Kalman Folding, Part 1 (review draft)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-06-29 Wed>

Contents

1 Abstract

Kalman filtering is commonplace in engineering, but less familiar to software developers. It is the central tool for estimating states of a model, one observation at a time. It runs fast in constant memory. It is the mainstay of tracking and navigation, but it is equally applicable to econometrics, recommendations, control: any application where we update models over time.

By writing a Kalman filter as a functional fold, we can test code in friendly environments and then deploy identical code with confidence in unfriendly environments. In friendly environments, data are deterministic, static, and present in memory. In unfriendly, real-world environments, data are unpredictable, dynamic, and arrive asynchronously.

The flexibility to deploy exactly the code that was tested is especially important for numerical code like filters. Detecting, diagnosing and correcting numerical issues without repeatable data sequences is impractical. Once code is hardened, it can be critical to deploy exactly the same code, to the binary level, in production, because of numerical brittleness. Functional form makes it easy to test and deploy exactly the same code because it minimizes the coupling between code and environment.

2 Applications of Kalman Filtering

Kalman filtering estimates states of a linear model from noisy observations. The most common example is the *tracking problem*. We need to know the position, attitude, and dynamics of a vehicle so we can predict its trajectory, but we can only observe radar time-of-flight and angles. We write a model that goes the other way and predicts radar observations from states, then statistically invert the model by incrementally accumulating observations. An active variant of tracking is *navigation*, where we control the vehicle by moving it from its estimated path to a desired path.

However, Kalman filtering and its many variants are much more broadly applicable, and this fact is perhaps under-appreciated in the computing mainstream. We can profitably apply such methods to any problem that fulfills the following criteria:

- we have a model that predicts observations from states
- we accumulate observations and want estimates of the states, that is, we want to invert the model
- estimation is continual; observations accumulate over time
- estimates must include uncertainty information
- uncertainty of the estimates decreases as observations accumulate
- the number of observations is unlimited
- computer memory cannot grow as observations accumulate

Some applications that come to mind include

Battery charging we have a model that predicts voltage from internal charge state. We observe voltage and estimate internal charge state to avoid over-charging and over-discharging the battery.

Tectonic drift and radio interferometry we have a model that predicts celestial positions of radio sources like quasars and satellites given geospatial positions of radio telescopes. We observe celestial positions iterferometrically and solve for geospatial positions and velocities of the telescopes. Over time, we can see tectonic drift of inter-telescope baselines.[1]

[1] JPL Geodynamics Program `http://www.jpl.nasa.gov/report/1981.pdf`

Recommendations we have a model that predicts customer's pages views and purchases (*conversions*) given the customer's interests. We observe the customer's page views and conversions and solve for the customer's interests. As observations accumulate, recommendations based on probability of future views and purchases improve.

Capacity planning we have models that predict shipping rate, queue depth and latency, and package-network bandwidths from warehouse and transportation capacity. We observe shipping rates, queue depth and latency, and package-network bandwidths and solve for warehouse and transportation capacities to pinpoint bottlenecks and prioritize capital improvements.

Supply and Demand we have models that predict prices from supply and demand. We observe prices and solve for supply and demand. This can become quite rich as the models may involve derivatives like futures and options and multiple levels of trading.

Expense Allocations we have models that predict expense reports from accounting categories like travel and office supplies. We observe actual expense reports and solve for the distribution across categories.

Geospatial we have models that predict images from terrain, surface (*e.g.*, terrain plus buildings) and entities (*e.g.*, political boundaries). We observe images and solve for terrain, surface, and entities.

Tomography we have models that predict spectral irradiance from geometry of reflective and absorptive surfaces and from states of radiation sources. We observe spectral irradiance and solve for geometry and source states.

Actuary we have models that predict life expectancy from lifestyle and age. We observe life expectancy and solve for states of the lifestyle and age model.

Incrementally, all these applications take a state estimate and an observation, and produce a new state estimate. This structure is exactly that of the the first argument of a functional fold,[2] the *accumulator function*. We argue in this paper that any Kalman filter or extension or variant with such a structure can and should be written as a functional fold.

3 Limitations and Extensions

Kalman filtering, *per se*, applies only when models are linear and noise is Gaussian. Extended Kalman Filtering (EKF) and Unscented Kalman Filtering (UKF) relieve these restrictions.[3] Sigma-point filtering and particle filtering can handle virtually any model at increased computational cost. We generally try Kalman filtering first because it is fast and small.

The calibration example below exhibits catastrophic cancellation. Information filtering and Square-Root Information Filtering (SRIF)[4] address this problem.

All examples in this paper have constant states. It is easy to add a linear model for time-evolving states as in tracking. That is the subject of a separate paper.

[2] `https://en.wikipedia.org/wiki/Fold_%28higher-order_function%29`

[3] Bar-Shalom, Yaakov, *et al.* Estimation with applications to tracking and navigation. New York: Wiley, 2001.

[4] `http://tinyurl.com/h3jh4kt`

4 Kalman Folding is Easy to Understand

Kalman Filtering is a natural extension of the *running average*, a routine computation.

4.1 Four Preludes

We write four functional folds, progressing from counting to the Kalman filter.

4.1.1 Prelude #1: Count

If I asked you to count the number of elements in a sequence **zs**, you might write code like the following:

```
zs = {55,89,144};
n = 0;
Do[n += 1,  (* run this code ...                          *)
  {z, zs}]; (* where z sequentially takes values from zs *)
Print[n];
~~> 3
```

Here, we use the Wolfram language.[5] All the code in this paper can be implemented in any modern, mainstream language. Embedded Kalman filters are typically written in C or C++. We pick Wolfram because it excels at concisely expressing mathematical code. Wolfram's `Do` is similar to Python's `for` and C#'s `foreach`[6] and literal sequences like `{55,89,144}` appear in curly braces.

If I asked you to write the same thing as a *functional fold*,[2] you might write

```
Fold[
  Function[{n, z}, n + 1], (* lambda expression of two arguments *)
  0,                       (* starting accumulation: the sum 'n' *)
  zs]                      (* input sequence                     *)
~~> 3
```

The first argument to `Fold` is the accumulator function. Here it is a *lambda expression*, written `Function` in Wolfram. It encapsulates the *business logic*;[7] does not depend on the mechanism of accessing the observations `zs`; and it has no coupling through free variables to its caller. Reducing dependencies and coupling is almost always good.

