Tutorial on Filtering, Restoration, and State Estimation

Edited by Christopher Brown

The University of Rochester
Computer Science Department
Rochester, New York 14627

Technical Report 534

June 1995

Abstract

This tutorial is dedicated to our long-suffering 442 students, and to the excellent authors from whom I shamelessly cribbed this work. It is a pure cut-and-paste job from my favorite sources on this material. This is *not* my own work – think of me as an editor working without his authors' permissions. Readers should know that original authors are usually easier to understand than rehashed versions. If this presentation helps you, good. If not it at least helped me sort a few things out.

I assume knowledge of all necessary linear systems theory, differential equations, statistics, control theory, etc. We start with the ideas of filtering, smoothing, prediction, and state estimation. Wiener filtering and its associated intellectual framework follows, with a brief foray into ARMA filtering. The idea of recursive estimation is introduced to give some motivation for the slog ahead, and then we start with basic concepts in maximum likelihood, maximum a posteriori, and least-squares estimation. The strategy is to work toward the Kalman Filtering equations by showing how they are simply related to general least-squares estimation. After Kalman filtering, some simpler versions of recursive filters are presented. There are appendices on the orthogonality principle, the matrix inversion lemma, singular value decomposition, partial C and LISP code, and a worked example.

The University of Rochester Computer Science Department supported this work.

Contents

1 Filtering, Smoothing, and Prediction

In this context, a *filter* is a procedure that looks at a collection or stream of data taken from a *system*, or *plant*, or *process*. The system is described by a *state equation*. The filter estimates *parameters* or system *state variables*. System parameters are usually taken to be time-invariant or slowly-varying properties of the system. The set of system state variables is any set of variables that completely describe its state as far as you care. You are you are trying to control or discover the system's state, and you want to estimate these state variables from the collection of data you gather. For instance you could be interested in the position, velocity, and acceleration of an object, so these would be its state variables. The state equation might be that of a point moving under constant acceleration. The data you might have available is a stream of noisy position measurements.

Usually in this business one thinks of data ordered through time, and a procedure that does not use "future" data is *causal*, else it is *non-causal*. Non-causal temporal filters are non-physical (impossible). In image processing the literature often assumes the signal comes in through time in scan-line order. Computer vision people often deal with images that have been captured and are now static in time. That means that a simple-minded computer-vision image smoothing filter that takes a boxcar average of a pixel and its 8-neighbors is "non-causal" if you think of the image as arriving in scanline order. Some image processing people use that sort of terminology, so watch for it: to them "the future" is any scan line below you or anything to the right on your scanline.

Back in the time domain, say you are getting a vector of data $\mathbf{x}(k), k = 0, 1, 2..., T$ through time (for the most part I stick to discrete data throughout this note) and feeding it to your filter. There are three *estimation* problems you could want to solve.

- Estimate the *current* value $\mathbf{x}(T)$ given all the data so far, including $\mathbf{x}(T)$: this is called *filtering* (in a restricted technical sense).
- Estimate some *past* value of $\mathbf{x}(k), k < T$ given all the data so far, including $\mathbf{x}(T)$: this is called *smoothing*.
- Estimate some *future* value of $\mathbf{x}(k), k > T$ given all the data so far, including $\mathbf{x}(T)$: this is called *prediction*.

You should be able to see why these forms of estimation have these names.

"Optimal" estimation techniques assume you have a model for some underlying form of the data you are observing, and for the noise processes that affect your measurements, and that you have an error criterion you want to minimize. A good example is the family of least-squared error estimates (of which we'll see a lot). Consider fitting a straight line to some scattered data in some normal, intuitive way (you could use techniques you may have learned under the names of linear regression, principal components analysis, least-squares fit, etc. or you just eyeball it and draw a line through the middle with a ruler). Why would you do that? Why pick that particular line, or indeed any straight line at all? You'd have to believe that the "true" data actually are produced by a linear process. If you believed it was an exponential decay type phenomenon you'd want to fit a straight line

on a log scale, for instance. If you knew you were looking at positions of a freely falling object you might want to fit a parabola. Further, you have to believe that the noise that is affecting your measurements (*measurement noise*) or its behavior (*plant noise*) has mean zero. Otherwise your whole data set would be displaced in one direction or the other. Thus running through all of this work are certain important assumptions about the underlying properties of the systems or processes being estimated, and about the various noise processes that can interfere with the estimation.

2 Pseudoinverse Methods

This section and the next are written from the context of image processing. The techniques are of course more general than this family of applications, but this application has the advantage that the results can be shown visually. You should go to the real literature for these compelling images.

2.1 Vanilla Pseudoinverse Approach

Pratt's book [8] has excellent coverage of all manner of digital image processing lore, with many illustrative images of the effects of various processing methods. This section is stolen from his treatment.

Idea: if your image degradation could be modeled as a linear transformation of the image (this of course includes any degradation like blurring that results from a convolution), then why not just write down the transformation matrix and invert it? This idea is the basis of pseudoinverse and SVD (Appendix B) methods. These methods have some practical difficulties but they are elegant in conception.

Recall that the pseudo-inverse is the canonical way to solve over-determined equations in a least squared-error sense. Recall (e.g. the appendix of [1]) that if $\mathbf{B}$ is a matrix, then $\mathbf{B}^\dagger$ is the pseudoinverse, where

$$\mathbf{B}^\dagger = (\mathbf{B}'\mathbf{B})^{-1}\mathbf{B}'$$

where as always in this document the prime denotes matrix transpose.

Then if the blurring is described by

$$\mathbf{g} = \mathbf{B}\mathbf{f}$$

and the best restoration is given by

$$\hat{\mathbf{f}} = \mathbf{B}^\dagger\mathbf{g}.$$

Easy, eh? There are several problems, such as that for a 512×512 image and observation, $\mathbf{B}$ has 68,719,476,736 elements. Of course for a normal blurring-type distortion $\mathbf{B}$ is very sparse and very structured, but still... Luckily if the blur function is separable into row and column blur operators, then the pseudoinverse is also separable, and the restoration can be found by applying the generalized inverse of the row blur matrix to the observations, then the inverse of the column blur to the result.

Furthermore, think about getting an "over-determined" observation from the blurred image. Suppose you just sample the blurred input at a very high resolution, higher than the desired output? What happens is that the rows of the blur matrix become more and more linearly dependent, and the rank of the matrix drops relative to its size.

If a noise vector $\mathbf{n}$ is added to the blurred image, then the reconstruction will be off by the additive vector $\mathbf{B}^\dagger\mathbf{n}$.

There are some techniques for making this approach practical [8].

2.2 SVD Approach

The strategy is to decompose the blur matrix $\mathbf{B}$ into eigenmatrices via the SVD (Appendix B). Then sequentially estimate by iteratively taking into account eigenmatrices of smaller and smaller singular values. Since conditioning problems occur when you start effectively using rank-deficient matrices, and the SVD measures how rank-deficient the matrix is and orders its components by their contribution to the blur matrix, the SVD (as always) allows you to control the effects of rank-deficiency.

Formally and for example, consider an image that has been corrupted by a blur matrix and then has had noise added:

$$\mathbf{g} = \mathbf{B}\mathbf{f} + \mathbf{n}$$

By eq. (42) in Appendix B, write $\mathbf{B}$ as

$$\mathbf{B} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}'$$

Since $\mathbf{U}$ and $\mathbf{V}$ are unitary (if real, they are orthogonal) it is easy to write down the pseudoinverse $\mathbf{B}^\dagger$ as

$$\mathbf{B}^\dagger = \mathbf{V}\mathbf{\Lambda}^{-1/2}\mathbf{U}' = \sum_{i=1}^{R}[\lambda(i)]^{-1/2}\mathbf{v}_i\mathbf{u}_i'.$$

The generalized inverse estimate is then just the result of multiplying this expression for $\mathbf{B}^\dagger$ on the right by $\mathbf{g}$. Note then that $\mathbf{u}_i'\mathbf{g}$ is a scalar and so can be moved around like a scalar, and the resulting sequential estimation formula is

$$\hat{\mathbf{f}}_k = \hat{\mathbf{f}}_{k-1} + [\lambda(i)]^{-1/2}(\mathbf{u}_k'\mathbf{g})\mathbf{v}_k.$$

This looks like we are just summing up weighted column vectors, and it's true: remember here we are representing the whole image as a column vector. Again, considerable computational savings can be achieved if the blur is spatially invariant [8].

3 Stochasticity, Stationarity, Wiener Filtering

We'll start out with the motivation of reconstructing or removing noise from signals that have been perturbed by additive Gaussian noise. We shall use linear theory so we can characterize our reconstructing or noise-removing filters as convolutions with a filter kernel.

What is *normally distributed* or *Gaussian* additive noise? The idea is that whatever you are dealing with (state variable, measurement...) has added to it a Gaussian random variable with mean zero and variance σ^2. Remember that the *probability density function*, or PDF, of such a variable is

$$p(x) = N(x; \bar{x}, \sigma^2) \doteq \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{-\frac{(x-\bar{x})^2}{2\sigma^2}\right\}. \tag{1}$$

(Here and throughout I use $\doteq$ to denote "defined as".)

In the context of stochastic signals, if the amplitude of the noise process varies with (spatial) frequency it is called "colored noise"; if the noise is equally powerful at all frequencies it is called "white".

This section is stolen from Horn's excellent chapter on continuous signal processing [6]. We want to reason about classes of signals that are random in the sense that they cannot be exactly predicted: telephone conversation sound waveforms, weather radar returns, television images. Characterizing such ensembles and their joint behaviors is very difficult, and a usual approximation is to talk about the first few moments of the relevant distributions. Thus to describe the ensemble of random signals emitted by some process we could talk about its mean value, or its mean and (auto)covariance (the generalization of the variance of a random variable). In what follows, we indeed simplify things by assuming that the signals in which we are interested are described only up to this rather rough statistical level. Further, if the statistics of the noise and signal do not vary in time, the processes are called *stationary*; there are several sorts of stationarity and the technical definitions are straightforward, but all we need to know is that stationarity captures the time-invariance of the process. Another useful word and concept in this area is *ergodicity*, which captures another sort of invariance; basically the ergodic property is what lets us assume that we can learn about "along-the process" behavior by looking at samples of the signals that are obtained from the ensemble "across the process".

Another way to think about the mean and (co)variance properties of the signal is in terms of (spatial) frequencies.[1] The auto-covariance is intimately related to the autocorrelation (a sign difference in the integral) and the autocorrelation is the Fourier transform of the power spectral density. So we are going to be designing filters that work optimally on signals with particular power spectra. So you might imagine that all "talking heads" images look about alike in their spatial frequency content, or that outdoor nature scenes or city scenes might have characteristic amounts of high- and low-frequency image content, and we might be designing filters for them.

Now there is really only one idea in this whole filtering game:

> *Pay more attention to the signal as it overpowers the noise, and pay less attention as the noise overpowers the signal.*

[1] In this document, "frequency" should be interpreted as spatial or temporal depending on context.

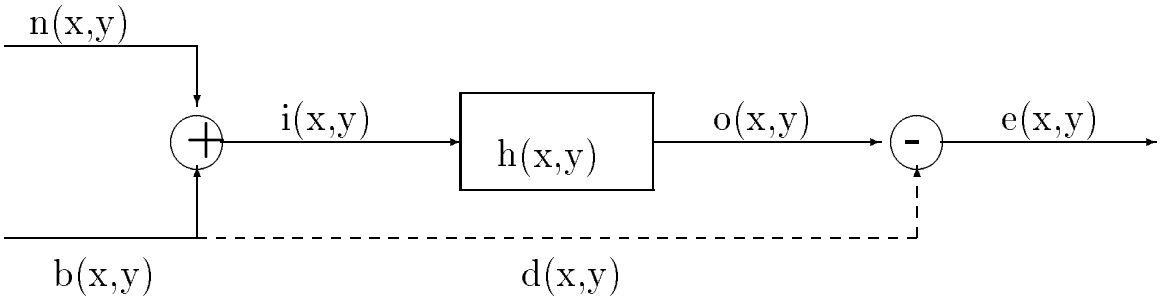

Figure 1: The optimal filter of impulse response h minimizes the difference (in a least-squared sense) e between its output and some desired signal d (here equal to the input signal b), assuming that noise n has been added to b.

So in terms of power spectra, imagine that we can characterize the signal's power spectrum, so we know at what frequencies there is most signal power. Also suppose we know the noise's power spectrum. At any frequency we could imagine attenuating the signal by the ratio of noise to signal at that frequency; this would mean we would lose information about the signal but we would not inject the noise into our estimate as good data. As it happens, exactly this strategy is what gives the best (least squared error) expected estimate of the signals from the ensemble.

To hammer this point home, imagine the normal 2-D plot of a power spectrum (squared modulus of Fourier transform) of an image. A bandpass filter that rejects certain frequencies reduces or eliminates an annulus in the Fourier transform, and inverting the transform gives the filtered image. Attenuating or removing a pair of symmetric pie-slices from the transform removes a range of spatial frequency orientations (thus removing vertical edges, say). The filters we are describing do that sort of thing by attenuating the signal for frequencies at which noise power is highest.

Fig. 1 describes the situation for optimal filtering. The problem is formalized as least-squares minimization of e in any book on filtering, and the result is the Wiener-Hopf equation. Pratt's book [8] shows an elegant solution based on the orthogonality principle (Appendix A).

