Kalman Folding 1.5: Running Statistics

Extracting Models from Data, One Observation at a Time

Brian Beckman

31 August 2016

Abstract

This paper fills in some blanks left between part 1 of this series, Kalman Folding (http://vixra.org/abs/1606.0328), and the rest of the papers in the series. In part 1, we present basic Kalman filtering as a functional fold, highlighting the advantages of this form for hardening code in a test environment. In that paper, we motivated the Kalman filter as a natural extension of the running average and variance, writing both as functional folds computed in constant memory. We expressed the running statistics as recurrence relations, where the new statistic is the old statistic plus a correction. We write the correction as a gain factor times some transform of a residual. The residual is the difference between the current (old) statistic and the incoming (new) observation. In both expressions, for brevity, we left derivations to the reader. Here, we present those derivations in full "school-level" detail, along with some basic explanation of the programming language that mechanizes the computations.

Preliminaries

This is a PDF print of interactive document in the Wolfram language (CDF: https://www.wolfram.com/cdf-player/). We choose this language because it excels at concise expression of mathematics and it supports functional programming well. The ideas presented here can be transcribed into any modern mainstream language that supports closures, even if only approximately. For example, it is easy to write them in C++11 and beyond, in Python, in any modern Lisp such as Clojure, HyLang, LFE, and Common Lisp, not to mention in Haskell, Scala, Erlang, and OCaml. Much can be written without full closures; function pointers will suffice, so they are easy to write in C.

Running Average

Consider a stream of scalar data or ***observations***, zs, and a running average x, a variable meant to suggest a state variable as one might see in a Kalman filter. Our job is to improve the estimate of this state as new observations arrive, one observation at a time.

Our state is actually a pair of the running average x and the running count, n. We write this pair, algebraically,

as {*x*, *n*}. In Wolfram, curly braces enclose sequential (ordered) data, whereas in ordinary mathematical notation, curly braces enclose unordered sets.

Fold, called ***reduce*** or ***aggregate*** in other programming languages, always takes three arguments

- a binary function, the ***accumulator function***
- an initial value of the state
- the sequence of observations

As explained in part 4 of this series (http://vixra.org/abs/1607.0141), there are few restrictions on the sequence of observations: they can be distributed in space (computer memory), in time (asynchronous observable streams), or algorithmically in virtual time (lazy infinite streams). The accumulator function is binary --- it takes two arguments. The first is the current (old) value of the state {*x*, *n*} and the second is the current (new) observation *z*. The accumulator function must return a new value of the state, a new pair in our case. *Fold* iterates the accumulator function over all the observations in the sequence and eventually produces the final value of the state. As stated, obviously, it can only work on finite sequences of observations. Its sister function, ***FoldList***, also known as ***scan*** or ***reductions***, produces all the intermediate states and is more applicable to these situations. There are more subtleties: we can distinguish between left folds and right folds, but these distinctions are not directly relevant to our current exposition.

To derive our recurrence relation, we write the old average as *x*, the new observation as *z*, the old count as *n*, the ***residual*** as the difference between the new observation and the old average, namely as *z* – *x*. The residual tells us how much the new observation differs from the old average. Because we are averaging the observations, we don't expect the new observation to be very different from the average, especially after we have accumulated a lot of observations. Therefore, we can imagine that the residual is proportional to a correction to the old average. The proportionality constant is called the ***gain***.

More precisely, we want the new average as the sum of the old average *x* plus the gain *K* times the residual:

$$x \longleftarrow x + K\,(z - x) \tag{1}$$

This is the specification of a ***filter***, a generic name for almost any procedure that sequentially transforms data, but especially for those that have multiplicative quantities like *K*, identifiable as gains. To solve for *K*, we need only know the definition of the average, now written with subscripts for clarity:

$$x_{n+1} = \frac{1}{n+1}\,(n\,x_n + z) \tag{2}$$

The quantity $n\,x_n$ is the sum of all observations prior to the current one, *z*, so $n\,x_n + z$ is the sum of all $n + 1$ observations seen so far. This sum, divided by by $n + 1$, is the average of all the data seen so far. Note that this definition works even before we have any observations, that is, when $n = 0$. A "divide-by-zero" error is not possible. Therefore, the filter is ***self-starting***.