The second argument to *Fold* is the initial accumulation, zero in the present case.

The third and final argument to *Fold* is the list of observations. It is easy to implement fold over lazy streams or over asynchonous observables, and such illustrates the flexibility to move code amongst wildly different data-delivery environments. That illustration is the subject of a separate paper.

`Fold`'s job is to pass the elements of `zs` to the accumulator function one `z` at a time and to ultimately return the final accumulation.

[5] `http://reference.wolfram.com/language/`
[6] `http://rosettacode.org/wiki/Loops/Foreach`
[7] `https://en.wikipedia.org/wiki/Business_logic`

4.1.2 Prelude #2: Mean

If I asked you to compute the *mean* (assuming **zs** is not empty), you might try

```
Fold[Function[{sum, z}, sum + z], 0, zs] /
Fold[Function[{n,   z}, n   + 1], 0, zs]
~~> 96
```

a ratio of two folds. If I pressed you to do it in one *Fold* and hinted *pattern matching*, you might come up with

```
cume[{n_, sum_}, z_] := {n + 1, sum + z};
{n, sum} = Fold[cume, {0, 0}, zs];
Print[sum/n];
~~> 96
```

Pattern matching is not available for Wolfram's anonymous literal functions, so we create a *named* accumulator function, `cume` for short. Like all accumulator functions, it is binary. This one takes an accumulation `{n, sum}` and an observation `z`. The accumulation is, in turn, a pair. When called, `cume` expects its first actual argument to be a pair, and instantiates the variables `n` and `sum` to the values in that pair; `cume` must produce an updated pair of count and sum.

I press you to get rid of the outer assignment and the `Print` statement, which, unfortunately, does the critical arithmetic. That arithmetic must be done inside the accumulator function. You write:

```
cume[{x_, n_, sum_}, z_] := {(sum + z)/(n + 1), n + 1, sum + z};
Fold[cume, {0, 0, 0}, zs]
~~> {96, 3, 288}
```

where `x` is the current estimate of the mean. This is better: all arithmetic done inside `cume` at the cost of one more pattern variable in the accumulation and a few extra outputs.

1. An Insight: A Recurrence for the Mean

   We simplify the expression above by writing the new estimate x for the mean as a correction to the old. This gets rid of the running sum. We write this correction as a *gain* K times a *residual* $(z-x)$. We find the gain K by setting $x+K\times(z-x)$ equal to $(\text{sum}+z)/(n+1)$, noting that sum equals $n\,x$, and solving for K. We get $K = 1/(n+1)$. The improved code is

   ```
   cume[{x_, n_}, z_] :=
     With[{K = 1 / (n + 1)},
       {x + K * (z - x), n + 1}];
   Fold[cume, {0, 0}, zs]
   ~~> {96, 3}
   ```

   We prefer this form because

   (a) it expresses the new mean entirely in terms of the old, as a *recurrence relation*[8]

   (b) it is easy to memorize because it is an affine[9] update

   We see below that ***a Kalman filter looks almost exactly like this***.

[8] https://en.wikipedia.org/wiki/Recurrence_relation

[9] https://en.wikipedia.org/wiki/Affine_transformation

4.1.3 Prelude #3: Variance

I press on: give me the variance computed in constant memory. Variance is the sum of squared residuals divided by the count less one, Bessel's correction.[10] The variance is also the square of the standard deviation.

1. School Variance

   You'll soon break up the sum of squared residuals with this "school formula:"

   $$\sum_{i=1}^{n} (z_i - \bar{z})^2 = \sum_{i=1}^{n} z_i^2 - n\,\bar{z}^2$$

   where $\bar{z} \stackrel{\text{def}}{=} \sum_{i=1}^{n} z/n$ is the mean. You write

   ```
   cume[{var_, ssq_, x_, n_}, z_] :=
     With[{n2 = n + 1},
      With[{K = 1/n2},
       With[{x2 = x + K (z - x), ssq2 = ssq + z^2},
        {(ssq2 - n2 x2^2)/Max[1, n], ssq2, x2, n2}]]];
   Fold[cume, {0, 0, 0, 0}, zs]
   ~~> {2017, 31682, 96, 3}
   ```

   where `Max[1,n]` both accounts for Bessel's correction and prevents divide-by-zero. The accumulation argument to `cume` now pattern-matches the variance and three auxiliary values:

   `var` the variance so far

   `x` mean so far

   `n` count so far

   `ssq` sum of squares so far

   This form keeps the recurrence for the mean that we found preferable above. Can we find a recurrence for the variance?

2. Recurrent Variance

   A recurrence should express the new variance as the old variance plus a correction depending only on old values. Start by seeking a recurrence for the sum of squared residuals

   $$\Sigma \stackrel{\text{def}}{=} \sum_{i=1}^{n} (z_i - \bar{z})^2$$

   This is not hard to find:

   $$\Sigma \leftarrow \Sigma + Kn(z - \bar{z})^2$$

   remembering that $K = 1/(n+1)$. Code this as an accumulator function:

[10] https://en.wikipedia.org/wiki/Bessel's_correction

```
cume[{var_, x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   With[{x2 = x + K (z - x),
     ssr2 = (n - 1) var + K n (z - x)^2},
    {ssr2/Max[1, n], x2, n + 1}]];
Fold[cume, {0, 0, 0}, zs]
~~> {2017, 96, 3}
```

As before, a recurrence lets us get rid of an auxiliary variable, this time, the sum of squared residuals. Getting rid of variables is almost always better.

3. Welford's Variance, Catastrophic Cancellation, SRIF

   We have discovered something equivalent to Welford's formula.[11, 12]

   $$\Sigma \leftarrow \Sigma + (z - \bar{z}_n)(z - \bar{z}_{n+1})$$

   Our recurrence and Welford's reduce the chances of catastrophic cancellation,[13] the chief numerical torment of variances. Welford's subtracts before squaring, whereas the School variance squares before subtracting. It is all too easy to get negative or meaningless variances by subtracting large floating-point numbers resulting from squares.

   The analogue to Welford's in the world of Kalman filtering is Square-Root Information Filtering (SRIF).[4] SRIF is beyond the scope of this paper, but we point out places below where it might be employed.