Consider the image and our observations of it to be written as column vectors. Let the observation be a $P \times 1$ vector $\mathbf{i}$ and the image be a $Q \times 1$ vector $\mathbf{d}$. We desire a linear operation to estimate $\mathbf{i}$:

$$\hat{\mathbf{i}} = \mathbf{W}\mathbf{d} + \mathbf{b},$$

where $\mathbf{W}$ is a $Q \times B$ restoration matrix and $\mathbf{b}$ is a $Q \times 1$ bias vector. We must choose $\mathbf{W}$ and $\mathbf{b}$ to minimize the mean-squared error of our estimate. Orthogonality principle to the rescue. From the last equation and eq. (40),

$$\mathbf{b} = E[\mathbf{d} - \mathbf{W}E[\mathbf{i}], \tag{2}$$

And from eq. (41),

$$E[(\mathbf{W} + \mathbf{i} - \mathbf{d})(\mathbf{i} - E[\mathbf{i}])'] = 0. \tag{3}$$

Substituting in this last equation for **b** using eq. (2) yields directly

$$\mathbf{W} = \mathbf{K}_{id}\mathbf{K}_{ii}^{-1},$$

where $\mathbf{K}_{ii}$ is the $P \times P$ covariance matrix of the observation vector (assumed nonsingular) and $\mathbf{K}_{id}$ is the $Q \times P$ cross-covariance matrix between image and observation. Just to repeat, this shows that the optimal restoration matrix is determined only in terms of first and second moments of the image and observations – truly elegant. Further, this solution is true even for nonlinear and space-variant degradations.

The following completely outrageous demonstration is just to make the result reasonable and does NOT constitute a derivation of the Wiener-Hopf equation. Let us denote the convolution operation by *, and let's leave off the (x, y) indices for the quantities like input, output, desired signal in Fig. 1. The resulting equations then work for time series too. Assume that the filter output should equal the desired signal

$$d = o.$$

We know that the output is the convolution of the input with h:

$$h * i = o.$$

Now practically we don't know much about these inputs and outputs, but as we have argued above it is reasonable that we might measure basic statistical properties like their power spectra, or equivalently that we would know their autocorrelation and cross-correlation functions. Thus if we have $o = d$, we can convolve both sides of $o = i * h$ with i:

$$h * i * i = o * i.$$

Now $i * i$ is the autocorrelation of the input (often written ϕ_{ii}) and $o * i$ is the cross-correlation of the input and desired signals, often written ϕ_{id}. The Wiener-Hopf equation can be written (with continuous versions of the autocorrelation etc functions) as

$$\phi_{id} = \phi_{ii} * h.$$

To solve it, Fourier transform both sides: the convolution goes to a multiplication and the convolutions go to power spectra:

$$H = \frac{\Phi_{id}}{\Phi_{ii}}$$

To see how a solution to the Wiener-Hopf equation in a particular instance fits with our intuition of paying attention to the data that is less noisy, consider the goal of signal estimation, or noise removal, when b is an image we want to recover and n is additive noise that is uncorrelated with the signal[2]. We want the output o to be a close as possible in a least squares sense to b. Thus

$$d = b.$$

[2]Many noise processes are correlated with the signal – the natural variation in the number of photons captured by a sensitive detector is dictated by "photon statistics" in which the variance in the number of detections goes up with the number of detections.

We know

$$i = b + n,$$

so correlating each side of this last equation with d (that is, also b) gives

$$i * d = b * b + n * b.$$

Likewise with the expanded expression for i above we can write

$$i * i = b * b + b * n + n * b + n * n.$$

Since the noise is uncorrelated with the signal, we can substitute

$$b * n = n * b = 0.$$

Fourier transforming, we get

$$H = \frac{\Phi_{id}}{\Phi_{ii}} = \frac{\Phi_{bb}}{\Phi_{bb} + \Phi nn} = \frac{1}{1 + \frac{\Phi nn}{\Phi_{bb}}}.$$

In the last form of this equation, you can see that the filter passes the signal unmodified if Φ_{nn}, the noise power, is zero. For any frequency, as the ratio of noise power to signal power grows, the denominator grows, and the filter attenuates more.

I refer you to [6; 5] for more examples and two different derivations of the Wiener-Hopf equation.

4 ARMA Filters: Filtering meets Control Theory

In 1994 Jeff Schneider and I were trying to design a simple-but-stable controller for a simulated flexible beam. The model had springs in it and the beam "rang" at different frequencies when hit with an impulse. (To get the impulse response, Jeff torqued the simulation one way on tick 0 and back the other way at tick 1, kept a record of the tip deflections, Fourier transformed them and *voila!*). These frequencies were being excited by the output of the PID (proportional, integral, derivative) controller, which was demanding very jerky torques, and eventually they would make the system go unstable. We knew the system was approximately linear, and that if we could filter out those frequencies in the control output, we'd keep the beam from ever responding to them.

Now as it happened we had an analytical handle on these frequencies, and it may have been possible to design a PID (thus second-order) controller to cancel out the first couple of poles (resonances). But we had a lot of empirical work behind us from B.J. Fesq, who had tried lots of different PID gains and couldn't find a combination to damp out those vibrations effectively.

So Jeff went to the MATLAB signal-processing workbench and got coefficients for a filter to damp out those frequencies, implemented as a tapped delay-line line that just outputs a linear combination of the last n inputs. The coefficients were provided by MATLAB, the

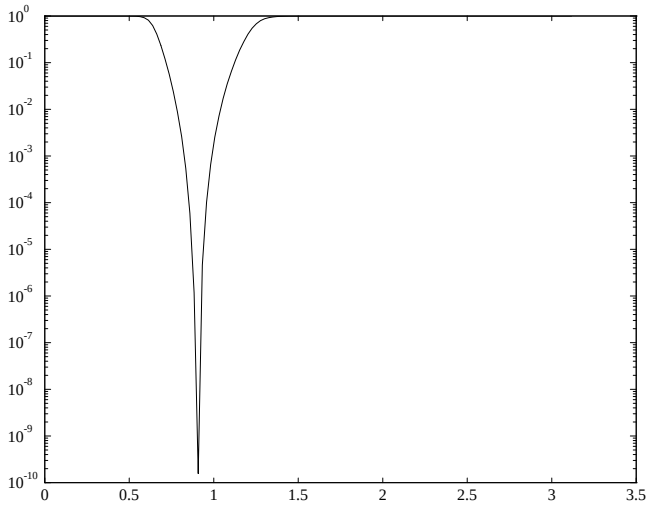

Figure 2: Amplitude response of digital "band-stop" filter

elegant and simple implementation of the tapped-delay line was provided by Jeff. MATLAB was useful because as you know, sharp cutoffs in filters cause nasty ringing effects, and you pay MATLAB to know how to give you coefficients for filters with nice smooth properties. Since the system was almost linear it didn't much matter where we put the filter, and Jeff put it in the feedback path, to remove the two resonant frequencies from the reported tip position. It worked: the resonant frequencies were banished, and a slight adjustment to the PID controller stabilized the system. Figs. 2, 3, and 4 show frequency, phase, and impulse responses of a "Butterworth" filter designed by MATLAB that is to block frequencies between 100 and 200Hz.

This section is a very brief introduction to these sorts of digital filters, and it relates them and their behavior to the concepts of "poles" and "zeros" that are common in the traditional control- and filter-theoretic literature. Basically, the poles are eigenvalues that tell you the resonant frequencies of a system whose transfer function is a rational function. The poles are roots that make the denominator of the transfer function go to zero, hence its response goes to infinity. Zeros are roots of the numerator, which make the response go to zero.

A general tapped delay-line filter that produces a combination of a *moving average* (MA) of its input and a feedback *auto-recursive* (AR) component is shown in Fig. 5. It is easy to see that with only the upper MA connections an impulse input (that is, a single 1 coming down the wire, preceded and followed by infinite zeros) will yield a response that dies away after a certain time (the length of the delay line, in fact) – it is a *finite impulse response* (FIR) filter. Contrariwise, the loopback of the AR connections means that the output can be fed back into the filter forever; it is an *infinite impulse response* (IIR) filter.

We can write the time-domain behavior of an ARMA filter as a difference equation:

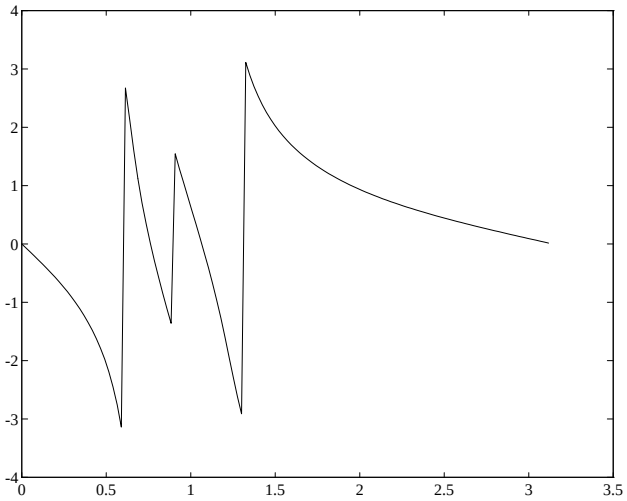

Figure 3: Phase response of the digital filter of Fig. 2.

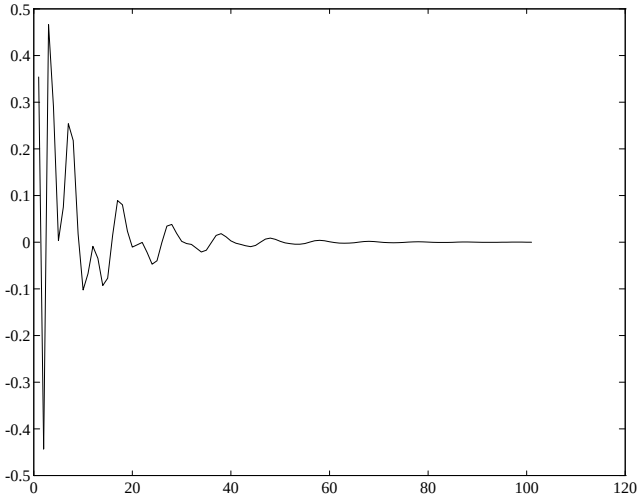

Figure 4: Impulse response of the digital filter of Fig. 2.

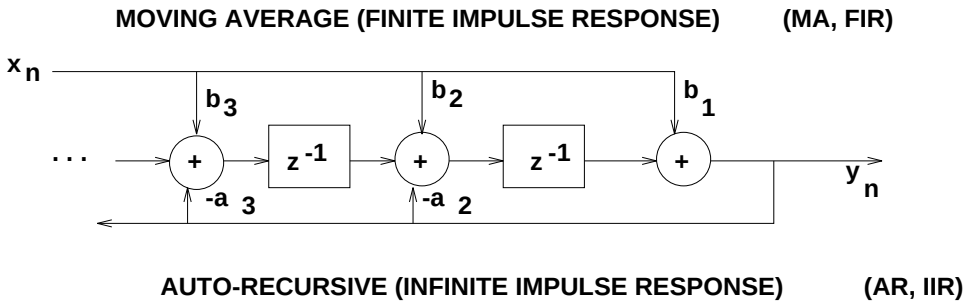

Figure 5: ARMA filter

$$y[n] = b_1 x[n] + b_2 x[n-1] + \cdots + b_{nb+1} x[n-nb] - a_2 y[n-1] - \cdots - a_{na+1} y[n-na], \quad (4)$$

where na, nb are the numbers of AR and MA weights a_i and b_j.

Going over into the transform domain (here it's z-transforms), we are after the transfer function, defined as the z-transform of the impulse response $h[t]$. We know

$$Y(z) = H(z)X(z)$$

by definition, and so from eq. (4) we can just write

$$Y(z) = \frac{b_1 + b_2 z^{-1} + \cdots + b_{nb+1} z^{-nb}}{1 + a_2 z^{-1} + \cdots + a_{na+1} z^{-na}} X(z). \quad (5)$$

The fraction in eq. (5) is a pretty, rational transfer function of the sort you're used to dealing with in control theory. You find the roots of the upper and lower polynomials and those are your zeros and poles. Note that MA filters have all their $a_i = 0$ so all their poles are at zero, hence for instance they are stable.

I'd like to relate the poles and zeros of the transfer function of a controller or a filter to its *frequency response.* A linear filter transforms an input sine wave of a given frequency in two ways: it attenuates or amplifies its *magnitude* and it advances or retards its *phase*. Recall that at the start of this section, Jeff and I were interested in attenuating certain frequencies. You can imagine that if a filter retarded the phase of a periodic signal by 180 degrees, its output would be the sign-reversed input. If the output were to be proportional to the input, as in a P controller, instead it would have exactly the wrong sign for that frequency. Thus phase phenomena can be devastating for control systems. In fact, if you do the math (or just think about it) you'll see that pure delay in a control system is equivalent to pure phase lag, with the phase of higher frequencies affected more than lower ones; this is why delay is devastating for control.