Combining 1 and 2, we get

$$x_{n+1} = x_n + K\,(z - x_n) = \frac{n\,x_n + z}{n+1} = \frac{n}{n+1}\,x_n + \frac{1}{n+1}\,z \tag{3}$$

Isolating *K* on the left and rearranging:

$$K\,(z - x_n) = \left(\frac{n}{n+1} - 1\right)\,x_n + \frac{1}{n+1}\,z$$

$$K\,z - K\,x_n = \frac{1}{n+1}\,z - \frac{1}{n+1}\,x_n$$

Which is clearly satisfied if and only if

$$\boxed{K = \frac{1}{n+1}} \tag{4}$$

We can let Wolfram check our symbolic arithmetic:

In[2]:=
```
Solve[x + K (z - x) == (n x + z)/(n + 1), K]
```

Out[2]=
$$\left\{\left\{K \to \frac{1}{1+n}\right\}\right\}$$

Folding It

We mechanize the recurrence relation, equation 2, in Wolfram as a single function **cume**, which we can test on sequences of numbers and later deploy verbatim on asynchronous data streams. We present its definition then clarify the notation.

In[3]:=
```
ClearAll[cume];
cume[{x_, n_}, z_] :=
  With[{K = 1/(n + 1)},
   {x + K (z - x), n + 1}];
```

The notation means that `cume[{x_,n_},z_]` is a function of two parameters. It's a binary function, as any foldable accumulator function must be. The first parameter is a pair, `{x_,n_}` denoting the old state, and the second parameter `z_` denotes the new observation. The underscores emphasize that symbols in the parameter position are actually ***pattern variables***: at call sites of the function, the pattern variables must match actual arguments, where they are bound to the values of the actual arguments. Wolfram is explicit, in its notation, about the difference between pattern variables --- with trailing underscores --- and other kinds of variables --- without trailing underscores. In fact, any programming language that does pattern matching must make this distinction, but most hide it from the user, increasing ambiguity. Wolfram's explicitness is refreshing; while it takes a little getting-used-to, it deepens understanding and clarity.

Wolfram will replace the pattern, `cume[{x_,n_},z_]`, when matched to some actual arguments, by the right-hand side of the assignment operator `:=`, which evaluates to the pair `{x+K(z-x),n+1}`, after defining the temporary, local variable `K` as the value $1/(n+1)$, exactly as derived above.

Let's test this on a couple of samples. Start with an initial state --- running average and count --- of `{0,0}`, and accumulate into it the value 1.414. We expect the new state --- the new running average and count --- to be `{1.414, 1}`:

In[5]:=
```
cume[{0, 0}, 1.414]
```

Out[5]=
```
{1.414, 1}
```

If we accumulate another observation of 1.414, we expect the average to be the same and the count to go up

by one. So far, the only tool we have to iterate **cume** is direct application:

```
In[6]:= cume[cume[{0, 0}, 1.414], 1.414]
Out[6]= {1.414, 2}
```

This is where **Fold** comes in:

```
In[7]:= Fold[cume, {0, 0}, {1.414, 1.414}]
Out[7]= {1.414, 2}
```

The last argument to **Fold** is the sequence of observations. We can get more adventurous and compute the mean of a random sequence. First, assign a random sequence of ten numbers between -0.5 and +0.5, inclusive of both ends, to a symbol, **zs**

```
In[8]:= zs = RandomReal[{-0.5, 0.5}, 10]
Out[8]= {-0.175225, 0.00125951, 0.183725, 0.0680458,
  -0.460934, -0.186079, 0.492767, -0.404038, 0.48475, -0.354006}
```

We expect the average of zs to be closer to zero than most of the data:

```
In[9]:= Fold[cume, {0, 0}, zs]
Out[9]= {-0.0349735, 10}
```

Finally we check the result against Wolfram's built-in:

```
In[10]:= Mean[zs]
Out[10]= -0.0349735
```

Running Variance

Variance is a measure of volatility or dispersion. It is defined as the sum of squared residuals of the observations from their mean, divided by the count less one, Bessel's correction (https://en.wikipedia.org/wiki/Bessel%27 s_correction).

We now write the mean of the first n observations as $\bar{z}_n$ instead of as x_n because we want the symbol x to denote the state of our filter, which, this time, will be the running variance.

By definition, the sum of squared residuals, $S_n \stackrel{\text{def}}{=} \sum_{i=1}^{n} (z_i - \bar{z}_n)^2$, is the variance times the length less one. Let's check that assertion with a new fold, this time, over an anonymous function. Recall that the first argument of any fold must be an accumulator function, a function of two arguments: the current accumulation and the new datum. Let's write an anonymous accumulator function that computes the sum of squared residuals. Wolfram lets us write anonymous function values literally, for instance, as $\{s, z\} \mapsto s + (z - \text{Mean[zs]})^2$. A literal function value like this is no different in kind from a literal integer like 42 or a literal sequence like {6, 7, 42}.