4. All Intermediate Results

   If we want to see all intermediate results, which include the *running* variance, mean, and count, you make one tiny change, calling *FoldList* instead of *Fold*:

   ```
   FoldList[cume, {0, 0, 0}, xs]
   ~~>
   ```

   $$\begin{bmatrix} 0 & 0 & 0 \\ 0 & 55 & 1 \\ 578 & 72 & 2 \\ 2017 & 96 & 3 \end{bmatrix}$$

   still in constant memory. *FoldList*, often called *scan* in other programming languages, produces a *Sequence* of *Accumulation*, thus has a slightly different type to *Fold*.

5. Variance of the Mean

   As observations accumulate, the estimate of the mean should improve. We know how to incrementally compute variance of the observations with respect to the running mean. Can we compute variance of the running mean with respect to the unknown, abstract, true, constant mean $\aleph$?

[11] http://tinyurl.com/nfz9fyo
[12] http://rebcabin.github.io/blog/2013/02/04/welfords-better-formula/
[13] https://en.wikipedia.org/wiki/Catastrophic_cancellation

In practical applications, outside of testing, we do not know this ground truth $\aleph$. Remarkably, however, we can calculate the variance of the running mean with respect to $\aleph$ assuming only statistics of the observation noise. Then, in testing, we can check the percentage of residuals that lie within one standard deviation of this truth. If our computations behave well, about 68% of residuals should be within one theoretical sigma of ground truth. This is a good test for Kalman filters, extensions, and variants.

Write this variance as the expectation value over the probability distribution of the observation noise. Consider the $(n+1)$-st estimate of the mean, $\bar{z}_{n+1}$, where z is the current observation:

$$\bar{z}_{n+1} = \bar{z}_n + K(z - \bar{z}_n)$$

Its residual with respect to the unknown truth $\aleph$ is

$$(\aleph - \bar{z}_{n+1}) = (\aleph - \bar{z}_n) - K(z - \bar{z}_n)$$

The observation z is the truth $\aleph$ plus a sample ζ_n of the random noise distribution:

$$\begin{aligned}(\aleph - \bar{z}_{n+1}) &= (\aleph - \bar{z}_n) - K(\aleph + \zeta_n - \bar{z}_n) \\ &= K\,n\,(\aleph - \bar{z}_n) - K\,\zeta_n\end{aligned}$$

because $K = 1/(n+1)$. Squaring

$$(\aleph - \bar{z}_{n+1})^2 = (K\,n\,(\aleph - \bar{z}_n))^2 - 2K^2\,n\,(\aleph - \bar{z}_n)\zeta_n + (K\,\zeta_n)^2$$

Take the expectation value of this square over the distribution of ζ. This distribution must always have zero mean, so the middle term vanishes. It has constant variance, which we write as capital zeta, Z. Also write P_n, a traditional notation, for $E\left[(\aleph - \bar{z})^2\right]$. We get

$$P_{n+1} = K^2(n^2 P_n + Z)$$

This recurrence has a closed-form solution for $n > 0$, that being $P_n = Z/n$. It follows that

$$P_{n+1} = P_n - K^2 D$$

where $D \stackrel{\text{def}}{=} Z + P_n$. For $n = 0$, $P_1 = K^2 Z$.

This is the preferred mathematical form for the variance of the mean because the correction is negative, reminding us that this variance decreases as observations accumulate. We see a similar form below where we develop the Kalman filter. However, because it is a difference of squares, it is exposed to catastrophic cancellation, just like the School variance.

We also find $K = P/D$, another preferred form that shows up in the Kalman filter. In fact, we justify the notation D through this form, because D appears as a denominator.

4.1.4 Prelude #4: Linear Least Squares

Finally, I press you to find the coefficients of a cubic polynomial from noisy observations using linear least-squares and a fold (either *Fold* or *FoldList*), still in constant memory. If you succeed, you'll invent basic Kalman folding.

In more depth, I want you to estimate a column vector of four unknowns:

$$\mathbf{x} = \begin{bmatrix} x_0 & x_1 & x_2 & x_3 \end{bmatrix}^\intercal$$

where the polynomial is the inner product of a row of powers of t

$$\mathbf{A} = \begin{bmatrix} 1 & t & t^2 & t^3 \end{bmatrix} = \begin{bmatrix} t^0 & t^1 & t^2 & t^3 \end{bmatrix}$$

and $\mathbf{x}$, and the observations are (values of) the polynomial plus noise ζ:

$$z = \mathbf{A}\,\mathbf{x} = x_0 t^0 + x_1 t^1 + x_2 t^2 + x_3 t^3 + \zeta$$

The system is *linear* in the *model states* $\mathbf{x}$ through the *observation partials*, $\mathbf{A}$. The partials are obviously nonlinear in t, by design.

An oracle gives you some specific values of t and noisy z, in arbitrary order:

$$\begin{bmatrix} t_0 & z_0 \\ t_1 & z_1 \\ t_2 & z_2 \\ t_3 & z_3 \\ t_4 & z_4 \end{bmatrix} = \begin{bmatrix} 0. & -2.28442 \\ 1. & -4.83168 \\ -1. & -10.4601 \\ -2. & 1.40488 \\ 2. & -40.8079 \end{bmatrix}$$

which plot as in figure 1, where the solid line represents the true model only the oracle knows, and the dots represent the noisy observations the oracle gives you. Your job is to estimate the states $\mathbf{x}$ so that your polynomial gets as close to the oracle's as the data allow. You must also produce the running covariance[14] matrix $\mathbf{P}$ along with the states so that we have a measure of their uncertainty.

You think for a *very* long time, remembering everything you learned above, and eventually realize that the following `cume` will do it:

```
cume[Zeta_][{x_, P_}, {A_, z_}] :=
 Module[{D, K},
  D = Zeta + A.P.Transpose[A];
  K = P.Transpose[A].Inverse[D];
  {x + K.(z - A.x), P - K.D.Transpose[K]}]
```

where all quantities are matrices and the lower dot is Wolfram's matrix multiplication. Specifically:

- $\mathbf{Z}$ is a 1×1 matrix, the covariance of observation noise
- $\mathbf{x}$ is a column 4-vector, the model states
- $\mathbf{P}$ is a 4×4 matrix, the theoretical covariance of $\mathbf{x}$
- $\mathbf{A}$ is a row 4-vector, the *observation partials*

[14] We use the terms *covariance* for matrices and *variance* for scalars.

