solve the following problem:

$$\min_{\hat{x}} \max_{w,v} J$$

where J is some measure of how good our estimator is. We can view the noise terms w and v as adversaries that try to worsen our estimate. Think of w and v as manifestations of Murphy's Law: they will be the worst possible values. So, given the worst possible values of w and v, we want to find a state estimate that will minimize the worst possible effect that w and v have on our estimation error. This is the problem that the H_∞ filter tries to solve. For this reason, the H_∞ filter is sometimes called the minimax filter; that is, it tries to minimize the maximum estimation error. We will define the function J as follows:

$$J = \frac{ave\|x_k \quad \hat{x}_k\|_Q}{ave\|w_k\|_W + ave\|v_k\|_V}$$

where the averages are taken over all time samples *k*. In other words, we want to find an $\hat{x}$ that minimizes *J*, so we want to find an $\hat{x}$ that is as close to *x* as possible. Murphy wants to maximize *J*, so Murphy wants to find noise sequences *w* and *v* that cause our estimate $\hat{x}$ to be far from the true state *x*. However, Murphy could make $\hat{x}$ far from *x* simply by making the noise very large. We prevent this by putting the average of the weighted norms of *w* and *v* in the denominator of *J*. That way, Murphy has to find noise terms that will make *J* large without simply using brute force.

The previous equation is the statement of the H_∞ filtering problem. Our task is to find a state estimate that makes *J* small even while Murphy is finding noise terms that make *J* large. The *Q*, *W*, and *V* matrices that are used in the weighted norms in *J* are chosen by us, the designers, to obtain desired trade-offs. For example, if we know that the *w* noise will be smaller than the *v* noise, we should make the *W* matrix smaller than the *V* matrix. This will de-emphasize the importance of the *w* noise relative to the *v* noise. Similarly, if we are more concerned about estimation accuracy in specific elements of the state vector *x*, or if the elements of the state vector *x* are scaled so that they differ by an order of magnitude or more, then we should define the *Q* matrix accordingly.

Well, it turns out that the estimation problem is too difficult to solve mathematically. However, we can solve a related problem. We can solve the problem:

$$J < 1/$$

where γ is some constant number chosen by us, the designers. That is, we can find a state estimate so that the maximum value of *J* is always less than $\hat{x}$, regardless of the values of the noise terms *w* and *v*. The state estimate that forces $J < 1/\gamma$ is given as follows:

$$L_k = \left(I \quad QP_k + C^T V^{-1} C P_k\right)^{-1}$$
$$K_k = AP_k L_k C^T V^{-1}$$
$$\hat{x}_{k+1} = A\hat{x}_k + Bu_k + K_k\left(y_k \quad C\hat{x}_k\right)$$
$$P_{k+1} = AP_k L_k A^T + W$$

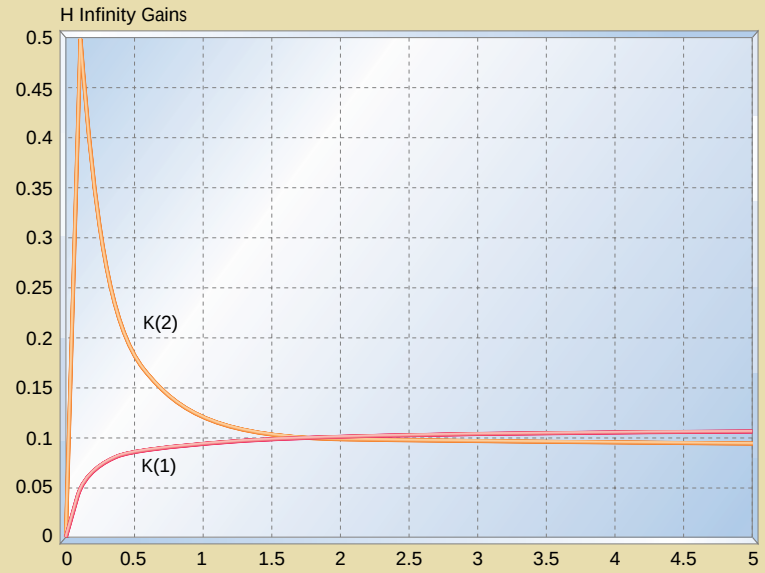
These are the H_∞ filter equations. They have the same form as the Kalman filter equations, but the details are different. *I* is the identity matrix. This means it is a matrix consisting entirely of zeros, except for the main diagonal, which is made up of ones. For instance, a 3×3 identity matrix would look like this:

$$I = \begin{matrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{matrix}$$

K_k is the H_∞ gain matrix. The initial state estimate $\hat{x}_0$ should be initialized to our best guess of x_0, and the initial value P_0 should be set to give acceptable filter performance. In general, P_0 should be small if we are highly confident of our initial state estimate $\hat{x}_0$. If we use these H_∞ filter equations, we guarantee that $J < 1/\gamma$. That means that no matter what Murphy does with the noise terms *w* and *v*, the ratio of the estimation error to the noise will always be less than $1/\gamma$.