Let's parse this literal-function syntax. The first sequence, {s, z}, to the left of the function arrow, ↦, is a sequence of formal parameters. These are not pattern variables in Wolfram, so they don't have underscores. Wolfram does not support pattern-matching for literal functions, therefore the formal parameters should not be pattern variables. The right-hand side of the function arrow is the replacement value we want, presuming substitution of actual-argument values for the formal parameters. In our case, the replacement value is

$s + (z - \text{Mean}[zs])^2$, which clearly accumulates the new squared residual $(z - \text{Mean}[zs])^2$ into the old running sum s, returning the new value of the running sum.

We point out that this literal function is a ***closure*** because it ***closes over*** the variable **zs** (and, more pedantically, even over the functions **Mean**, **+**, and squaring, which, like all functions, anonymous or not, are just values like numbers in any functional programming language). A function is said to ***close over*** a variable if that variable does not appear in the formal parameter list of the function. Such variables are called ***free variables***. That's pretty much all there is to this fancy lingo about closures. Well, there is the somewhat subtle issue of dynamic versus lexical binding, also known as scoping (https://goo.gl/XiOSHx). But this is not germane to this paper and we may assume that all free variables mean just what they appear to mean.

The only thing left for us to do to verify that the sum of squared residuals, $S_n = \sum_{i=1}^{n} (z_i - \bar{z}_n)^2$, is the variance times the length less one is to fold our anonymous, binary, literal function over our observations **zs**:

In[11]:=
```
Fold[{s, z} ↦ s + (z - Mean[zs])^2, 0, zs] == Variance[zs] * (Length[zs] - 1)
```

Out[11]=
```
True
```

Wolfram's double equals is a Boolean operator that produces **True** if its left- and right-hand sides are numerically equal.

We're now equipped to pursue our real objective, to write the running variance as a recurrence relation: the old variance plus a correction that depends only on old values, written as a filter: a gain times some transform of the residual of the observation from its mean.

Start with a recurrence for the sum of squared residuals:

$$S_n = \sum_{i=1}^{n} (z_i - \bar{z}_n)^2 \tag{5}$$

Because our algorithm must run in constant memory, we don't have all the z_i. Just as with the running mean, however, where we implicitly keep the sum of all data as $n\, x_n$, we can find ways to keep the needed sums for S_n. Expand the square:

$$S_n = \sum_{i=1}^{n} \left(z_i^2 - 2\, z_i\, \bar{z}_n + \bar{z}_n^2 \right)$$

Distribute the summation

$$S_n = \sum_{i=1}^{n} z_i^2 - \sum_{i=1}^{n} 2\, z_i\, \bar{z}_n + \sum_{i=1}^{n} \bar{z}_n^2$$

Factor out constants (values that don't depend on the summation index, i):

$$S_n = \sum_{i=1}^{n} z_i^2 - 2\, \bar{z}_n \sum_{i=1}^{n} z_i + \bar{z}_n^2 \sum_{i=1}^{n} 1$$

Observe that $\sum_{i=1}^{n} z_i = n\, \bar{z}_n$, as before with the running mean, and that $\sum_{i=1}^{n} 1 = n$. We get

$$S_n = \sum_{i=1}^{n} z_i^2 - 2\, n\, \bar{z}_n^2 + n\, \bar{z}_n^2 \sum_{i=1}^{n} 1 = \sum_{i=1}^{n} z_i^2 - n\, \bar{z}_n^2 \tag{6}$$

$$S_n = \sum_{i=1}^{n} z_i^2 - 2\, n\, \bar{z}_n^2 + n\, \bar{z}_n^2 \sum_{i=1}^{n} 1 = \boxed{\sum_{i=1}^{n} z_i^2 - n\, \bar{z}_n^2} \tag{7}$$

This is a great result. It means that we can accumulate the running sum-of-squared-residuals just by accumulating the running squares, a trivial extension to accumulating the running sum, and by accumulating the running mean and count, which we already know how to do.

This form suffers from a numerical hazard, however: ***catastrophic cancellation***. Squaring first, then subtracting, can lose significant digits. By writing the sum of squared residuals in recurrent form, as the old value plus a correction, we can subtract first and then square, holding off the numerical dragons just a little longer.