The frequency response can be described as the response to a set of frequency inputs that form a unit circle in the z-plane (Fig. 6). We want to compute the magnitude $M(\omega)$ and the phase $\Theta(\omega)$ for any given position around the unit circle. The math for this

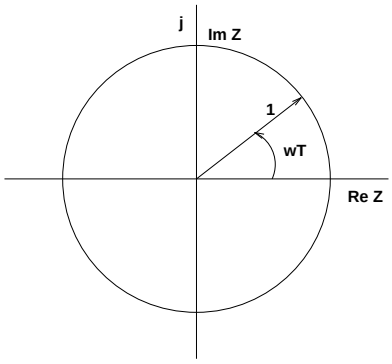

Figure 6: Frequency response as filter response around unit circle. T is the sampling period.

involves expressing the effect of the transfer function for any sinusoidal input as a product of magnitude and phase effects

$$H(e^{j\omega T}) = M(\omega)e^{j\Theta(\omega)},$$

and rewriting the factorized rational H. The details are available in several books or you can work them out yourselves. The result is given by eqs. (6,7). In these equations, there are n zeros and m poles, M_{pi} is a length and ϕ_{pi} is an angle, with semantics shown in Fig. 7. The magnitude of the frequency response is determined by the ratio of distances of poles and zeros from its point on the unit circle, and the phase is determined by the difference of angles as shown. I hope this is some help in visualizing the effect of controllers or filters displayed as pole and zero diagrams.

$$M(\omega) = \frac{H_0 \prod_{i=1}^{n} M_{zi}}{\prod_{1}^{m} M_{pi}} \tag{6}$$

$$\Theta(\omega) = \sum_{i=1}^{n} \phi_{zi} - \sum_{i=1}^{m} \phi_{pi} \tag{7}$$

Finally, I should mention *lattice filters*. These are tapped delay line MA filters, slightly enhanced in the sense that the input signal is split and routed forward along two delay lines, with weighted connections connecting the two lines at each step. The picture is like a ladder with X's for rungs. The weighting coefficients can be "learned" by using the orthogonality criterion (Appendix A), which demands that for an optimal filter the innovation sequence (unexplained part of the signal emerging from the filter) should look like white noise. Anyway it's all easier than it sounds (see `/u/brown/src/pred`). It turns out that the *reflection* or *partial correlation* coefficients in the filter have the same information about the signal as does its power spectrum, so lattice filters are intellectually in the same world as other filters that assume an ensemble of signals characterizable by their first and second order statistics. The lattice filter can be used as a predictor, and will learn to track a sine wave or more complicated periodic signal much better than will a Kalman filter that has the usual (for periodic signals, wrong) assumptions of constant velocity or acceleration signals. Lattice filters have been used at U. Sheffield in the COMODE project [11], and are described in the filtering literature [10; 4; 5].

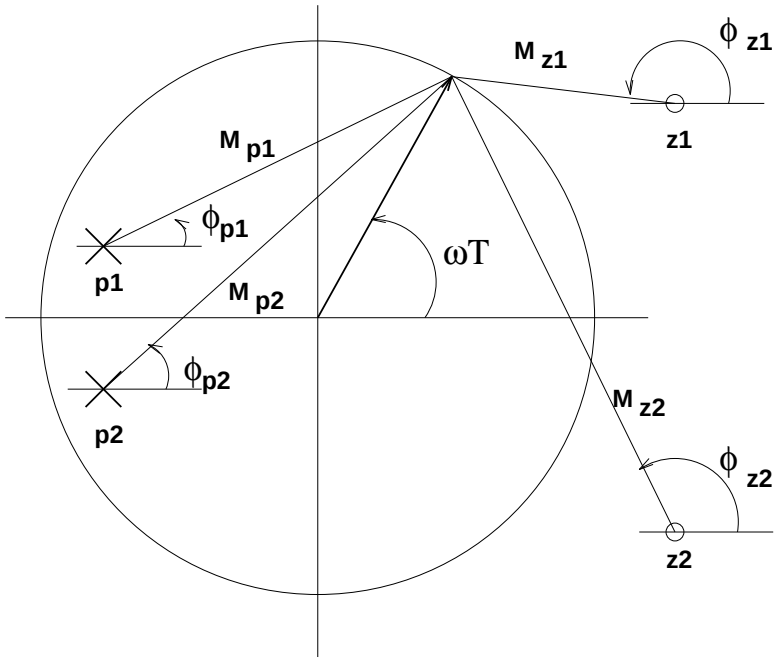

Figure 7: Computing effect of poles and zeros on frequency response amplitude and phase.

5 Recursive Estimation

In the remainder of this tutorial the primal filtering idea is very slightly elaborated. In the techniques to come, we have a prediction of the signal as well as a current input that is corrupted by noise. The new primal idea becomes:

> *Pay more attention to the signal as it overpowers the noise, and pay more attention to the prediction as the noise overpowers the signal.*

You're probably used to *batch* estimation techniques, where for instance you have all your data in a table and you want to construct the best-fit line to that batch of data. However, *recursive* techniques exist. Remember how to compute a running average? If you've got an average $\bar{x}^k$ for the first k terms in the series, the updated average given the $k+1^{th}$ term x^{k+1} is

$$\bar{x}^{k+1} = \bar{x}^k + \frac{1}{k+1}(x^{k+1} - \bar{x}^k). \tag{8}$$

Here the current value of the average serves as our prediction. Congratulations, you're already doing estimation by recursive filtering. As a matter of fact, if you care to look forward to results at the end of this tutorial, such as eq. 30 on page 24 or the Estimate row

of Table 1 on page 28, you'll see that a very close relation of this little formula forms the climax of our story. Thus while we are here we should note the form of this equation. Your new estimate is just the old estimate plus an *innovation* term, which is how much the new data point deviates from your current estimate, weighted by some *gain* that tells you how seriously to take the disagreement, (surprise, or innovation.) Remember these ideas!

R.E. (no, not "Recursively Estimating") Kalman formulated a recursive form of least-squares estimation for possibly time-varying parameters of linear systems like we have been studying in CSC442, under conditions of Gaussian noise. Thus *Kalman Filter* (KF) denotes such an "optimal" filter, but often the pure filter equations are modified, the assumptions aren't met, etc. etc. and the generic idea of linear prediction techniques is often called "KFing" in a very loose way.

In a general KF, the state estimate and data become vectors, the gains become matrices, and the KF also maintains a running account of quality, or how well it is doing, something like the sum of squared innovations. The KF equations simply tell you how to update your guess and its quality (called the *state covariance*) recursively. It turns out that the KF estimate *exactly is* a Gaussian distribution, characterized by a *mean* (the estimate) and a *(co) variance*, as usual. If your plant is not linear, you can apply KF techniques locally to the data by linearizing it around the current guess, and you have an *Extended Kalman Filter*. If the system behavior is time-invariant and kinematic (e.g. it is a moving point with approximately constant velocity (or acceleration)), KFing has a very simple form, called an $\alpha - \beta(-\gamma)$ *filter*.

In every case, it turns out that the form of the estimation equations is basically that of eq. 8; your new estimate is your old estimate plus a weighted innovation.

The reader should know that almost all, if not absolutely all, of the following material is lifted directly, i.e. basically transcribed, from [2]. I'm not really sure what my added value is, but one thing I've done is slightly reorder the presentation. The other classic text on estimation is [3], and I like [5] for its treatment of linear filters, Wiener filters, Lattice Filters, and of course Kalman filters. Horn's book [6] has a nice short section on optimal image restoration that is related to the math that comes up in Wiener filtering. Another classic you should know about is Goodwin and Sin [4].

When do you want to use a (predictive or smoothing or filtering) filter? Whenever your problem fits nicely into a state-estimation formalism: you need a plant model, a gaussian noise model, and you are estimating some plant parameters or state varibles from noisy data. KFs or their kin should always spring to mind when you are tracking some physical system in physical terms...e.g. tracking a centroid of a blob in an image, tracking a point in a scene. Less obviously, KFs are used to track grey-level values in dynamic scenes. In the Electrical Engineering Department at UR, Prof. Murat Tekalp puts a filter on every individual pixel's grey-level to follow its changes in moving scenes.

6 Basic Concepts in Estimation

6.1 Maximum Likelihood and Maximum A Posteriori

Suppose you are estimating a time-invariant parameter x, and you get *measurements* or *observations* in at discrete times indexed by j. Your observations are some function $h(.)$ of the time, the parameter, and some noise process that might also depend on the time.

$$z(j) = h[j, x, w(j)], j = 1, ...$$

Let the collection of k such observations be

$$Z^k \doteq \{z(j), j = 1, ..., k\},$$

and say we want to estimate the value of x with a function

$$\hat{x}(k) = \hat{x}[k, Z^k].$$

There are two ways to think about the time-invariant parameter x.

- There exists a true value but it is unknown. This is a non-random, *non-Bayesian, Fisher* approach.
- There is no single true value; the parameter is a random variable with some (*prior*) PDF $p(x)$. This is a *Bayesian* approach.

Both these estimates should converge in their own ways as $k \to \infty$.

In a Bayesian approach, knowing the prior PDF allows its posterior PDF to be computed from Bayes Rule:

$$p(x \mid Z^k) = \frac{p(Z^k \mid x)p(x)}{p(Z^k)}. \tag{9}$$

For a non-random case there is no prior PDF and we cannot define a posterior one, but we can talk about the *likelihood function*

$$\Lambda_k(x) = p(Z^k \mid x).$$

Thus a common method for estimating nonrandom parameters is the *maximum likelihood* method, which gives the ML estimate, or that value of x which makes the observed measurements most likely:

$$\hat{x}^{ML}(k) \doteq argmax_x p(Z^k \mid x). \tag{10}$$

The corresponding estimate for a random parameter is the *maximum a posteriori* (MAP) estimate

$$\hat{x}^{MAP}(k) \doteq argmax_x p(x \mid Z^k) = argmax_x p(Z^k \mid x)p(x). \tag{11}$$

Note that for the maximization we can ignore the denominator in eq. 9, since it is independent of x.

How do these two estimates differ? In the second you have some prior idea of the answer and in the first you don't. In the second case you are able to make an estimate that combines your knowledge of the prior probability with the measurement you get. In a homey example, say you guess some person's age knowing nothing about her and you get an estimate (say 35). If I then tell you a fact, like she's a grandmother or a high-school student, that might cause you to raise or lower your estimate since the prior age distributions of those groups might not agree with your initial judgment.

In a less homey example, you can work out fairly easily ([2], p. 11) that if you are estimating an unknown parameter x from a measurement z which is just x's value with additive zero-mean noise (say Gaussian), then z just **is** the ML estimate for x. Geometrically, it is the value at the peak of x's PDF since the noise is mean zero. However, you may know something about the parameter x you are trying to estimate. Suppose you know it is drawn from another Gaussian distribution with mean $\bar{x}$ and some variance σ^2, and that the value of x is independent of the measurement noise. Then it turns out that your optimal estimate is a weighted sum of your prior mean $\bar{x}$ and the innovation $(\bar{x} - z)$. And as you might expect, the weight, or gain, depends on the variances of the noise and of the prior knowledge. If the measurement noise has lots more variance than the prior PDF, you trust your measurement less and you weight the innovation less. Contrariwise, if your prior knowledge is only that the true value can vary hugely about its mean, but you have low-variance, high-certainty measurements, you weight the innovation more. We are already seeing duplication of our original running-average idea (eq. 8), and it seems that the results so far are intuitive and not surprising. Suppose you really don't know anything about your parameter? You can still do MAP with what is called a *diffuse prior*. The obvious thing to do is let the variance of your prior PDF go to infinity, so even a Gaussian looks like a uniform density. It turns out, again unsurprisingly, that in this case $\hat{x}^{MAP}$ coincides with $\hat{x}^{ML}$.

6.2 Least Squares and Minimum Mean-Square Error Estimation

For nonrandom parameters, the *least squares* (LS) method takes measurements

$$z(j) = h(j, x) + w(j), j = 1, ... \tag{12}$$

and produces an estimate

$$\hat{x}^{LS}(k) = argmin_x \sum_{j=1}^{k} [z(j) - h(j, x)]^2. \tag{13}$$

If the noises $w(j)$ are independent, identically-distributed (IID), zero-mean Gaussian random variables, then least squares gives the same result as ML for those assumptions.

For random parameters, we wish to minimize the *mean-square error*

$$E[(\hat{x} - x)^2 \mid Z^k], \tag{14}$$

where E is the expectation operator. Thus by definition, the *minimum mean-square error* (MMSE) or *minimum variance* estimate is:

$$\hat{x}^{MMSE} = argmin_{\hat{x}} E[(\hat{x} - x)^2 \mid Z^k]. \tag{15}$$

The solution is the *conditional mean*. Finding the solution is easy; you just differentiate the expected value you're minimizing with respect to $\hat{x}$, set equal to zero, and *voilà*.

$$\hat{x}^{MMSE}(k) = E[x \mid Z^k] = \int x p(x \mid Z^k) dx. \tag{16}$$

This solution is expressed in terms of the conditional PDF (see eq. 9). The estimate is called minimum variance because eq. 16 implies that the mean-square error actually is the variance. This relation between (co)variance and error is basic and must be got used to.

Practically speaking, we may not know $p(x \mid Z^k)$. That's a lot to ask, actually. So we desire a way to estimate that is simple – for instance make an estimate that is a linear combination of the observables. Also we may have to get along with less information than the full PDF. In fact usually one assumes that only its first two moments (mean and variance, basically) can be measured.

7 Gaussian Random Variables

Just a motivational reminder: The Kalman Filter equations we are working toward turn out to be straightforward consequences of these basic facts about matrices and Gaussian random variables. In fact the Kalman Filter produces a Gaussian estimate, which (like all Gaussian random variables) can be described by its first two moments, the mean and the variance. Here the mean is the conditional mean of the data, or the estimate itself, and the variance is the state covariance matrix, or the sum of squared error in the estimate. So we are interested in how Gaussians interact; normally they're pretty clannish, in that Gaussness tends to be preserved across the outcome of all sorts of operations, and we take advantage of that elegance in the Kalman recursive filtering, which may be looked at as creating one Gaussian from another, plus some new information (also Gaussian).