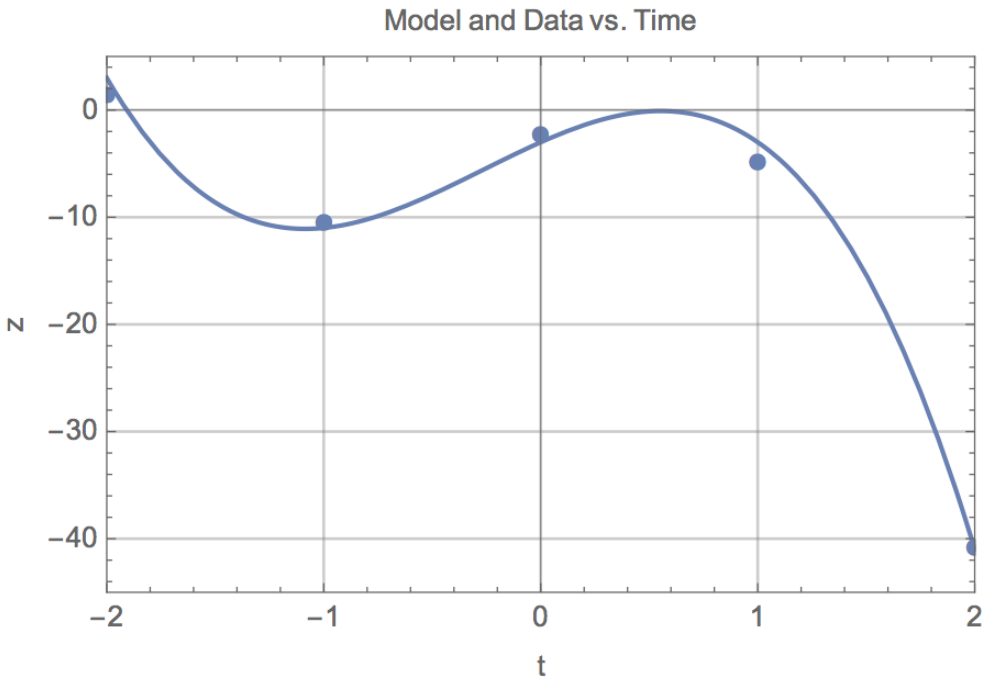


Figure 1: Model (solid line) and fake data (dots)

- $\mathbf{z}$ is a 1×1 matrix, effectively a scalar
- $\mathbf{D}$ is a 1×1 matrix, generalization of the *denominator* D we defined above in Prelude #3
- $\mathbf{K}$ is a column 4-vector, the *Kalman gain*, generalization of the gain K from Prelude #3

1. Dimensional Arguments

   Recall Prelude #3, where D is $Z + P$. We see it here as $\mathbf{Z} + \mathbf{A}\,\mathbf{P}\,\mathbf{A}^{\intercal}$. We didn't see $\mathbf{A}$ previously because our observations z were equal to the states. However, here we need two factors of $\mathbf{A}$ to make the dimensions work out. We mean "dimensions" in two senses:

   (a) dimensions of the matrices, as in numbers of rows and columns

   (b) physical dimensions of units of measure when $\mathbf{z}$ and $\mathbf{x}$ denote physical quantities, as in most applications.

   We see another factor of $\mathbf{A}$ in $\mathbf{K}$, again required by dimensional analysis.

   The requirement of dimensional consistency almost forces this Kalman accumulator function. Here is the full dimensional breakdown. If the physical and matrix dimensions of $\mathbf{x}$ are $[[\mathbf{x}]] \stackrel{\text{def}}{=} (\mathcal{X}, n \times 1)$ and of $\mathbf{z}$ are $[[\mathbf{z}]] \stackrel{\text{def}}{=} (\mathcal{Z}, b \times 1)$, then

$$
\begin{array}{lcccc}
[[\mathbf{Z}]] & = & (& \mathcal{Z}^2 & \mathfrak{b} \times \mathfrak{b} \quad) \\
[[\mathbf{A}]] & = & (& \mathcal{Z}/\mathcal{X} & \mathfrak{b} \times \mathfrak{n} \quad) \\
[[\mathbf{P}]] & = & (& \mathcal{X}^2 & \mathfrak{n} \times \mathfrak{n} \quad) \\
[[\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{Z}^2 & \mathfrak{b} \times \mathfrak{b} \quad) \\
[[\mathbf{D}]] & = & (& \mathcal{Z}^2 & \mathfrak{b} \times \mathfrak{b} \quad) \\
[[\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{X}\mathcal{Z} & \mathfrak{n} \times \mathfrak{b} \quad) \\
[[\mathbf{K}]] & = & (& \mathcal{X}/\mathcal{Z} & \mathfrak{n} \times \mathfrak{b} \quad)
\end{array}
$$

In all examples in this paper, the observations $\mathbf{z}$ are 1×1 matrices, equivalent to scalars, so $\mathfrak{b} = 1$, but the theory and code carry over to multi-dimensional vector observations.

We can see *by inspection* that the dimensions of the estimate[15] $\mathbf{x} + \mathbf{K} \cdot (\mathbf{z} - \mathbf{A}\,\mathbf{x})$ and the covariance $\mathbf{P} - \mathbf{K}\,\mathbf{D}\,\mathbf{K}^\intercal$ are minimal and correct. This "correct-by-inspection" formulation is invaluable for checking mathematics and code.

2. Lambda Lifting $\mathbf{Z}$

   This `cume` *lambda lifts*[16] the observation covariance $\mathbf{Z}$ because it's constant for the lifetime of the filter. This means that you must call this `cume` once with a $\mathbf{Z}$ to get the accumulator function. We've introduced one extra level of function-ness to avoid either

   - coupling `cume` to an external value of $\mathbf{Z}$ that is, *closing*[17] over $\mathbf{Z}$ (we insist on pure functional coupling only)
   - passing around a constant $\mathbf{Z}$ in every observation packet `{A, z}`. Below, where we allow $\mathbf{Z}$ to change with each observation, we will put it in the observation packet.