Well then, let's just set γ to a very large number, say a zillion or so! That way we can guarantee that our estimation error is almost zero, since $1/\gamma$ will be almost zero. Well, we can't quite do that. The mathematical derivation of the H_∞ equations is valid only if γ is chosen such that all of the eigenvalues of the *P* matrix have magnitudes less than one. If we choose too large of a γ, a solution to the H_∞ filtering problem does not exist. That is, we cannot find an estimator that will make the estimation error arbitrarily small.

FIGURE 3 Technical art specs.



Matrix inversion!?

The observant reader will notice that a matrix inversion is required at every time step, in order to compute the L_k terms. As discussed in the Kalman filter article, the inversion of small matrices is fairly easy, but the inversion of a matrix larger than 3×3 could require too much computational time for a practical implementation. Fortunately,

LISTING 1 H_∞ filter equations

```
float CT[n][r];                 // This is the matrix CT
float CTVinv[n][r];             // This is the matrix CT*V 1
float CTVinvC[n][n];            // This is the matrix CT*V 1*C
float CTVinvCP[n][n];           // This is the matrix CT*V 1*C*P
float QP[n][n];                 // This is the matrix Q*P;
float QPgamma[n][n];            // This is the matrix Q*P*gamma
float EYEQPgamma[n][n];         // This is the matrix EYE Q*P*gamma
float Linv[n][n];               // This is the inverse of the L matrix
float L[n][n];                  // This is the L matrix
float AP[n][n];                 // This is the matrix A*P
float APL[n][n];                // This is the matrix A*P*L
float APLCT[n][r];              // This is the matrix A*P*L*CT
float K[n][r];                  // This is the H∞ gain matrix
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 AT
float APLAT[n][n];              // This is the matrix A*P*L*AT

// The following sequence of function calls computes the L matrix.
MatrixTranspose((float*)C, r, n, (float*)CT);
MatrixMultiply((float*)CT, (float*)Vinv, n, r, r, (float*)CTVinv);
MatrixMultiply((float*)CTVinv, (float*)C, n, r, n, (float*)CTVinvC);
MatrixMultiply((float*)CTVinvC, (float*)P, n, n, n, (float*)CTVinvCP);
MatrixMultiply((float*)Q, (float*)P, n, n, n, (float*)QP);
MatrixScalarMultiply((float*)QP, gamma, n, n, (float*)QPgamma);
MatrixSubtraction((float*)EYE, (float*)QPgamma, n, n, (float*)EYEQPgamma);
MatrixAddition((float*)EYEQPgamma, (float*)CTVinvCP, n, n, (float*)Linv);
MatrixInversion((float*)Linv, n, (float*)L);

// The following sequence of function calls computes the K matrix.
MatrixMultiply((float*)A, (float*)P, n, n, n, (float*)AP);
MatrixMultiply((float*)AP, (float*)L, n, n, n, (float*)APL);
MatrixMultiply((float*)APL, (float*)CT, n, n, r, (float*)APLCT);
MatrixMultiply((float*)APLCT, (float*)Vinv, 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);
```

Listing 1 continued on p. #.

```
// The following sequence of function calls updates the P matrix.
MatrixTranspose((float*)A, n, n, (float*)AT);
MatrixMultiply((float*)APL, (float*)AT, n, n, n, (float*)APLAT);
MatrixAddition((float*)APLAT, (float*)W, n, n, (float*)P);
```

this problem is not insurmountable. Notice from the H_∞ filter equations that the calculation of L_k, P_k, and K_k can be performed off line. That is, we do not need the measurements to compute the H_∞ gain matrix K_k. We can compute K_k off line (on our development system, for example) and then hard-code K_k into our embedded system.

This solution, however, just shifts the problem to another arena. Now, instead of being strapped for throughput, we are out of memory. If we try to store a K_k matrix in our embedded system for each time step of our program execution, we will quickly run out of memory. But again, a solution exists. When we compute the K_k matrices off line, we notice that they very quickly approach what is called a *steady state solution*. Although K_k is written as a function of the time step k, the matrices K_k will be constant after just a few time steps. So we can run the H_∞ gain equations off line until we see K_k converge to a constant matrix, and then simply hard-code that constant matrix into our embedded system. The result is called the steady-state H_∞ filter.