We're looking for a form like this

$$S_{n+1} = S_n + K\, (z - \bar{z}_n)^2$$

Substituting S_n from equation 7 and expanding the square, we get

$$S_{n+1} = \left(\sum_{i=1}^{n} z_i^2 - n\, \bar{z}_n^2 \right) + K\, z^2 - 2\, K\, z\, \bar{z}_n + K\, \bar{z}_n^2 \tag{8}$$

Invoking equation 7 again, this time for $n + 1$, here is the new sum of squared residuals:

$$S_{n+1} = \sum_{i=1}^{n+1} z_i^2 - (n+1)\, \bar{z}_{n+1}^2 = \sum_{i=1}^{n} z_i^2 + z^2 - (n+1)\, \bar{z}_{n+1}^2 \tag{9}$$

The second equality holds because the current observation, z is the same as the $(n + 1)$-st observation, z_{n+1}. Expand $\bar{z}_{n+1}^2$ from its definition in equation 2:

$$\bar{z}_{n+1}^2 = \left(\frac{1}{n+1} (n\, \bar{z}_n + z) \right)^2 = \left(\frac{1}{n+1} \right)^2 (n\, \bar{z}_n + z)^2 = \left(\frac{1}{n+1} \right)^2 \left(n^2\, \bar{z}_n^2 + 2\, n\, \bar{z}_n\, z + z^2 \right)$$

Substituting this form into equation 9 gives

$$S_{n+1} = \sum_{i=1}^{n} z_i^2 + z^2 - \frac{1}{n+1} \left(n^2\, \bar{z}_n^2 + 2\, n\, \bar{z}_n\, z + z^2 \right)$$

$$= \sum_{i=1}^{n} z_i^2 + z^2 - \frac{n^2\, \bar{z}_n^2}{n+1} - \frac{2\, n\, \bar{z}_n\, z}{n+1} - \frac{z^2}{n+1}$$

The easiest way to spot K is to isolate the only terms that depend on z^2:

$$S_{n+1} = \sum_{i=1}^{n} z_i^2 - \frac{n^2\, \bar{z}_n^2}{n+1} - \frac{2\, n\, \bar{z}_n\, z}{n+1} + \left(-\frac{z^2}{n+1} + z^2 \right)$$

$$= \sum_{i=1}^{n} z_i^2 - \frac{n^2\, \bar{z}_n^2}{n+1} - \frac{2\, n\, \bar{z}_n\, z}{n+1} + \left(\frac{n}{n+1}\, z^2 \right)$$

Examining equation 8 suggests that $K = \frac{n}{n+1}$. Does this hold up? The second term, $-\frac{2\, n\, \bar{z}_n\, z}{n+1}$, is clearly $-2\, K\, z\, \bar{z}_n$, so we're left only to check the terms that depend on $\bar{z}_n^2$, namely

$$\left(-n\, \bar{z}_n^2 + K\, \bar{z}_n^2 = -\frac{n^2\, \bar{z}_n^2}{n+1} \right) = \left(-n\, \bar{z}_n^2 + \frac{n}{n+1}\, \bar{z}_n^2 \right) = \left(\frac{-n\,(n+1) + n}{n+1}\, \bar{z}_n^2 \right)$$

and it checks out. We conclude that

$$\boxed{K = \frac{n}{n+1}} \tag{10}$$

Folding It

The following fold recurrently computes mean, count, and variance, using both gains from equations 4 and 10:

In[16]:=
```
ClearAll[cume];
cume[{var_, x_, n_}, z_] :=
  With[{KforMean = 1/(n + 1)},
   With[{KforVariance = n KforMean},
    With[{x2 = x + KforMean (z - x),
      ssr2 = (n - 1) var + KforVariance (z - x)^2},
     {ssr2/Max[1, n], x2, n + 1}]]];
Fold[cume, {0, 0, 0}, zs]
```

Out[18]=
```
{0.118924, -0.0349735, 10}
```

We avoid divide-by-zero when there are no observations by a trick: dividing the sum of squared residuals by the max of 1 and *n*. This trick makes the filter self-starting at the expense of defining an artificial value for the variance after there is only one observation.

Now check that the variance, the first part of the sequence returned by the fold above, equals the value returned by Wolfram's built-in variance function:

In[15]:=
```
Fold[cume, {0, 0, 0}, zs][[1]] === Variance[zs]
```

Out[15]=
```
True
```