3. Folding the Accumulator Function

   The effect of a *Fold* or *FoldList* with this `cume` is to incrementally replace the prior estimate, $\mathbf{x}$, with $\mathbf{x} + \mathbf{K} \cdot (\mathbf{z} - \mathbf{A}\,\mathbf{x})$ and to replace the prior covariance, $\mathbf{P}$, with $\mathbf{P} - \mathbf{K}\,\mathbf{D}\,\mathbf{K}^\intercal$.

   Here is this `cume` in action. Assume an observation-noise covariance of 1. Our initial estimate for $\mathbf{x}$ is the column vector $\begin{bmatrix}0 & 0 & 0 & 0\end{bmatrix}^\intercal$, meaning we know nothing, and our initial state covariance is a 'practical' infinity of $1,000$ on the diagonal, emphasizing that we know nothing.

$$
\mathbf{P}_0 = \begin{bmatrix}
1000. & 0. & 0. & 0. \\
0. & 1000. & 0. & 0. \\
0. & 0. & 1000. & 0. \\
0. & 0. & 0. & 1000.
\end{bmatrix} = 1000. \times \mathbf{1}_{4\times 4}
$$

   Use *Fold* rather than *FoldList* because we want just the final result. `Chop` removes quantities within a floating-point quantum of zero.

[15] We sometimes use the center dot or the $\times$ symbols to clarify matrix multiplication. They have no other significance and we can always write matrix multiplication just by juxtaposing the matrices.

[16] `https://en.wikipedia.org/wiki/Lambda_lifting`

[17] `https://en.wikipedia.org/wiki/Closure_(computer_programming)`

```
Fold[cume[IdentityMatrix[1]],
  {ColumnVector[{0, 0, 0, 0}], IdentityMatrix[4]*1000.0},
  {{{{1,  0., 0.,  0.}}, { -2.28442}},
   {{{1,  1., 1.,  1.}}, { -4.83168}},
   {{{1, -1., 1., -1.}}, {-10.46010}},
   {{{1, -2., 4., -8.}}, {  1.40488}},
   {{{1,  2., 4.,  8.}}, {-40.8079}}}
  ] // Chop
~~>
```

$$\mathbf{x} = \begin{bmatrix} -2.97423 \\ 7.2624 \\ -4.21051 \\ -4.45378 \end{bmatrix} \tag{1}$$

$$\mathbf{P} = \begin{bmatrix} 0.485458 & 0 & -0.142778 & 0 \\ 0 & 0.901908 & 0 & -0.235882 \\ -0.142778 & 0 & 0.0714031 & 0 \\ 0 & -0.235882 & 0 & 0.0693839 \end{bmatrix}$$

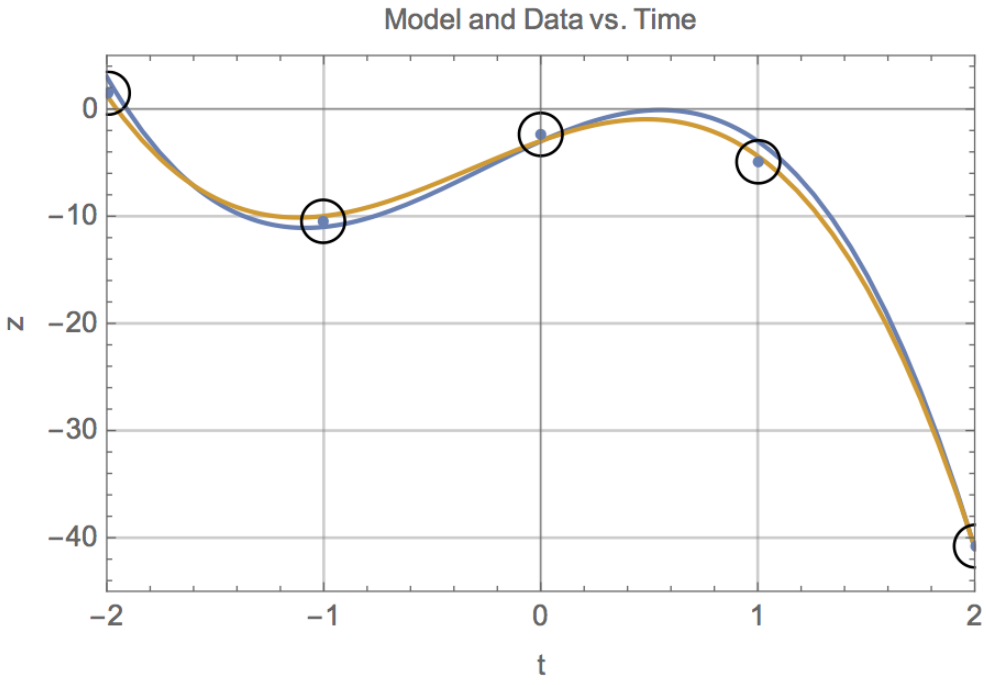


Figure 2: Model (dark line), fake data (dots), and estimated model (light line)

Results are not bad: our oracle reveals that ground-truth is $\begin{bmatrix}-3 & 9 & -4 & -5\end{bmatrix}^\intercal$; compare this to equation 1. Figure 2 plots the solution.

To find out quantitatively how bad or how good, inspect $\mathbf{P}$. Its diagonal elements are the squares of the standard deviations of the estimated states. Those standard deviations are:

$$\begin{bmatrix}0.6967 & 0.9497 & 0.2672 & 0.2634\end{bmatrix}^\intercal$$

All our estimates are within two standard deviations of ground truth, so we would expect worse performance in only 5 percent of cases.