This approach has a bottoms-up feeling I regret, but in a way it is a tops-down approach too. The hope is that after establishing these basics, we can slide downhill, applying them in increasingly complex and specialized situations with increasing ease. Basically I'd like everything that follows to be seen as a corollary of this fundamental stuff.

OK, *vorwärts*! The Gaussian PDF has wonderful properties: the convolution of two Gaussians is Gaussian, Gaussian random variables remain Gaussian under either linear or affine (linear plus a constant) transformations, and on and on. So here we are going to show

that the conditional probability of two Gaussians is Gaussian. First, here is a Gaussian PDF for one random variable, x.

$$p(x) = N(x; \bar{x}, \sigma^2) \doteq \frac{1}{\sqrt{2\pi}\sigma} \exp\left\{ -\frac{(x-\bar{x})^2}{2\sigma^2} \right\}.$$

Now for a vector random variable, there is a parallel but more complex formula. In it, $'$ means "transpose", P is a *covariance matrix*, $|\ .\ |$ means "determinant", and vectors are boldface.

$$N(\mathbf{x}; \hat{\mathbf{x}}, \mathbf{P}) \doteq \mid 2\pi\mathbf{P} \mid^{-\frac{1}{2}} \exp[-\frac{1}{2}(\mathbf{x}-\hat{\mathbf{x}})'\mathbf{P}^{-1}(\mathbf{x}-\hat{\mathbf{x}})].$$

If P is a diagonal, then the components of $\mathbf{x}$ are uncorrelated, and they are independent too, since the joint PDF = the product of marginal densities.

Two vectors $\mathbf{x}$ and $\mathbf{z}$ are *jointly Gaussian* if the stacked vector

$$\mathbf{y} = \begin{bmatrix} \mathbf{x} \\ \mathbf{z} \end{bmatrix}$$

is Gaussian. Then

$$p(\mathbf{x}, \mathbf{z}) = p(\mathbf{y}) = N(\mathbf{y}; \bar{\mathbf{y}}, \mathbf{P_{yy}}),$$

and

$$\bar{\mathbf{y}} = \begin{bmatrix} \bar{\mathbf{x}} \\ \bar{\mathbf{z}} \end{bmatrix}$$

and

$$\mathbf{P_{yy}} = \begin{bmatrix} \mathbf{P_{xx}} & \mathbf{P_{xz}} \\ \mathbf{P_{zx}} & \mathbf{P_{zz}} \end{bmatrix},$$

where

$$\mathbf{P_{xx}} = E[(\mathbf{x}-\bar{\mathbf{x}})(\mathbf{x}-\bar{\mathbf{x}})']$$

$$\mathbf{P_{xz}} = E[(\mathbf{x}-\bar{\mathbf{x}})(\mathbf{z}-\bar{\mathbf{z}})'] = \mathbf{P_{zx}}.$$

Now the conditional PDF of $\mathbf{x}$ given $\mathbf{z}$ is

$$p(\mathbf{x} \mid \mathbf{z}) = \frac{p(\mathbf{x}, \mathbf{z})}{p(\mathbf{z})}. \tag{17}$$

So our problem is to write down the explicit form of eq. 17. Dividing out the normal densities is the same as subtracting their exponents. Defining some intermediate variables:

$$\xi \doteq (\mathbf{x}-\bar{\mathbf{x}})$$

$$\eta \doteq (\mathbf{z}-\bar{\mathbf{z}}),$$

the resulting exponent is (after multiplication by -2)

$$\begin{bmatrix} \xi \\ \eta \end{bmatrix}' \begin{bmatrix} \mathbf{P_{xx}} & \mathbf{P_{xz}} \\ \mathbf{P_{zx}} & \mathbf{P_{zz}} \end{bmatrix}^{-1} \begin{bmatrix} \xi \\ \eta \end{bmatrix} - \eta' \mathbf{P_{zz}^{-1}} \eta.$$

Rewriting the inverse matrix terms $\mathbf{P_{xz}}$ etc as $\mathbf{T_{xz}}$ etc. via the block matrix inverse equations 44 – 47 from Appendix C, we get for instance that

$$\mathbf{T_{xx}^{-1}} = \mathbf{P_{xx}} - \mathbf{P_{xz}} \mathbf{P_{zz}^{-1}} \mathbf{P_{zx}}, \tag{18}$$

$$\mathbf{T_{xx}^{-1}} \mathbf{T_{xz}} = -\mathbf{P_{xz}} \mathbf{P_{zz}^{-1}}. \tag{19}$$

In fact, the exponent can be rewritten as

$$(\xi + \mathbf{T_{xx}^{-1}} \mathbf{T_{xz}} \eta)' \cdot \mathbf{T_{xx}} \cdot (\xi + \mathbf{T_{xx}^{-1}} \mathbf{T_{xz}} \eta),$$

which (you recall) is of the form $mean' \cdot CovarianceInverse \cdot mean$.

This result is a *quadratic form* in $\mathbf{x}$, so the conditional PDF of $\mathbf{x}$ given $\mathbf{z}$ is also Gaussian. If you use eq. 19 to substitute for the $\mathbf{T}$'s in the expression for the mean, and substitute the original definitions of ζ and η as difference of $\mathbf{x}$'s and $\mathbf{z}$'s, you can find the conditional mean of $\mathbf{x}$ given $\mathbf{z}$ to be

$$\hat{\mathbf{x}} \doteq E[\mathbf{x} \mid \mathbf{z}] = \bar{\mathbf{x}} + \mathbf{P_{xz}} \mathbf{P_{zz}^{-1}} (\mathbf{z} - \bar{\mathbf{z}}). \tag{20}$$

Does this last equation look familiar? Although it's just derived from facts about Gaussians it looks like an estimation equation (e.g. see eq. 8 on page 14.) Read on.

The estimation's covariance is (using eq. 18)

$$\mathbf{P_{xx|z}} \doteq E[(\mathbf{x} - \hat{\mathbf{x}})(\mathbf{x} - \hat{\mathbf{x}})' \mid \mathbf{z}] = \mathbf{T_{xx}^{-1}} = \mathbf{P_{xx}} - \mathbf{P_{xz}} \mathbf{P_{zz}^{-1}} \mathbf{P_{zx}}. \tag{21}$$

8 Linear Estimates – Static Case

8.1 Gaussian Random Vectors

OK enough background, let's get to work. Let's make an estimate that is a linear combination of the measurements, in the case that the parameters we are estimating do not vary with time.

Here we simply copy eqs. 20 and 21 over from the last section. The MMSE estimate of two jointly Gaussian random vectors $\mathbf{x}$ (to be estimated) and $\mathbf{z}$ (the measurements) is the conditional mean, given by

$$\hat{\mathbf{x}} \doteq E[\mathbf{x} \mid \mathbf{z}] = \bar{\mathbf{x}} + \mathbf{P_{xz}} \mathbf{P_{zz}^{-1}} (\mathbf{z} - \bar{\mathbf{z}}). \tag{22}$$

We know the corresponding conditional error covariance to be:

$$\mathbf{P}_{\mathbf{xx}|\mathbf{z}} \doteq E[(\mathbf{x}-\hat{\mathbf{x}})(\mathbf{x}-\hat{\mathbf{x}})' \mid \mathbf{z}] = \mathbf{T}_{\mathbf{xx}}^{-1} = \mathbf{P}_{\mathbf{xx}} - \mathbf{P}_{\mathbf{xz}}\mathbf{P}_{\mathbf{zz}}^{-1}\mathbf{P}_{\mathbf{zx}}. \tag{23}$$

Clearly, eq. 22 is in a familiar form: it fits the mold of all our estimation equations. It says that the MMSE estimate is just the mean of $\mathbf{x}$ plus a term that is proportional to the innovation in $\mathbf{z}$ — the departure of $\mathbf{z}$ from its expected value. The constant of proportionality (the gain) depends directly on the variation of $\mathbf{x}$ with $\mathbf{z}$ and inversely on the variance of $\mathbf{z}$ itself. The gain term scales the contribution and attenuates the contribution of measurements known to have large variance about their mean value.

Eq. 23 says that the variance goes down as you take measurements. It goes down by the inverse of the variation of measurements (highly varying measurements don't help much), and proportional to the square of the covariance of the measurements with $\mathbf{x}$.

8.2 Least-Squares Estimation

Here the problem is to estimate, according to the criterion of eq. 13, an unknown, nonrandom, constant vector $\mathbf{x}$ from the measurements

$$\mathbf{z}(i) = \mathbf{H}(i)\mathbf{x} + \mathbf{w}(i), i = 1, 2, .., k$$

Our job is thus to develop an estimate $\hat{\mathbf{x}}$ that minimizes the squared error, which is

$$\begin{aligned} J(k) &= \sum_{i=1}^{k} [\mathbf{z}(i) - \mathbf{H}(i)\mathbf{x}]'\mathbf{R}^{-1}[\mathbf{z}(i) - \mathbf{H}(i)\mathbf{x}] \\ &= [\mathbf{z}^k - \mathbf{H}^k\mathbf{x}]'(\mathbf{R}^k)^{-1}[\mathbf{z}^k - \mathbf{H}^k\mathbf{x}] \end{aligned}$$

where

$$\mathbf{H}^k = \begin{bmatrix} \mathbf{H}(1) \\ \vdots \\ \mathbf{H}(k) \end{bmatrix}$$

is a stacked measurement matrix,

$$\mathbf{z}^k = \begin{bmatrix} \mathbf{z}(1) \\ \vdots \\ \mathbf{z}(k) \end{bmatrix} = \mathbf{H}^k\mathbf{x} + \mathbf{w}^k$$

is the stacked vector of measurements, similarly $\mathbf{w}^k$ is the stacked vector of measurement errors, and

$$\mathbf{R}^k = \begin{bmatrix} \mathbf{R}(1) & \dots & \mathbf{0} \\ \vdots & & \vdots \\ \mathbf{0} & \dots & \mathbf{R}(k) \end{bmatrix}$$

is a block-diagonal, positive-definite matrix. Here we're assuming $\mathbf{x}$ is an unknown constant, and the $\hat{\mathbf{x}}(k)$ is a random variable if the measurement errors are modeled as random. The

LS criterion assumes the errors are zero-mean and independent, with covariance $\mathbf{R}(i)$. If they are Gaussian, then minimizing the LS error criterion is equivalent to maximizing the likelihood function

$$\Lambda(\mathbf{x}) = p[\mathbf{z}^k \mid \mathbf{x}] = \prod_{i=1}^{k} p[\mathbf{z}(i) \mid \mathbf{x}],$$

so the LS and ML estimates are the same for Gaussian disturbances.

Further we have seen this sort of calculation before every time we do a least squares problem (e.g. in the appendix of [1]). To work through it, you set $J(k)$'s gradient to zero:

$$\nabla_{\mathbf{x}} J(k) = 2(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}[\mathbf{z}^k - \mathbf{H}^k\mathbf{x}] = 0,$$

so

$$\hat{\mathbf{x}}(k) = [(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}\mathbf{H}^k]^{-1}(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}\mathbf{z}^k. \tag{24}$$

If the noises are independent, zero-mean random variables with covariances $\mathbf{R}(i)$, the LS estimate is unbiased; the expected value of $\hat{\mathbf{x}}(k)$ is $\mathbf{x}$. Further, the estimation error

$$\tilde{\mathbf{x}} = \mathbf{x} - \hat{\mathbf{x}}(k) = -[(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}\mathbf{H}^k]^{-1}(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}\mathbf{w}^k.$$

Thus, finally, the mean-square error or *covariance of the estimate* is (after a little algebra)

$$\mathbf{P}(k) \doteq E[\tilde{\mathbf{x}}(k)\tilde{\mathbf{x}}'(k)] = [(\mathbf{H}^k)'(\mathbf{R}^k)^{-1}\mathbf{H}^k]^{-1}. \tag{25}$$

This is telling us that if the noise has high variance, $\mathbf{P}(k)$ will be big.

8.3 Least Squares Estimation Example

We're given noisy observations of a scalar x. Our observations are

$$z(i) = x + w(i), i = 1, 2, ...k$$

For a batch LS formulation like we've just gone through, the measurement matrix is simple since we're observing x every time:

$$\mathbf{H}^k = \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix}$$

and let

$$\mathbf{R}^k = \sigma^2\mathbf{I},$$

with $\mathbf{I}$ the $k \times k$ identity matrix. We just substitute into eq. 24 and get our estimate:

$$\hat{x}(k) = \left\{ [1 \ldots 1](\sigma^2\mathbf{I})^{-1} \begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} \right\}^{-1} [11 \ldots 1](\sigma^2\mathbf{I})^{-1} \begin{bmatrix} z(1) \\ z(2) \\ \vdots \\ z(k) \end{bmatrix} = \frac{1}{k}\sum_{i=1}^{k} z(i).$$

So $\hat{x}(k)$ is the *sample mean* of the observations, which makes sense. The variance of this estimate is, by eq. 25,

$$P(k) = \frac{\sigma^2}{k},$$

which is what we expect: it goes down with the number of measurements.

8.4 Recursive Least Squares Estimation

Just to take stock for a minute, we are working through lots of well- known least-squares estimation techniques in a rather general vector formulation that is generalizable easily to recursive (iterative, one measurement at a time) reformulation. When we get to the end of this subsection, we are basically done.