As an aside, notice that we can do the same thing with the Kalman filter equations shown at the beginning of this article. We can compute the Kalman gain matrix K_k off line until we see it converge to a constant matrix. Then we can hard-code that constant matrix into our embedded Kalman filter. The result is called the steady-state Kalman filter.

But does the steady-state H_∞ filter perform as well as the time-varying H_∞ filter? And does the steady-state Kalman filter perform as well as the time-varying Kalman filter? We can save computational resources by using the steady-state versions of the filter, but how much performance will we sacrifice? The answer depends on the application, but often we lose very little performance. In many cases, the drop in performance will be negligible. We will see this in the simulation results that follow.

Vehicle navigation

To demonstrate the H_∞ filter, we will consider a vehicle navigation problem, similar to the one that we used for our Kalman filter example. Say we have a vehicle traveling along a road, and we are measuring its velocity (instead of position). The velocity measurement error is 2 feet/sec (one sigma), the acceleration noise (due to potholes, gusts of wind, motor irregularities, and so on) is 0.2 feet/sec^2 (one sigma), and the position is measured 10 times per second. The linear model that represents this system is similar to the model discussed in the Kalman filter article. The x_{k+1} equation is the same, but the y_k equation is different. In the Kalman filter article, we measured position, and in this article we are measuring velocity.

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

$$y_k = \begin{matrix}[0 & 1]\end{matrix} x_k + z_k$$

The state vector x is composed of vehicle position and velocity, u is the known commanded vehicle acceleration, and w is acceleration noise. The measured output y is the measured velocity corrupted by measurement noise z. As explained in the Kalman filter article, we derive the S_z and S_w matrices used in the Kalman filter equations as follows:

$$S_z = 100$$

$$S_w = \begin{matrix} 10^{6} & 2\ 10^{5} \\ 2\ 10^{5} & 4\ 10^{4} \end{matrix}$$

It looks, though, as if a fly has landed in the ointment. We *thought* that the velocity measurement noise was Gaussian with a standard deviation of 2 feet/sec. In reality, the measurement noise is uniformly distributed between ±1 foot/sec. That is, the true measurement noise is a random number with all values between ±1 foot/sec equally likely. In addition, we thought that the acceleration noise had a standard deviation of 0.2 feet/sec^2. In reality, the acceleration noise is twice as bad as we thought. It actually has a standard deviation of 0.4 feet/sec^2. This is because we took the road less traveled, and it has more potholes than expected.

When we simulate the Kalman filter and the H_∞ filter (γ=0.01) for this problem, we obtain the results shown in Figures 1 and 2 for the velocity estimation error and the position estimation error. Figure 1 shows that the velocity error is noticeably less for the H_∞ filter than for the Kalman filter. Figure 2 shows that the position error is much better with the H_∞ filter. The Kalman filter does not perform as well as we expect because it is using S_z and S_w matrices that do not accurately reflect reality. The H_∞ filter, on the other hand, makes no assumptions at all about the noise w and z, except that nature is throwing the worst possible noise at our estimator.

Next, we consider the steady-state H_∞ filter. Figure 3 shows the H_∞ filter gain matrix K_k as a function of time for the first five seconds of the simulation. Since we are estimating two states, and we have one measurement, K_k has two elements. Figure 3 shows that the elements of K_k quickly approach the approximate steady state values $K_k(1) = 0.11$ and $K_k(2) = 0.09$.

Let's pretend that we are poor struggling embedded systems engineers who can't afford a fancy enough microcontroller to perform real-time matrix inversions. To compensate, we simulate the steady-state H_∞ filter using the hard-coded gain matrix:

$$K = \begin{matrix} 0.11 \\ 0.09 \end{matrix}$$

The results are practically indistinguishable from the time-varying H_∞ filter. We can avoid all the work associated with real-time matrix inversion and pay virtually no penalty in terms of performance. The Matlab code that I used to generate these results is available at *www.embedded.com/code.htm*. If you use Matlab to run the code you will get different results every time because of the random noise that is simulated. Sometimes the Kalman filter will perform even better than the H_∞ filter. But the H_∞ filter generally outperforms the Kalman filter for this example.