4.2 Kalman is a Swiss-Army Knife

This is remarkable. An accumulator function, small enough to fit on a business card, suffices to estimate states of *any* linear model in constant memory. It does not need data to get started, just *a-priori* guesses, even vacuous ones. Its code fits on tiny embedded systems. It empowers radar, navigation, tracking, the entire space age, industrial robotics, battery-charge estimation, radio interferometry, econometrics. It can be extended to non-static and nonlinear models, to non-scalar observations, and to non-Gaussian noise sources. It's the sweet spot in a spectrum of statistical estimators that progress from counting to Markov-chain Monte Carlo (MCMC) filters.[18]

4.3 More Intuition

In *ex-post-facto* justification for choosing the Wolfram language, it allows the code above to look very similar to the mathematical specification:

$$\text{cume}\left(\mathbf{Z}\right)\left(\{\boldsymbol{x},\mathbf{P}\},\{\mathbf{A},\boldsymbol{z}\}\right)=\{\boldsymbol{x}+\mathbf{K}\,\left(\boldsymbol{z}-\mathbf{A}\,\boldsymbol{x}\right),\mathbf{P}-\mathbf{K}\,\mathbf{D}\,\mathbf{K}^\intercal\} \tag{2}$$

where

$$\mathbf{K}=\mathbf{P}\,\mathbf{A}^\intercal\,\mathbf{D}^{-1} \tag{3}$$

$$\mathbf{D}=\mathbf{Z}+\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal \tag{4}$$

A good way to intuit this solution is to think of $\mathbf{K}$ as the derivative $\mathrm{d}\boldsymbol{x}/\mathrm{d}\boldsymbol{z}$. We've already seen that the physical dimensions of $\mathbf{K}$ are the required $\mathcal{X}/\mathcal{Z}$. Then, the update resembles

$$\boldsymbol{x}\leftarrow\boldsymbol{x}+\frac{\mathrm{d}\boldsymbol{x}}{\mathrm{d}\boldsymbol{z}}\left(\boldsymbol{z}-\mathbf{A}\,\boldsymbol{x}\right)$$

which looks like a differential update to the estimate $\boldsymbol{x}$, where the differential of $\boldsymbol{z}$ is the residual difference between an actual observation $\boldsymbol{z}$ and a prediction of $\boldsymbol{z}$ from the current model partials matrix $\mathbf{A}$ times the prior estimate $\boldsymbol{x}$.

The gain $\mathrm{d}\boldsymbol{x}/\mathrm{d}\boldsymbol{z}$ has the form, in a slight abuse of notation:

$$\mathbf{K}=\frac{\mathbf{P}\,\mathbf{A}^\intercal}{\mathbf{Z}+\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal}$$

[18] https://en.wikipedia.org/wiki/Particle_filter

An elegant but non-trivial derivation of this appears in a separate paper in this series.[19] Just remember that the *denominator matrix*, $\mathbf{D}$, is $\mathbf{Z}$ *plus* a correction. Likewise, I do not know an easy demonstration that the new $\mathbf{P}$ is the old $\mathbf{P}$ minus $\mathbf{K}\,\mathbf{D}\,\mathbf{K}^\intercal$. In this case, just remember the *minus* sign. Intuitively, it's plausible that the new $\mathbf{P}$ should be 'less than' the old $\mathbf{P}$ because the new observations add information and decrease uncertainty. That plausibility motivates the minus sign in the formula.

5 Functional Form Encapsulates Code

We stated above that we can run *exactly* the same Kalman code over lazy streams and over asynchronous observables. This is possible because the only coupling between the environment and the Kalman code is function invocation.

Why is this important? Because developers can test in friendly environments — where observations are deterministic, pseudo-random, static, present in memory all at once, and where code can be stopped in a debugger. Once the code is hardened, developers can then deploy it with confidence in unfriendly, real-world environments — where data are unpredictable, asynchronous, real-time, infinite in length, and logging is the only forensic tool.

This advantage can be critically important when detecting, diagnosing, and correcting numerical issues like catastrophic cancellation, as we see in the calibration example below. Numerical issues can substantially complicate code, and being able to move exactly the same code, without even recompiling, between testing and deployment can make the difference to a successful application. We have seen many cases where differences in compiler flags, let alone differences in architectures, even between different versions of the same CPU family, introduce enough differences in the generated code to cause qualitative differences in the output. A filter that behaved well in the lab can fail in practice. In embedded applications like tracking and navigation, filter failure can cause vehicles to crash. It *must* be caught in testing and it must *not* be allowed to happen in deployment.

6 An Instrument-Calibration Example

Imagine an accelerometer that comes from the factory with unknown bias, scale, and drift errors. The user's manual instructs us to calibrate the instrument before fielding. That means estimating these errors so that we can subtract their effects in post-processing.

In this example, we allow the observation covariance $\mathbf{Z}$ to vary with independent variable, so we don't lambda-lift $\mathbf{Z}$, but pass it into the filter each step along with $\mathbf{A}$ and $\boldsymbol{z}$.

6.1 Functional Form, Again

Because the values of these errors are small, their variances, by squaring, are even smaller, and we invite catastrophic cancellation into the filter, on purpose. We verified numerically that in all examples above, three, mathematically equivalent expressions for the covariance, namely

[19] Beckman, *Kalman Folding 3: Derivations*, to appear.

$$\mathbf{P} \leftarrow \mathbf{L}\,\mathbf{P}$$
$$\mathbf{P} \leftarrow \mathbf{L}\,\mathbf{P}\,\mathbf{L}^\mathsf{T} + \mathbf{K}\,\mathbf{Z}\,\mathbf{K}^\mathsf{T}$$
$$\mathbf{P} \leftarrow \mathbf{P} - \mathbf{K}\,\mathbf{D}\,\mathbf{K}^\mathsf{T}$$

where $\mathbf{L} = \mathbf{1} - \mathbf{K}\,\mathbf{A}$, produce results identical to six significant figures.[20] This means high confidence that numerical issues are not important in these examples.

However, in the present example, all three expressions produce different results: two converge with different numerical details, and one fails. The first two occasionally produce negative covariances, and the third produces mostly negative covariances, exhibiting catastrophic cancellation.

Once a covariance produces bad values, the estimates soon follow because the covariance feeds back through $\mathbf{D} = \mathbf{Z} + \mathbf{A}\,\mathbf{P}\,\mathbf{A}^\mathsf{T}$ and then into $\mathbf{K} = \mathbf{P}\,\mathbf{A}^\mathsf{T}/\mathbf{D}$. Once $\mathbf{K}$ destabilizes, the estimates $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{K}(z - \mathbf{A}\,\mathbf{x})$ quickly become nonsense.

This example was adapted from Zarchan and Musoff.[21] We verified numerically that one of our examples matches their results, including negative variances, to six significant figures. Presumably because their code eventually converged, they didn't discuss catastrophic cancellation, but we focus on it here.

6.2 Details of the Example

The documented calibration procedure is to hold the accelerometer vertically at a sequence of angles from 0 degrees to 180 degrees in 2-degree increments, allowing Earth's gravitation to provide a fiducial acceleration. Measure the output of the accelerometer and compare it to a theoretical model that depends on the bias, scale, and drift errors. The model is linear, so the procedure is a perfect prescription for a Kalman filter.