Now we are going to formulate our batch least-squares estimator of eq. 24 and eq. 25 recursively by stacking the past observations and the current one, likewise the measurement matrices, noises, and covariances like this:

$$\mathbf{z}^{k+1} = \begin{bmatrix} \mathbf{z}^k \\ \mathbf{z}(k+1) \end{bmatrix}$$

$$\mathbf{H}^{k+1} = \begin{bmatrix} \mathbf{H}^k \\ \mathbf{H}(k+1) \end{bmatrix}$$

$$\mathbf{w}^{k+1} = \begin{bmatrix} \mathbf{w}^k \\ \mathbf{w}(k+1) \end{bmatrix}$$

$$\mathbf{R}^{k+1} = \begin{bmatrix} \mathbf{R}^k & \mathbf{0} \\ \mathbf{0} & \mathbf{R}(k+1) \end{bmatrix}.$$

We want to work out the recursive form of our estimate and also we need to do the accumulated covariance, since our Gaussness demands both moments to be calculated and brought along. First the covariance (eq. 25), which we need to work out at time $k+1$. It is expressed recursively as

$$\mathbf{P}^{-1}(k+1) = (\mathbf{H}^{k+1})'(\mathbf{R}^{k+1})^{-1}\mathbf{H}^{k+1} = \mathbf{P}^{-1}(k) + \mathbf{H}'(k+1)\mathbf{R}^{-1}(k+1)\mathbf{H}(k+1).$$

You get this by writing out the stacked forms and working it through. One interpretation of the inverse covariance is (Fisher) *information*, and this says that the information at time $k+1$ is the information at time k plus the new information gained at time $k+1$ from $\mathbf{z}(k+1)$.

Now the derivation gets a little long-winded so I'm going to wave my hands a bit. The matrix inversion lemma (Appendix C) is used to get a rather large expression for the new covariance $\mathbf{P}(k+1)$, which I shall spare you, but which is written compactly in terms of two new concepts, the *measurement prediction covariance* $\mathbf{S}$(k+1) and the *gain* $\mathbf{W}$(k+1) (nothing to do with lower case $\mathbf{w}$, the noises), defined as follows:

$$\mathbf{S}(k+1) \doteq \mathbf{H}(k+1)\mathbf{P}(k)\mathbf{H}'(k+1) + \mathbf{R}(k+1) \tag{26}$$

$$\mathbf{W}(k+1) \doteq \mathbf{P}(k)\mathbf{H}'(k+1)\mathbf{S}^{-1}(k+1). \tag{27}$$

As an aside, these pesky $\mathbf{H}$ matrices are to convert from state variables to measurement variables, which is why they keep cropping up. Ignoring $\mathbf{H}$ in the equation for the gain, you recognize a gain very much like that in eq. 20, which is again inversely proportional to the measurement covariance. So you're still on familiar ground.

Using these two definitions, the recursive equation for the covariance may be written neatly as

$$\mathbf{P}(k+1) = \mathbf{P}(k) - \mathbf{W}(k+1)\mathbf{S}(k+1)\mathbf{W}'(k). \tag{28}$$

or perhaps even more neatly as

$$\mathbf{P}(k+1) = [\mathbf{I} - \mathbf{W}(k+1)\mathbf{H}(k+1)]\mathbf{P}(k+1). \tag{29}$$

OK, now to figure out the recursive form of the estimate. We start with the batch form of the estimate (eq. 24), replace k by $k+1$ in that equation and eq. 25, do some not too obvious rewriting of a few terms to get rid of $\mathbf{H}^{k+1}$ and $\mathbf{R}^{k+1}$, and the final result is

$$\begin{aligned} \hat{\mathbf{x}}(k+1) &= \mathbf{P}(k+1)(\mathbf{H}^{k+1})'(\mathbf{R}^{k+1})\mathbf{z}^{k+1} \\ &\cdots \\ &= \hat{\mathbf{x}}(k) + \mathbf{W}(k+1)[\mathbf{z}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k)]. \end{aligned} \tag{30}$$

If you're still conscious at all, you'll see that Here we are again. The new estimate is equal to the previous one plus a correction term. The correction depends on the innovation, or the difference between the observation and the predicted value of the observation based on what we knew at the time before, at time k. The innovation is weighted by a gain $\mathbf{W}$. Shades of the "running average"!

Equations 26, 27, 28, and 30 are our recursive least-squares estimator.

8.5 Recursive Least Squares Estimator Example

Let's do the previous least-squares estimation problem recursively. We take from Section 8.3 the values of $\mathbf{H}^k$ and $\mathbf{R}^k$, which tell us about the measurements (we measure x) and variance of the noise (each measurement is disturbed by a random process whose variance is σ^2). Look back if you don't believe me. We computed the value of $\mathbf{P}(k)$ there...that is the declining variance of our estimate, which drops proportionally with the number of measurements we make. We need to compute our prediction covariance:

$$S(k+1) = \mathbf{HPH}' + \mathbf{R} = 1 \cdot \frac{\sigma^2}{k} \cdot 1 + \sigma^2 = \sigma^2(\frac{k+1}{k}).$$

Likewise we work out the gain:

$$W(k+1) = \mathbf{PH}'\mathbf{S}^{-1} = \frac{\sigma^2}{k} \cdot 1 \cdot (\frac{k+1}{k}\sigma^2)^{-1} = \frac{1}{k+1}.$$

Finally, using eq. 30:

$$\hat{x}(k+1) = \hat{x} + \frac{1}{k+1}[z(k+1) - \hat{x}(k)].$$

If you want to check this last, *deja-vu*-ridden equation, you can get it from simple manipulation of the batch equation...just write $k+1$ for k in the batch solution and work it out (hint: you have to add zero in the form of plus and minus $\hat{x}(k)$).

9 The Kalman Filter

The Kalman filter extends least-squares estimate to *time-varying* quantities. In that sense it is a generalization of Wiener filters, ARMA filters, lattice filters, and other linear estimators. All these latter sorts of filters assume time-invariant (in statistical terms, *stationary*) processes. The Kalman filter description given below (with some notable gaps in the derivation) is the discrete version. It turns out that the discrete KF mathematics introduces terms that complicate the semantics of the filter, which (you can bet) is actually just the same as all the other least-squares estimators we've seen so far. To get us started, I'd like to jump to the punch-line for the *continuous* Kalman filter, which can be derived from the discrete formulation by taking a few limits (e.g. see [3]).

In the continuous formulation the system of interest is modeled

$$\dot{\mathbf{x}} = \mathbf{F}\mathbf{x} + \mathbf{G}\mathbf{w},$$

$$\mathbf{z} = \mathbf{H}\mathbf{x} + \mathbf{v}$$

with the usual semantics (**v**,**w** noise processes with covariance matrices **R** and **Q**).

The equation for the (here, derivative of the) estimate is (again, this is without-proof preview just to show the non-mysterious semantics of the filter)

$$\dot{\hat{\mathbf{x}}} = \mathbf{F}\hat{\mathbf{x}} + \mathbf{P}\mathbf{H}'\mathbf{R}^{-1}[\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}]. \tag{31}$$

This equation is similar in form and semantics to eqs. 23 and 27.

The propagation of covariance is, again as a differential equation, expressed by

$$\dot{\mathbf{P}} = \mathbf{F}\mathbf{P} + \mathbf{P}\mathbf{F}' + \mathbf{G}\mathbf{Q}\mathbf{G}' - \mathbf{P}\mathbf{H}'\mathbf{R}^{-1}\mathbf{H}\mathbf{P}. \tag{32}$$

The first two terms involving **F** result from accumulation of covariance from the homogeneous, unforced dynamic system before any measurements are taken. The term in **Q** accounts for the increase of uncertainty due to process noise in the system, and the last term measures the decrease in uncertainty as a result of measurements. The quadratic dependence on the unknown **P** in eq. 32 makes it by definition a (matrix) *Riccati equation*. The Riccati equation turns up in optimal control too, because of the quadratic, "squared error" dependencies we are interested in.

Now let us turn to the discrete Kalman filter. The time-varying systems considered are the familiar linear, discrete-time dynamic systems. Thus the *plant equation* is

$$\mathbf{x}(k+1) = \mathbf{F}(k)\mathbf{x}(k) + \mathbf{G}(k)\mathbf{u}(k) + \mathbf{v}(k), \tag{33}$$

where $\mathbf{x}(k)$ is the state at time k, $\mathbf{u}(k)$ is the (known) input or control signal, and $\mathbf{v}(k)$ is a sequence of zero-mean, white, Gaussian process noise with covariance $\mathbf{Q}(k)$. The measurement equation is

$$\mathbf{z}(k) = \mathbf{H}(k)\mathbf{x}(k) + \mathbf{w}(k)$$

where $\mathbf{w}(k)$ is a sequence of zero-mean, white, Gaussian process noise with covariance $\mathbf{R}(k)$. The initial $\mathbf{x}(0)$ is assumed to be Gaussian with mean $\hat{\mathbf{x}}(0 \mid 0)$ and covariance $\mathbf{P}(0 \mid 0)$. The two noise processes are assumed independent. All this Gaussness is useful because we know (from Section 7) that the PDF of a Gaussian conditioned on a Gaussian is Gaussian again, and we are looking for the conditional mean of the state given the measurements. Thus the estimation preserves the Gaussian nature of the estimate.

One cycle of the estimation algorithm starts with the current estimate

$$\hat{\mathbf{x}}(k \mid k) \doteq E[\mathbf{x}(k) \mid Z^k],$$

(that is the conditional mean of the state given the measurements) and the "how're we doing?" quantity, the state error covariance (i.e. the squared error) matrix

$$\mathbf{P}(k \mid k) \doteq E\{[\mathbf{x}(k) - \hat{\mathbf{x}}(k \mid k)][\mathbf{x}(k) - \hat{\mathbf{x}}(k \mid k)]' \mid Z^k\}$$

The cycle derives the corresponding variables at the next stage, which are written $\hat{\mathbf{x}}(k+1 \mid k+1)$ and $\mathbf{P}(k+1 \mid k+1)$. We only need the estimate (mean) and the covariance to characterize our Gaussian estimate. One of the neat things about the KF is that all the memory it needs is provided by the current estimate and covariance.

Just to get a bit more practice with the notation, the *one-step prediction* of the state is written $\hat{\mathbf{x}}(k+1 \mid k)$, the *predicted measurement* is $\mathbf{z}(k+1 \mid k)$, the *one-step prediction covariance* is $\mathbf{P}(k+1 \mid k)$.

The mathematical derivation of the state estimate and its covariance follow from substituting all the dynamic system generality into equations 20 and 21. For details, see any book, but the answer is pretty succinct.

Let us specialize away much of the generality of the Kalman filter to see how it works for a time-invariant prediction problem. Suppose we have a single measurement that gives us a linear combination of the state vector components

$$z_i = \mathbf{a}_i \cdot \mathbf{x}_i.$$

Let the variance associated with measurement i be σ_i^2. Suppose our current estimate of the state is $\check{\mathbf{x}}_i$ and its covariance is $\check{\mathbf{P}}_i$. We want to produce the new estimate $\hat{\mathbf{x}}_i$ and the updated covariance $\hat{\mathbf{P}}_i$. (This notation is different from the $(k+1|k)$ indices but we don't need all that power here.)

The Kalman gain matrix is

$$\mathbf{W}_i = \check{\mathbf{P}}_i \mathbf{a}_i (\mathbf{a}_i' \check{\mathbf{P}}_i \mathbf{a}_i + \sigma_i^2)^{-1}.$$

The Kalman filter increments the state estimate by adding the weighted residual or innovation

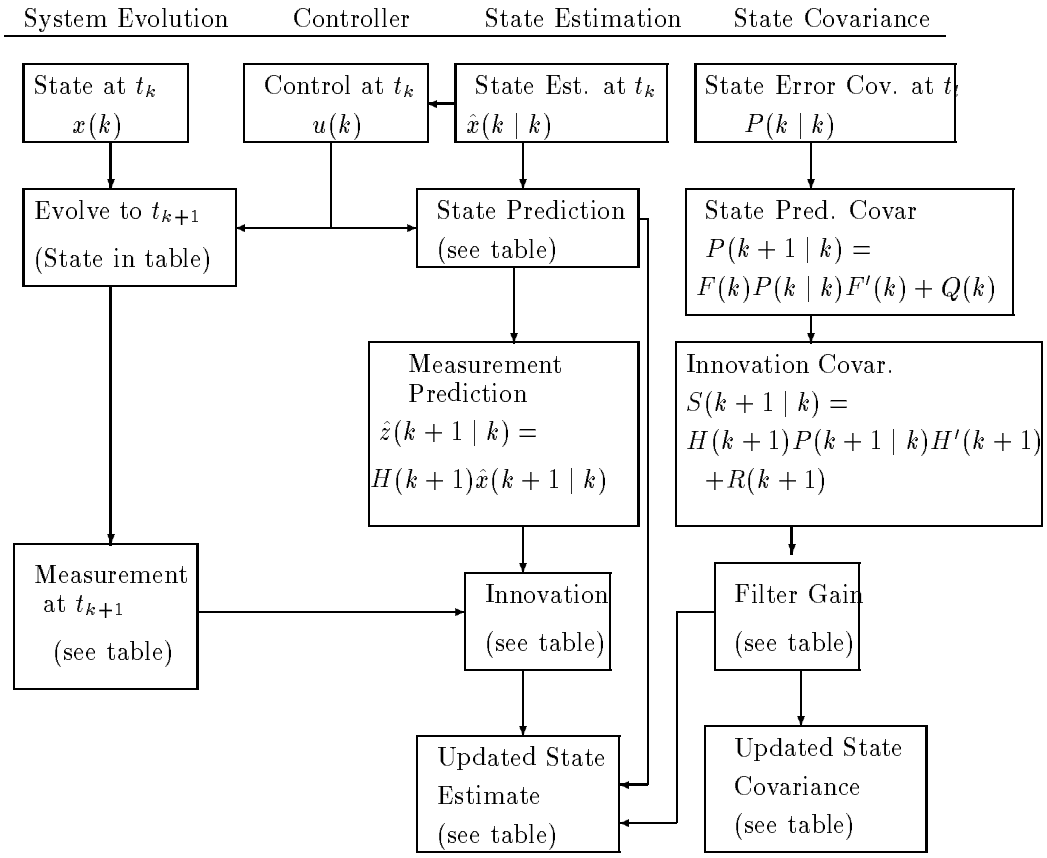

Figure 8: The Discrete Kalman Filter State Estimator for Linear Systems

$$\hat{\mathbf{x}}_i = \check{\mathbf{x}}_i + \mathbf{W}_i(z_i - \mathbf{a}_i \cdot \check{\mathbf{x}}_i)$$

and decrements the covariance matrix

$$\hat{\mathbf{P}}_i = \check{\mathbf{P}}_i - \mathbf{W}_i \mathbf{a}_i' \check{\mathbf{P}}_i.$$

A more interesting example (and code) is given in Appendix E. The full-blown algorithm is shown in Fig. 8, which has a few equations to define intermediate steps and also refers to Table 1 for the rest of the relevant equations.