Pseudocode for implementing an H_∞ filter in a high-level language is shown in Listing 1. This code assumes that the linear system has n states, m inputs, and r outputs. The following variables are assumed:

`A`: an $n \times n$ matrix
`B`: an $n \times$m matrix
`C`: an $r \times n$ matrix
`xhat`: an $n \times 1$ vector
`y`: an $r \times 1$ vector
`u`: an $m \times 1$ vector
`P`: an $n \times n$ matrix
`Vinv`: an $r \times r$ matrix
`W`: an $n \times n$ matrix
`EYE`: the $n \times n$ identity matrix
`Q`: an $n \times n$ matrix
`gamma`: a scalar

Infinity and beyond

The simple example shown in this article demonstrates the potential benefits of H_∞ filtering. However, H_∞ theory is actually much more general than we have discussed here. It can also be used to design control systems (not just estimators) that are robust to unknown noise sources. Our example showed the robustness of H_∞ filtering to unknown noise sources. However, the H_∞ filtering problem can also be formulated in such a way that the filter is robust to system modeling errors. In our example, we assumed that the system model was given by:

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

where A and B were known matrices. However, suppose we have some uncertainty in our model. Suppose the system model is given by:

$$x_{k+1} = (A + \quad A)x_k + (B + \quad B)u_k + w_k$$

where, as before, A and B are known, but ΔA and ΔB are unknown matrices. We may have some bounds on ΔA and ΔB, but we do not know their exact values. Then the H_∞ filtering problem can be reformulated to minimize the effect of these modeling errors on our estimation error. The H_∞ filter can therefore be designed to be robust to uncertainty in the system model.

It appears that H_∞ filtering was first introduced in 1987 by Mike Grimble, a professor at the University of Strathclyde in the UK. It has its roots in the mathematics developed in 1981 by George Zames, a professor at McGill University in Montreal. Since the late 1980s, there have been several different approaches to the formulation of the H_∞ filter. In Kalman filtering, different approaches all lead to the same (or similar) equations. With H_∞ filtering, different approaches lead to widely different equations. Perhaps this is one reason that H_∞ filtering is not as well known as Kalman filtering; so many different formulations of the H_∞ filter are available that it is difficult to get an overarching view of the entire field.

The Kalman filter is more widely used and understood, and generally gives better performance than the H_∞ filter. The H_∞ filter requires more tuning to get acceptable performance. Looking at the H_∞ filter equations, we can see that the tuning parameters include γ, P_0, V, W, and Q. That's a lot of parameters to tune, especially when P_0, V, W, and Q are all matrices. For this reason, it is often more difficult to tune an H_∞ filter than a Kalman filter. However, the effort required to tune an H_∞ filter can be worthwhile, as seen from the simulation results in this article.

Since Kalman filtering generally performs better than H_∞ filtering, but the H_∞ approach is so intuitively appealing, some researchers have proposed mixed Kalman/H_∞ filtering. This is usually referred to as mixed H_2/H_∞ filtering. This is an approach to find the best state estimator in the Kalman filter sense subject to the constraint that the maximum estimation error is bounded.

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 teaches a course on H_∞-based control and estimation, and is presently working on a project to evaluate aircraft engine health using H_∞ filtering. You can contact him at d.j.simon@csuohio.edu.

Endnotes

1. Also discussed in the Kalman filtering article was the concept of linear systems, which should be considered a prerequisite to this article.
2. The Kalman filter is also called the H_2 filter because it minimizes the two-norm of the transfer function from the system noise to the state estimation error.
3. Note that D has to be square in order for the concept of eigenvector to exist. If D is not square, then, by definition, it does not have any eigenvalues. The g vector in the eigenvalue equation is called an eigenvector, but we will not concern ourselves with eigenvectors in this article.
4. More specifically, this is the two-norm of a vector. However, we will refer to it simply as the norm for ease of notation.

For further reading

Shen, W. and L. Deng. "Game Theory Approach to Discrete H_∞ Filter Design," *IEEE Transactions on Signal Processing*, pp. 1092-1095, April 1997.
This is an academic paper that derives the equations on which this article is based.

Simon, D. and H. El-Sherief. "Hybrid Kalman/Minimax Filtering in Phase-Locked Loops," *Control Engineering Practice*, pp. 615-623, 1996.
This paper presents a formulation of the H_∞ filter that is different from the one given in this article. The H_∞ filter is then combined with a Kalman filter to obtain a new estimator that performs better than either the Kalman filter or the H_∞ filter.

www.innovatia.com/software/papers/minimax.htm
This web page (and its links) summarize the approach and results of the Control Engineering Practice paper referenced above.

Burl, J. *Linear Optimal Control*. Reading, MA: Addison Wesley, 1999.
This is one of the best and most accessible texts that covers H_∞ filtering. However, that's not saying much. There aren't many books on the subject, perhaps because of its novelty. One good point about this book is that the author makes all of the Matlab files used in the examples available on the web. However, this text is limited to continuous time systems and is full of confusing typographical errors. Hopefully these problems will be addressed in a future printing.

Green, M. and D. Limebeer. *Linear Robust Control*. Englewood Cliffs, NJ: Prentice Hall, 1995.
This reference, slightly more academically oriented than Burl's book, offers good theoretical coverage of discrete time H_∞ filtering, but it doesn't have any examples. It is out of print but, you can find copies (new and used) from bookstores on the Internet.