Let the angle of the accelerometer with respect to the vertical be θ. The accelerometer documentation dictates that the output should read

$$a = g\,\cos(\theta) + z \tag{5}$$
$$z = b + s\,g\,\cos(\theta) + d\,g^2\,\cos^2(\theta) \tag{6}$$

where we observe z and

$g =$	acceleration of Earth's gravitation	$\sim 32.2\,\text{foot/sec}^2$
$b =$	bias	foot/sec^2
$s =$	scale	dimensionless
$d =$	drift	sec^2/foot

Our vector of states is $\mathbf{x} = \begin{bmatrix} b & s & d \end{bmatrix}^\mathsf{T}$ and our partials are $\mathbf{A}(\theta) = \begin{bmatrix} 1 & g c_\theta & g^2 c_\theta^2 \end{bmatrix}$ where $c_\theta = \cos(\theta)$.

Our measurement rig delivers noisy angles, not noisy values of $\cos(\theta)$, so we must translate angle noise ζ_θ, of constant variance σ_θ^2, into c_θ noise ζ_{c_θ} with a variable variance σ_{c_θ}, and then into observation noise ζ_z with variance $\mathbf{Z}(\theta)$. We find

[20] Derivations of these forms appear in part 3 of this series.

[21] Zarchan and Musoff, *Fundamentals of Kalman Filtering, A Practical Approach, Fourth Edition*, Ch. 4

$$\mathbf{Z}(\theta) = (\sigma_\theta \, g \, \sin(\theta))^2 \tag{7}$$

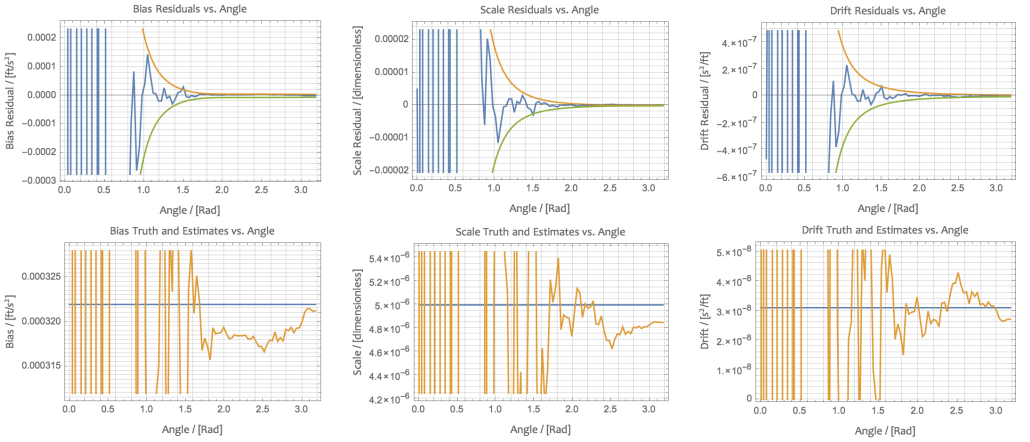
Posit a ground truth for $\mathbf{x}$ of

$$\aleph = \begin{bmatrix} 10 \times 10^{-6} \, g \\ 5 \times 10^{-6} \\ 1 \times 10^{-6} / g \end{bmatrix} \tag{8}$$

and generate deterministic pseudo-randomly noisy observations as follows:

$$\mathbf{A}(\theta) \cdot \aleph + g \, (\cos(\theta + \zeta_\theta) - \cos(\theta)) \tag{9}$$

6.3 Results



% Bias resids < σ (E >= 68)	86.9565
Bias resids >= σ	7.
Bias resids < σ	80.
average Bias resid	0.0113483
theoretical std dev	5.08884×10^{-6}
std dev of Bias resids	0.114917

% Scale resids < σ (E >= 68)	61.9565
Scale resids >= σ	30.
Scale resids < σ	57.
average Scale resid	−0.000714109
theoretical std dev	9.08003×10^{-8}
std dev of Scale resids	0.00717694

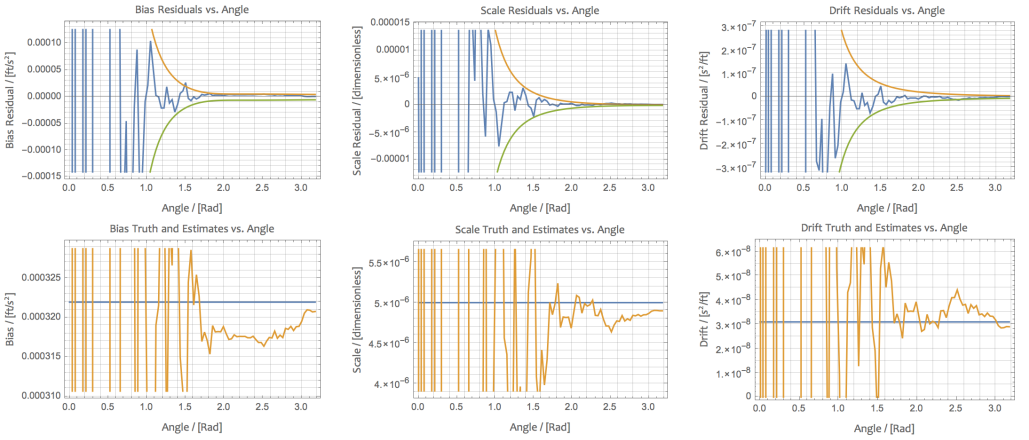
% Drift resids < σ (E >= 68)	85.8696
Drift resids >= σ	8.
Drift resids < σ	79.
average Drift resid	0.0000112456
theoretical std dev	6.39081×10^{-9}
std dev of Drift resids	0.000112081

Figure 3: Accelerometer calibration results, $\mathbf{P} \leftarrow \mathbf{L} \, \mathbf{P}$