The strong relation of the Kalman Filter to the static least square estimator is brought out in Table 1. The equations on the left are all copies from Section 8.4. The control input $\mathbf{u}$ is known, and so its effects are only seen in the state prediction step and do not appear elsewhere.

As a practical matter, *initialization* of a Kalman filter (or an $\alpha - \beta$ filter, see below) is a problem. For constant acceleration or constant velocity plants with observable positions, one can just initialize estimates with an average position difference (velocity) or second

Quantity	Least Squares	Kalman Filter
State	$\mathbf{x}$ constant	$\mathbf{x}(k+1) = \mathbf{F}(k)\mathbf{x}(k) + \mathbf{G}(k)\mathbf{u}(k) + \mathbf{v}(k)$
Noise Covar.	$\mathbf{R}$	$\mathbf{R}$
State Predict.	$\hat{\mathbf{x}}$	$\hat{\mathbf{x}}(k+1 \mid k) = \mathbf{F}(k)\hat{\mathbf{x}}(k \mid k) + \mathbf{G}(k)\mathbf{u}(k)$
Meas'ment	$\mathbf{z}(k) = \mathbf{H}(k)\mathbf{x} + \mathbf{w}(k)$	$\mathbf{z}(k) = \mathbf{H}(k)\mathbf{x}(k) + \mathbf{w}(k)$
Innovation	$\nu(k+1) = \mathbf{z}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k)$	$\nu(k+1) = \mathbf{z}(k+1) - \mathbf{H}(k+1)\hat{\mathbf{x}}(k+1 \mid k)$
Gain	$\mathbf{W}(k+1) = \mathbf{P}(k)\mathbf{H}'(k+1)\mathbf{S}^{-1}(k+1)$	$\mathbf{W}(k+1) = \mathbf{P}(k+1 \mid k)\mathbf{H}'(k+1)\mathbf{S}^{-1}(k+1)$
Estimate	$\hat{\mathbf{x}}(k+1) = \hat{\mathbf{x}}(k) + \mathbf{W}(k+1)\nu(k+1)$	$\hat{\mathbf{x}}(k+1 \mid k+1) = \hat{\mathbf{x}}(k+1 \mid k) + \mathbf{W}(k+1)\nu(k+1)$
State Covar.	$\mathbf{P}(k+1) = [\mathbf{I} - \mathbf{W}(k+1)\mathbf{H}(k+1)]\mathbf{P}(k)$	$\mathbf{P}(k+1) = [\mathbf{I} - \mathbf{W}(k+1)\mathbf{H}(k+1)]\mathbf{P}(k+1 \mid k)$

Table 1: Comparing LS to KF estimation.

difference (acceleration). As another practical matter, coming up with a time-varying plant model that can be synchronized with the unknown one is often difficult. Time-invariance simplifies the mathematics and assumptions, and often is a practical way out. Possibly the biggest advantage of Kalman (or recursive LS) filtering is the running covariance estimate, which allows you to put "error ellipses" around your estimate. So if you can simplify much of the machinery that is fine, just don't throw the baby out with the bath.

A practical issue is the matrix inverses that appear in the KF calculation. These can cause numerical problems and so often an LU decomposition technique is employed...the resulting method is known as a *square root* method.

Of course one of the paradigmatical state-estimation problems is that of an observer in a control system. As you probably know, a feedback control system is often formulated as feeding back state information. The control system often computes the difference between the desired state and the current one and emits a signal to the plant accordingly. Practically speaking, one can often not measure the state directly. For instance, one may be interested in speed but can only measure position, or in position but can only measure acceleration, or in temperature but can only measure spectrum of emitted light. Well, this calls for an $\mathbf{H}$ matrix, or its inverse...we need to estimate the state from these indirect readings. So KF is just the ticket here, and the $\mathbf{H}$ matrix relates the observables to the state. The normal and familiar considerations of *observability* come in here (e.g. see [7]). You have to have enough of the right kind of measurements to describe the state.

KFs have an interesting dual relationship with optimal control, which tries to minimize some measure of squared error over time as the system to be controlled deviates from the desired trajectory. Generally filtering and control techniques are closely related: controllers are just filters really: feedforward control is a form of moving average (MA) filter, and feedback provides autorecursive (AR) filtering. Both filters and controllers are often characterized by their frequency response, which (for the rational algebraic transfer functions you see in linear theory) is determined by the *poles* and *zeros* of the transfer function (Section 4). Kalman filtering goes beyond these models because it allows the plant model to vary with time.

The KF can be derived from the point of view of the orthogonality principle. I won't say much here, but consider looking at the sequence of innovations produced by the filter as a random process to be studied in its own right. It makes sense that if this sequence of "surprises" has too much structure in it then you're doing something wrong. For instance suppose it is not mean 0. Then you'd clearly want to change your state estimate until it **was** mean zero, no? Similar with other kinds of structure, like in particular its second moment, or autocorrelation. You want your innovation sequence also to be white noise. Optimal filters, such as the Kalman, drive the innovation sequence to look like white noise.

You can thus check the innovations sequence for clues about "how your model is doing". For example if you are using a constant-velocity model to track some airplane, you can notice when your performance is not consistent with this assumption and change your dynamic model in the filter. For instance, you could substitute a constant-acceleration model. Many books (e.g [5]) unify their treatment of all filters around this "innovations process" approach, so be warned.

10 A More Accessible Alternative: Time-Invariant KFs

10.1 The $\alpha - \beta$ filter

In a time-invariant filter, plant variations through time are accommodated by modeling them as noise. This is a bit of a hack but it is often the most realistic assumption if the variations are unknown to you in advance. Also, the equations are a lot simpler.

Linear dynamical systems with *time-invariant* coefficients in their state transition and measurement equations lead to simpler optimal estimation techniques than are needed for the time-varying case. The state estimation covariance and filter gain matrices achieve steady-state values that can often be computed in advance. Two common time-invariant systems are constant-velocity and constant-acceleration systems, so called *kinematic systems*.

Let us assume a *constant velocity* model: starting with some initial value, the object's velocity evolves through time by *process noise* of random accelerations, constant during each sampling interval but independent. With no process noise the velocity is constant; process noise can be used to model unknown maneuverings of a non-constant velocity target. The cumulative result of the accelerations can in fact change the object's velocity arbitrarily much, so we model a maneuvering object as one with high process noise. We assume position measurements only are available, subject to measurement noise of constant covariance. Clearly the more that is known *a priori* about the motion the better the predictions will be. Some sensors or techniques can provide retinal or world velocity measurements as well.

Assume the object state (its position and velocity) evolves independently in each of the (X, Y, Z) dimensions. For instance, in the Y dimension, it evolves according to

$$\mathbf{y}(k+1) = \mathbf{F_y}\mathbf{y}(k) + \mathbf{v}(k), \tag{34}$$

where

$$\mathbf{F_y} = \begin{bmatrix} 1 & \Delta t \\ 0 & 1 \end{bmatrix} \tag{35}$$

for sampling interval Δt, error vector $\mathbf{v}(k)$, and $\mathbf{y} = [Y, \dot{Y}]^T$. The equations for the other two spatial dimensions are similar, and in fact have identical $\mathbf{F}$ matrices. Thus for the complete object state $\mathbf{x} = [X, \dot{X}, Y, \dot{Y}, Z, \dot{Z}]^T$, $\mathbf{F}$ is a (6×6) block-diagonal matrix whose blocks are identical to $\mathbf{F_y}$. The error vector $\mathbf{v}(k)$ can be described with a simple covariance structure: $E(\mathbf{v}(k)\mathbf{v}^T(j)) = \mathbf{Q}\delta_{kj}$.

The $\alpha - \beta$ filter for state prediction has the form

$$\hat{\mathbf{x}}(k+1|k+1) = \hat{\mathbf{x}}(k+1|k) + \begin{bmatrix} \alpha \\ \beta/\Delta t \end{bmatrix} [\mathbf{z}(k+1) - \hat{\mathbf{z}}(k+1|k)], \tag{36}$$

where $\hat{\mathbf{x}}(k+1|k+1)$ is an updated estimate of $\mathbf{x}$ given $\mathbf{z}(k+1)$, the measurement at time $k+1$. Here we assume that $\mathbf{z}(k+1)$ consists of the three state components (X, Y, Z) (but not $(\dot{X}, \dot{Y}, \dot{Z})$). The state estimate is a weighted sum of a state $\hat{\mathbf{x}}(k+1|k)$ *predicted* from the last estimate to be $\mathbf{F}\hat{\mathbf{x}}(k|k)$ and the *innovation*, or difference between a predicted measurement and the actual measurement. The predicted measurement $\hat{\mathbf{z}}(k+1|k)$ is produced by applying (here a trivial) measurement function to the predicted state.

The $\alpha - \beta$ filter is a special case of the Kalman filter. For our assumptions, the optimal values of α and β can be derived (see [2], for example) and depend only on the ratio of the process noise standard deviation and the measurement noise standard deviation. This ratio is called the object's *maneuvering index* λ, and with the piecewise constant process noise we assume,

$$\alpha = -\frac{\lambda^2 + 8\lambda - (\lambda+4)\sqrt{\lambda^2 + 8\lambda}}{8} \tag{37}$$

and

$$\beta = \frac{\lambda^2 + 4\lambda - \lambda\sqrt{\lambda^2 + 8\lambda}}{4}. \tag{38}$$

The state estimation covariances can be found in closed form as well, and are simple functions of α, β, and the measurement noise standard deviation.

10.2 The α - β - γ Filter

The α - β - γ filter is like the α - β filter only based on a uniform acceleration assumption. Thus it makes a quadratic prediction instead of a linear one. Broadly, it tends to be more sensitive to noise but better able to predict smoothly varying velocities. Its equation is the following.

$$\hat{\mathbf{x}}(k+1|k+1) = \hat{\mathbf{x}}(k+1|k) + \begin{bmatrix} \alpha \\ \beta/\Delta t \\ \gamma/\Delta t^2 \end{bmatrix} [\mathbf{z}(k+1) - \hat{\mathbf{z}}(k+1|k)], \tag{39}$$

With the maneuvering index λ defined as before, the optimal α and β for the case that the target experiences random small changes in acceleration (random jerks) are the same as before and the optimal $\gamma = \beta^2/\alpha$.

Both the α - β and α - β - γ filters have been implemented as C++ classes, in versions with uniform and nonuniform timesteps. The nonuniform timestep versions are easy extensions in which the timestep is calculated at each iteration as the difference of the last two timestamps. Code for various α - β and α - β - γ filters is available from CB and is only a couple of pages long. Appendix D gives some code for the filter.

11 References

[1] D. Ballard and C. Brown. *Computer Vision.* Prentice-Hall, 1982.

[2] Y. Bar-Shalom and T. E. Fortmann. *Tracking and Data Association.* Academic Press, 1988.

[3] Arthur C. Gelb. *Applied Optimal Estimation.* The MIT Press, 1974.

[4] G. C. Goodwin and K. S. Sin. *Adaptive Filtering, Prediction and Control.* Prentice-Hall, 1984.

[5] Simon Haykin. *Modern Filters.* MacMillan, 1989.

[6] Berthold K.P. Horn. *Robot Vision.* MIT-Press, McGraw-Hill, 1986.

[7] D. G. Luenberger. *Introduction to Dynamic Systems.* John Wiley and Sons, 1979.

[8] W.K. Pratt. *Digital Image Processing, 2nd ed.* Wiley, 1991.

[9] W.H. Press, B. P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[10] B. Widrow and S. D. Stearns. *Adaptive Signal Processing.* Prentice-Hall, 1985.

[11] Y. Zheng, J. E. W. Mayhew, S. A. Billings, and J. P. Frisby. Lattice predictor for 3-d vision and intelligent tracking. Technical report, University of Sheffield, 1991.

A The Orthogonality Principle

Here is a fundamental and simple fact about estimation, which can be used to derive lots of the results in a slightly different way. Quite often we are going to construct our estimate out of a *linear combination* of some of our measurements (say all of them in a past finite interval). So what does this imply? It implies that our estimate must lie in the hyperplane spanned by those measurements. But the real value may not lie in this hyperplane. It is easy to see that the best linear estimate (the one that minimizes the difference between the real value and the estimate) is a point on the hyperplane that is the orthogonal projection of the real value onto the plane. In other words,*the error is orthogonal to the estimate.* By more or less accessible mathematics, it turns out that the best (in a MMSE sense) linear estimate of a random variable in terms of another, observable random variable is such that

- The estimate is unbiased: the error has mean zero.
- The estimation error is uncorrelated from the observables: they are orthogonal.

For image restoration, the consequences of the orthogonality principle are simple. Consider the image and our observations of it to be written as column vectors; just scan them out column-wise. Let the observation vector be $\mathbf{g}$ and the image be $\mathbf{f}$.

First, the expected value of the image estimate $\hat{\mathbf{f}}$ must equal the expected value of the image:

$$E[\hat{\mathbf{f}}] = E[\mathbf{f}] \tag{40}$$

Second, the error in the restoration must be orthogonal to the observation about its mean:

$$E[(\hat{\mathbf{f}} - \mathbf{f})(\mathbf{g} - E[\mathbf{g}])'] = 0 \tag{41}$$

Here the prime means matrix transpose, as elsewhere in this document.

This result is of practical use in thinking about and implementing other forms of linear estimators, such as the *lattice filter*, and figures heavily in Kalman's 1960 paper. It provides an elegant tool for reasoning about least-squared error solutions, as in the derivation of the Wiener Estimator (Section 3).

B Singular Value Decomposition

Any $M \times N$ matrix F of rank R can be decomposed into the sum of a weighted set of unit-rank $M \times N$ matrices by SVD. The decomposition (See [9] or better its 2nd edition or MATLAB) yields

$$\mathbf{F} = \mathbf{U}\mathbf{\Lambda}^{1/2}\mathbf{V}' \tag{42}$$

It turns out that the columns of $\mathbf{U}$ are the eigenvectors of the symmetric matrix $\mathbf{F}\mathbf{F}'$, the columns of $\mathbf{V}$ are the eigenvectors of the symmetric matrix $\mathbf{F}'\mathbf{F}$, and that $\mathbf{\Lambda}^{1/2}$ is a

matrix with an $R \times R$ upper left diagonal block, the diagonal being inhabited by singular values or eigenvalues $\lambda^{1/2}(j)$.

The way I like to think of this construction is that the singular values $\lambda^{1/2}(j)$ are weights that multiply simple "eigenmatrices", whose sum is the original matrix. Each of the simple eigenmatrices is the outer product of two eigenvectors, one from **U** and one from **V**. Each of these outer products of course has rank one, since each row (or column) is just a scaled version of any other row (or column). There are **R** of these eigenmatrices since the original matrix was rank **R**. In short, we can also write

$$\mathbf{F} = \sum_{j=1}^{R} \lambda^{1/2}(j) \mathbf{u}_j \mathbf{v}_j'.$$

C The Matrix Inversion Lemma

Some of the central manipulations in deriving state estimation filters are based on the Matrix Inversion Lemma (MIL). The MIL is a simple-minded set of relations between the blocks of a block-structured matrix and the blocks of its inverse.

Consider the following matrix formula:

$$\left(\begin{array}{c|cc} & n_1 & n_2 \\ \hline n_1 & A & B \\ n_2 & C & D \end{array}\right)^{-1} = \begin{array}{c|cc} & n_1 & n_2 \\ \hline n_1 & E & F \\ n_2 & G & J \end{array} \tag{43}$$

(This looks terrible but the n's are meant to be the sizes of blocks $A \ldots J$.)

Now all you have to do is notice that if this equation is true, then

$$AE + BG = I, AF + BJ = 0$$

$$CE + DG = 0, CF + DJ = I$$

From these, we get relations like this:

$$E = A^{-1} + A^{-1}BJCA^{-1} = (A - BD^{-1}C)^{-1} \tag{44}$$

$$F = -A^{-1}BJ = -EBD^{-1} \tag{45}$$

$$G = -JCA^{-1} = -D^{-1}CE \tag{46}$$

$$J = (D - CA^{-1}B)^{-1} = D^{-1} + D^{-1}CEBD^{-1} \tag{47}$$

The matrix inversion lemma follows from these relations by the substitutions

$$R \doteq -A, P \doteq D^{-1}, H \doteq B = C'$$

First, note that $C' \doteq C^T$, or C transpose. Then note the symmetry inherent in the above substitutions. Then, by substituting into eq. 47, then for E in eq. 44,

$$(P^{-1} + H'R^{-1}H)^{-1} = P - PH'(HPH' + R)^{-1}HP$$

$$(R + HPH')^{-1} = R^{-1} - R^{-1}H(P^{-1} + H'R^{-1}H)^{-1}H'R^{-1}.$$

Whew. Well, you can well ask yourself what sort of progress this is. It will turn out that we can use these re-writings to remove some inverse matrices from our calculations, or at least to replace them.

D Partial Code for $\alpha - \beta - \gamma$ Filter

```
/* this structure passes parameters to the filter */

typedef struct filter_params{
        double time_step;          /* if don't use timestamp method */
        double timeout_secs;       /* when to give up for lack of data */
        int init_ticks;            /* how long to spend initializing */
        double alpha, beta,gamma;  /* determined as in paper */
        char *name;
} *filter_params_t;


/* filter struct contains lots of boring intermediate storage.
   user mainly wants to know the values shown here */

typedef struct abc_filter{
        struct filter_params f_params;       /* all the setup information */
        filter_activity activity;            /* state of filter...initializing,
        tracking, estimating, timed out. */
        double x_est, v_est, a_est;          /* estimated variable */
        double x_future, v_future, a_future; /* predicted variable at some
time in the future;*/
        int age_ticks;                       /* total age; */
        int data_ticks;                      /*how long we've been tracking*/
        int nodata_ticks;                    /* how long since seen target */
                                    /* times in secs are also kept*/
    } *abc_filter_t;



/*Here is the version of the call that takes the current timestamp.
An alternate version of run() takes no timestamp argument and assumes
it is called on every tick. */
```

```
void run(abc_filter abc, enum signal, double X, double advance,
double time_stamp)
{
/* The abc filter takes in positional data and estimates the current and
future values of position, velocity, and acceleration.
The run() function expects a filter, an indication of
whether you have good data or not (the signal variable),
the value of the signal (one-dimensional), how far
you want the filter to estimate the signal's value into the future,
and the current time.   At any time while the filter is tracking or estimating
the variables like x_est and x_future may be accessed for current estimates
and predictions.*/

/* I am omitting a code segment to initialize the filter. It uses the input
data value to keep a running average of the velocity and acceleration.*/


if (signal == data)                          /* now come the a-b-c equations */
                {
                 abc->data_ticks++;
                 abc->data_secs+= this_step;  /* update some times --
 this_step is time since last timestamp*/

                 abc->x_pred =  abc->x_est + this_step * abc->v_est
                     + 0.5 * abc->a_est * this_step * this_step;

                 abc->v_pred =  abc->v_est + this_step* abc->a_est;

                 abc->innov = X -  abc->x_pred;

                 /* we computed predicted x and velocity. now have our innovation */

                abc->x_est = abc->x_pred + abc->f_params.alpha * abc->innov;
                abc->v_est =
                    abc->v_pred + (abc->f_params.beta / this_step) * abc->innov;
                abc->a_est =  abc->a_est + (abc->f_params.gamma /
                                  (this_step * this_step)) * abc->innov;

                 /* now we have our new estimated velocity, position, accel.*/
                }

                abc->x_future = abc->x_est + abc->v_est * adv
                    + 0.5 * abc->a_est * adv * adv;
                abc->v_future = abc->v_est + adv * abc->a_est;
                abc->a_future = abc->a_est;

                /* just roll the plant (i.e. the accel, vel, pos) forward in
   time to get the estimated state in the future at time adv ahead.*/

}               /* end of run() definition */
```

E Kalman Filter Example, Results, Code

E.1 The Problem

This example problem is again stolen from Bar-Shalom (his Example 2.4.1.); I chose it so you could go to the source if you don't believe me. This is a filter for tracking; it estimates the position and velocity of a point moving in a straight line. The state of the system is a column 2-vector of position and velocity $(x, \dot{x})$. The system evolves with constant velocity plus noise:

$$\mathbf{x}(k+1) = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \mathbf{x}(k) + \begin{bmatrix} .5 \\ 1 \end{bmatrix} v(k)$$

for $k = 0, 1, ...$ The initial state is

$$\mathbf{x}(0) = \begin{bmatrix} 0 \\ 10 \end{bmatrix},$$

so it is moving with velocity 10 starting from 0. The process (plant) noise is mean 0, white, with variance $q = E[v^2(k)]$. The measurements are only of the position (as usual), and corrupted by noise:

$$z(k) = [1\ 0]\mathbf{x}(k) + w(k),$$

where measurement noise is mean 0, white, and variance $E[w^2(k)] = 1$.

For this exercise I initialized the filter arbitrarily, somewhat near the correct values, at

$$\hat{\mathbf{x}}(0 \mid 0) = \begin{bmatrix} 0.5 \\ 11.0 \end{bmatrix}.$$

E.2 LISP Code

The following code fragments show how I translated this problem into arrays (I had to make 2×2 arrays for $\mathbf{P}$ and $\mathbf{Q}$, for example, and some scalars (like the measurement) are turned into 1×1 matrices for generality. To avoid turning over storage, I tend to preallocate arrays and reuse them. In C I use statics that are allocated on the first call to the routine. Here in Lisp everything is global. To adapt this code for another problem, all that is needed is to change the noise values and matrices: all the computations should be the same. Needless to say this would have been easier in MATLAB, but Lisp Is Good For You.

```
;;; This code relies on a simple matrix package that
;;; provides matrix-inverse, matrix-transpose, etc.
;;; Syntax pretty obvious; if function seems to have an
;;; extra argument, the last one is the result, which is also
;;; the value of the S-expression.

(defparameter *v-stdev* 0.0) ; plant noise  can go up to 1.0
(defparameter *w-stdev* 1.0) ; observ. noise
(defparameter *w-var* (* *w-stdev* *w-stdev* )) ; observ. noise  var.
```

```
(defparameter *v-var* (* *v-stdev* *v-stdev* )) ; plant noise  var.

;;;*******  Here we start defining matrices and kalman-filter
(setq F (make-array '(2 2) :initial-contents '((1.0 1.0)(0.0 1.0))))
(setq F-transpose (make-array '(2 2) ))
(matrix-transpose F F-transpose)
(setq plant-noise (make-array '(2 1) ))
(setq plant-noise-weights (make-array '(2 1) :initial-contents '((0.5)(1.0))))
(setq measurement-noise (make-array '(1 1) ))
(setq measurement (make-array '(1 1) ))
(setq measurement-prediction (make-array '(1 1) ))
(setq innovation (make-array '(1 1) ))
(setq state (make-array '(2 1))) ; position and velocity
(setq initial-state (make-array '(2 1) :initial-contents '((0.0) (10.0))))
(setq state-prediction (make-array '(2 1) ))
(setq state-estimate (make-array '(2 1) :initial-contents '((0.5) (11.0))))
(setq H (make-array '(1 2) :initial-contents '((1.0 0.0))))
(setq H-transpose (make-array '(2 1)))
(matrix-transpose H H-transpose)
(setq S (make-array '(1 1) ))
(setq S-inv (make-array '(1 1) ))
(setq R (make-array '(1 1) :initial-element *w-var*))
(setq P-updated (make-array '(2 2)))
(setq initial-P (make-array '(2 2)
                            :initial-contents (list (list *w-var* *w-var*)
                              (list *w-var* (* 2.0 *w-var*)))))
;; the above initialization is after B-S, but values here not important.

(setq Q (make-array '(2 2)
         :initial-contents (list (list *v-var* 0.0) (list 0.0 *v-var*))))
(setq P-prediction (make-array '(2 2) ))
(setq filter-gain (make-array '(2 1) ))

;; the following use a CLOS random number generator package --
;; e.g. from now on calling (sample v) produces a normal variate.
(setq v (make-instance 'normal :mean 0.0 :stdev *v-stdev*))
(setq w (make-instance 'normal :mean 0.0 :stdev *w-stdev*))

;;;***********  State and Measurement Simulation ***************
(defun state-evolve ()                          ; x = Fx + v
  (setq state (matrix-multiply-mat-mat F state))
  (setq plant-noise (matrix-mpy-num-mat
                     (sample v) plant-noise-weights ))
  (setq state (matrix-add state plant-noise)))

(defun measure ()       ; z = Hx + w
  (setq measurement (matrix-multiply-mat-mat H state))
  (setf (aref measurement-noise 0 0) (sample w))
  (setq measurement (matrix-add measurement measurement-noise)))

;;;***********  Kalman Filter Computations
```

```
;;;***********  First update P -- can be done without interacting
;;;***********  with the world (measurement) and without knowledge
;;;***********  of state estimation.
(defun state-covariance-predict ()              ; P = FPF' + Q
  (matrix-add
    (matrix-multiply-mat-mat
     (matrix-multiply-mat-mat F P-updated)  F-transpose) Q P-prediction))

(defun innovation-covariance-compute ()         ; S = HPH' + R
   (matrix-add
    (matrix-multiply-mat-mat
     (matrix-multiply-mat-mat H P-prediction) H-transpose) R S))

(defun filter-gain-compute ()                   ; W = PH'S-inv
   (matrix-multiply-mat-mat
    (matrix-multiply-mat-mat
     P-prediction (matrix-transpose H))
    (matrix-inverse S S-inv) filter-gain))

(defun state-covariance-update ()               ; P = P - WSW'
  (matrix-sub P-prediction
                  (matrix-multiply-mat-mat
                   (matrix-multiply-mat-mat filter-gain S)
                   (matrix-transpose filter-gain))
   P-updated))

(defun update-covars ()                         ;  Here's the order
  (state-covariance-predict)
  (innovation-covariance-compute)
  (filter-gain-compute)
  (state-covariance-update))

;;;***********  Now update the state estimation.  This must measure
;;;***********  the state and uses gain computed above
(defun state-predict ()                         ; xhat = F  xhat
  (matrix-multiply-mat-mat F state-estimate state-prediction))

(defun measurement-predict ()                   ; z = H xhat
  (matrix-multiply-mat-mat H state-prediction measurement-prediction))

(defun innovation-compute ()                    ; nu = z - zhat
  (matrix-sub measurement measurement-prediction innovation))

(defun estimate-state ()                        ; xhat = xhat + W nu
  (matrix-add state-prediction
              (matrix-multiply-mat-mat filter-gain innovation)
              state-estimate))

(defun compute-estimate ()                      ; Here's the order
  (state-predict)
  (measurement-predict)
```

```
  (innovation-compute)
  (estimate-state))

;;;***********  Heeeeeeeeeere's Kalman!
(defun kalman-filter ()
  (update-covars)
  (compute-estimate))

;;;***********  Simulate, Filter, and Report.
;;;***********  initialize  resets between runs, report writes results
(defun run-filt (n)                    ; n is number of iters
;;  (initialize)                       ; for convenience, not shown
  (dotimes (k n)
    (state-evolve)                     ; world ticks over
    (measure)                          ; we measure position
    (kalman-filter)                    ; we estimate position
;;    (report k)                       ; write results, not shown
  ))
```