We test three mathematically identical Kalman accumulator functions, differing only in their formula for $\mathbf{P}$. The first, illustrated in figure 3, reproduces Zarchan and Musoff to six significant figures, including occasional negative variances. This produces acceptable results, near ground truth. Zarchan and Musoff's code for the filter is fully enmeshed with their code for constructing and delivering observations, so there is no hope of extracting it without change into a deployment environment. Not so our code, which we can harden and move verbatim, according to the main message of this paper.

```
kalman[{x_, P_}, {Zeta_, A_, z_}] :=
  Module[{D, K, L},
   D = Zeta + A.P.Transpose[A];
```

```
K = P.Transpose[A].inv[D];
L = (id[len[P]] - K.A);
{x + K.(z - A.x), L.P}];
                    (* also L.P.Transpose[L] + K.Zeta.Transpoe[K] *)
                    (* or   P - K.D.Transpose[K] *)
```



% Bias resids < σ (E >= 68)	83.6957
Bias resids >= σ	4.
Bias resids < σ	77.
average Bias resid	0.00358082
theoretical std dev	5.02418×10^{-6}
std dev of Bias resids	0.0923306

% Scale resids < σ (E >= 68)	60.8696
Scale resids >= σ	25.
Scale resids < σ	56.
average Scale resid	−0.000193406
theoretical std dev	6.66051×10^{-8}
std dev of Scale resids	0.00597229

% Drift resids < σ (E >= 68)	83.6957
Drift resids >= σ	4.
Drift resids < σ	77.
average Drift resid	2.51554×10^{-6}
theoretical std dev	5.79413×10^{-9}
std dev of Drift resids	0.0000968455

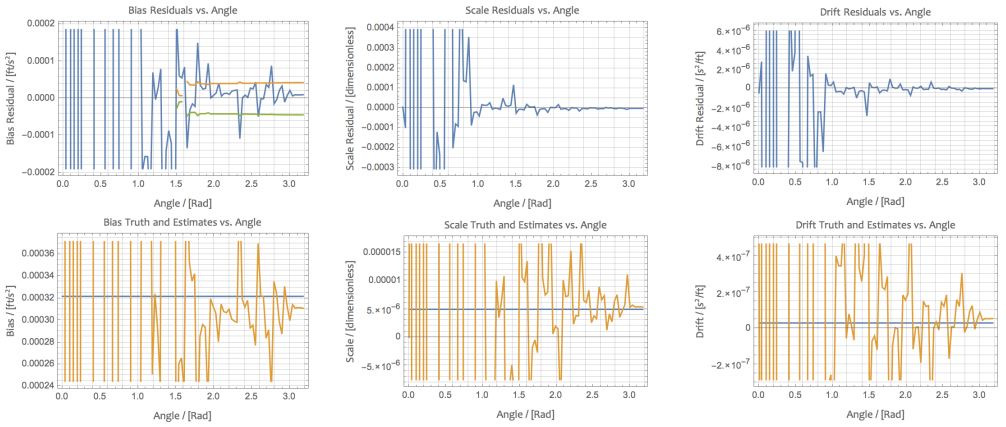
Figure 4: Accelerometer calibration results, $\mathbf{P} \leftarrow \mathbf{L}\,\mathbf{P}\,\mathbf{L}^\intercal + \mathbf{K}\,\mathbf{Z}\,\mathbf{K}^\intercal$

The second filter produces similar and acceptable results but with totally different numerical values even with exactly the same pseudo-random fakes. This is illustrated in figure 4. Tracking down the sources of these numerical differences would require a deep numerical study, but might be necessary in a safety-critical or mission-critical application.

We would be advised to move to the SRIF family of filters after seeing this much variability, but even more so after the third filter, our original Kalman fold, fails, as illustrated in figure 5. Remarkably, despite meaningless covariances, the estimates are still reasonable compared to ground truth, but they certainly cannot be trusted.

6.4 Robustness

It is a general feature of Kalman filters that they are difficult to break in practice. We can bend the rules, feeding them non-Gaussian noise with very large variances, and they will often converge despite the abuse. In fact, routine practice is to patch diverging filters with artificial process noise. However, this example illustrates that hazards exist and should be carefully analyzed. This is all the more reason to favor a functional form that enables easy movement between testing and deployment.



% Bias resids < σ (E >= 68)	48.913
Bias resids >= σ	13.
Bias resids < σ	45.
average Bias resid	−0.00498231
theoretical std dev	0.0000434082
std dev of Bias resids	0.0729717

% Scale resids < σ (E >= 68)	7.6087
Scale resids >= σ	3.
Scale resids < σ	7.
average Scale resid	0.000320187
theoretical std dev	$0. + 2.6977 \times 10^{-6}\,i$
std dev of Scale resids	0.00458956

% Drift resids < σ (E >= 68)	7.6087
Drift resids >= σ	4.
Drift resids < σ	7.
average Drift resid	-5.16615×10^{-6}
theoretical std dev	$0. + 7.28827 \times 10^{-8}\,i$
std dev of Drift resids	0.0000721718

Figure 5: Accelerometer calibration failure, $\mathbf{P} \leftarrow \mathbf{P} - \mathbf{K}\,\mathbf{D}\,\mathbf{K}^{\intercal}$

7 Concluding Remarks

The basic Kalman filter statistically inverts the basic model

$$\boldsymbol{z} = \mathbf{A}\,\boldsymbol{x}$$

yielding a sequence of estimates of $\mathbf{x}$, which may additionally evolve over time according to a linear state-transition model.

The flashes of insight required to invent the Kalman filter were really flashes of genius. There was a lot of prior work, mostly coming from radar and the tracking community, on incremental optimal estimation. But Kalman saw what others did not see: that new observations could be simply scaled by covariances and added to old. These insights enabled whole disciplines of guidance and control that propelled the space age and all the rest of aerospace since Kalman had them in 1960 because *we can perform state estimation fast, incrementally and in constant memory just as with other, lesser statistics like count, mean, and standard deviation.*

Writing the code for a Kalman filter as a foldable accumulator function gives us

- economy of expression, reducing otherwise complex and tricky code to a few, modularized and decoupled lines
- flexibility of testing and deployment, allowing exactly the same code, even and especially binary, to run over arrays in memory, lazy streams, asynchronous observables, any data source that can support a *fold* operator
- flexibility of application, reducing the inputs to a handful of matrices

We hope that by illuminating this flexibility we can expand the use of the Kalman filter into other application areas. We also hope to inspire the study and application of more advanced filters

like the Extended and Unscented Kalman filters, Information filters and Square-Root Information filters, Sigma-Point filters, and Particle filters.