E.3 The Results

Shown here are two sets of results, one with no plant noise ($q = 0$), corresponding to Bar-Shalom's Figs. 2-5 and 2-6, and one with ($q = 1$), corresponding to Bar-Shalom's Figs. 2-8 and 2-9. Fig. 9 shows the true velocity (10) and the KF-estimated velocity through 50 ticks. Fig. 10 shows the predicted and updated values predicted position and updated position from the **P** matrix Notice how the filter finds the true velocity and how the variance drops through time. Figs 11 and 12 are similar, but with plant noise of 1. Notice how a "constant velocity" system evolves to twice the velocity under the random walk in velocity driven by the plant noise. Also notice how the variances settle down in only a few steps.

E.4 MatLab Code

Jim Vallino coded up the Kalman filter in MatLab. Here is his version, followed by a file that works the same problem as in Section E.2.

```
function [Xhat,Zhat,Xpred,Pupdated,Ppredict,K] = Estimator(F,G,Q,H,R,P,Xest,Z)
% [Xhat,Zhat,Xpred,Pupdated,Ppredict,K] = Estimator(F,G,Q,H,R,P,Xest,Z)
%
% Run a Kalman filter where the parameters represent
%
% The state equation:
% x(k+1) = F * x(k) + G * w
% w is a gaussian random noise process with covariance Q
% The measurement equation:
% z(k) = H * x(k) + v
% v is a gaussian random noise process with covariance R
```

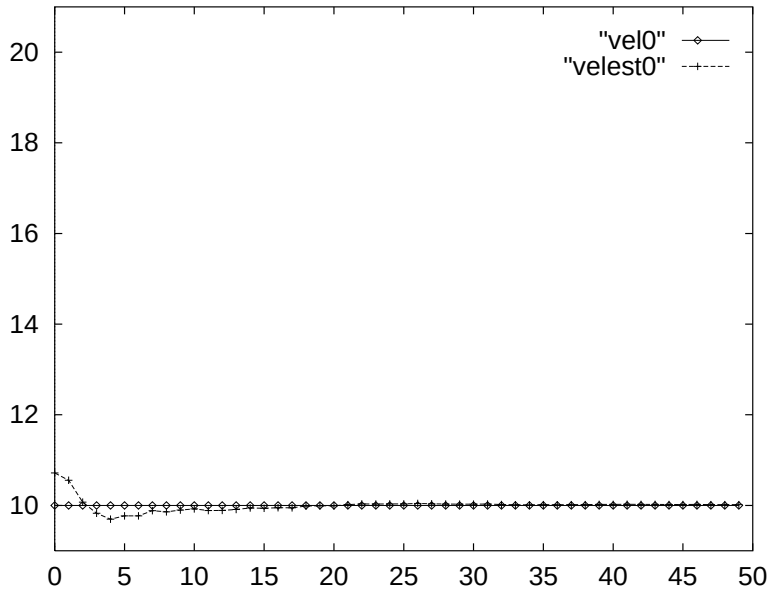

Figure 9: True and estimated velocities, no plant noise ($q = 0$).

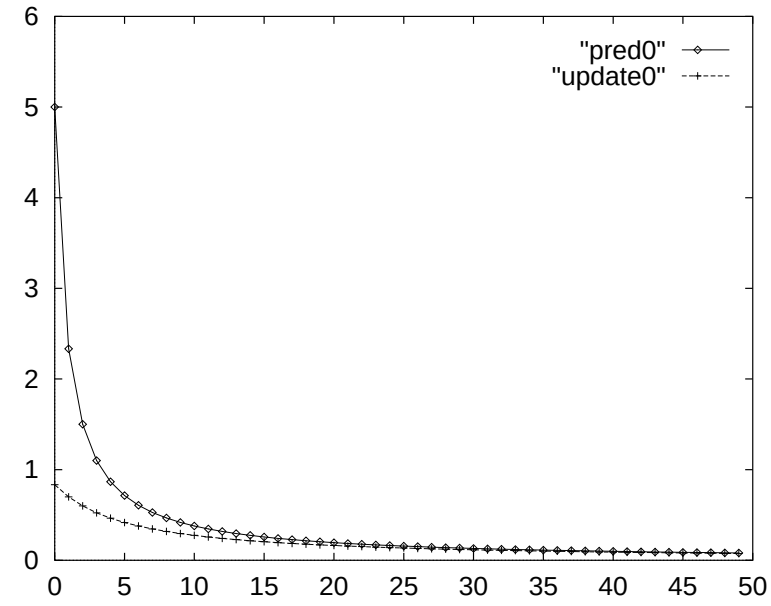

Figure 10: Position error variance, predicted and updated, $q = 0$.

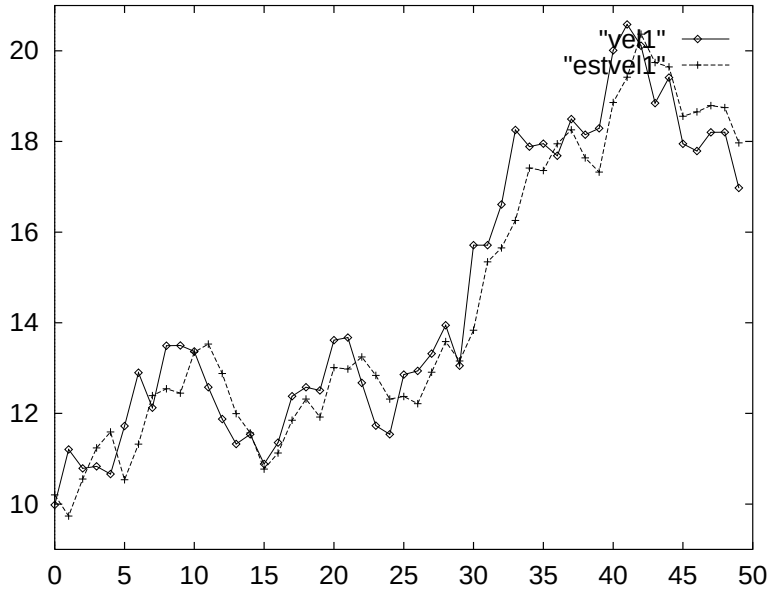

Figure 11: True and estimated velocities, $q = 1$.

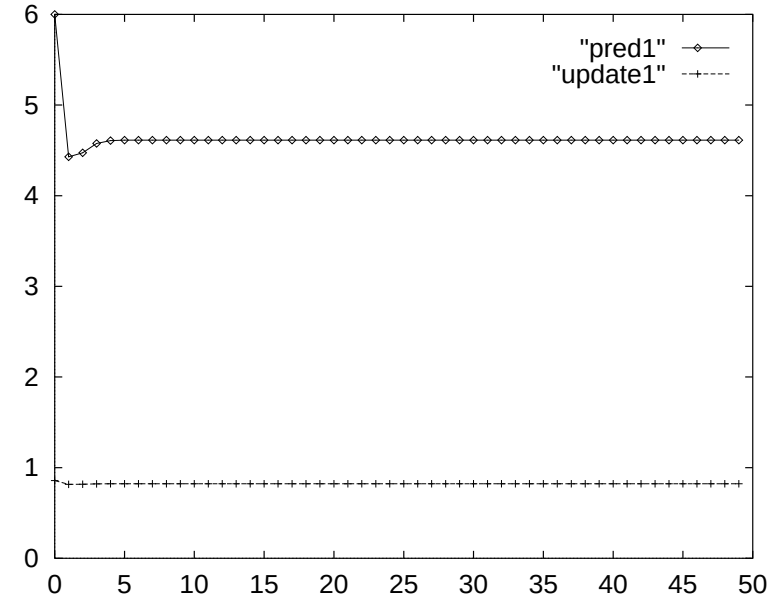

Figure 12: Position error variance, predicted and updated, $q = 1$.

```
% P is the previous Kalman estimate covariance
% Xest is the previous estimator value
% Z is the current measurement value
%
% Output from the function is:
% Xhat - new state estimate x(k+1|k+1)
% Zhat - predicted measurement z(k+1|k)
% Xpred - predicted state vector x(k+1|k)
% Pupdated - new estimate covariance P(k+1|k+1)
% Ppredict - predicted estimate covariance P(k+1|k)
% K - Kalman filter gain
%
% The system is defined by F,G,Q,H and R.  Initial values for P and
% Xest must be provided.  After that, the output values Xhat and
% Pupdated from the previous run are used for these parameters.  The
% new measurement vector Z is needed as input with each call.

% state covariance prediction
Ppredict = F * P * F' + G * Q * G';

% innovation covariance computation
S = H * Ppredict * H' + R;

% filter gain computation
K = Ppredict * H' * inv(S);

% state covariance update
[N M] = size(P);
Pupdated = Ppredict - K * S * K';

% state prediction
Xpred = F * Xest;

% measurement prediction
Zhat = H * Xpred;

% innovation computation
nu = Z - Zhat;

% state estimation
Xhat = Xpred + K * nu;


% This code calls the Kalman Estimator for the Bar-Shalom problem.
```

```
%randn('seed',550289913)
seed = randn('seed');

if (exist('mismatch') == 0)  mismatch = 1; end

vstd = 1.0; Count = 50; x = [0.0 10.0]';

Xhat = [0 0]';
P = [1 1;1 10];
F = [1 1;0 1];
G = [0.5 0;0 1];
H = [1 0];
Q = [vstd*vstd 0;0 vstd*vstd];
Qroot = sqrt(mismatch * Q);
R = [1];  Rroot = sqrt(R);

zMinus = H * x + Rroot * [randn];
z = H * x + Rroot * [randn];
Xhat = [z(1) z(1) - zMinus]';
Xtrue = zeros(size(x),Count);
Xest = zeros(size(x),Count);
Xpred = zeros(size(x),Count);
xVar = zeros(size(x),Count);
Ppred = zeros(size(x),Count * size(x));
nsee = zeros(1,Count);

Xtrue(:,1) = x;
Xest(:,1) = Xhat;
xVar(:,1) = [P(1,1) P(2,2)]';
Ppred(:,1:2) = zeros(2);

for i  = 2:Count + 1
w = Qroot * [randn randn]';
x = F * x + G * w;
v = Rroot * [randn];
z = H * x + v;
[Xhat,Zhat,Xp,P,Ppred(:,i * 2 + 1:(i + 1) * 2)] = ...
                                  Estimator(F,G,Q,H,R,P,Xhat,z);
Xtrue(:,i) = x;
Xest(:,i) = Xhat;
Xpred(:,i) = Xp;
xVar(:,i) = [P(1,1) P(2,2)]';
xtilde = x - Xhat;
nsee(i) = xtilde' * inv(P) * xtilde;
end;
```

```
Range = [1:Count + 1];

figure(1)
subplot(2,1,1);
plot(Range,Xtrue(2,:),2:Count + 1,Xest(2,2:Count + 1));
text(48,13,'solid - true')
text(48,11.5,'dash - est')
ylabel('x[2] : Velocity');
xlabel('Time step, k');

subplot(2,1,2);
plot(Range,xVar(2,:));
ylabel('x[2] : Variance');
xlabel('Time step, k');

figure(2)
if mismatch == 1
subplot(2,1,1);
plottitle = 'Matched System and Model';
else
subplot(2,1,2);
plottitle = 'Mismatched System and Model';
end
plot(2:Count+1,nsee(2:Count+1),1:Count,5.9*ones(1,Count));
title(plottitle);
ylabel('Normalized error');
xlabel('Time step, k');
```