Kalman Folding 4: Streams and Observables (WORKING DRAFT)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-05-03 Tue>

Contents

1 Abstract

In *Kalman Folding, Part 1*,[1] we present basic, static Kalman filtering as a functional fold, highlighting the unique advantages of this form for deploying test-hardened code verbatim in harsh, mission-critical environments. In that paper, all examples folded over arrays in memory for convenience and repeatability. That is an example of developing filters in a friendly environment.

Here, we prototype a couple of less friendly environments and demonstrate exactly the same Kalman accumulator function at work. These less friendly environments are

- lazy streams, where new observations are computed on demand but never fully realized in memory, thus not available for inspection in a debugger
- asynchronous observables, where new observations are delivered at arbitrary times from an external source, thus not available for replay once consumed by the filter

[1] B. Beckman, *Kalman Folding, Part 1*, to appear.

Streams are a natural fit for integration of differential equations, which often arise in applications. As such, they enable unique modularization for all kinds of filters, including non-linear Extended Kalman Filters.

The fact that the Kalman accumulator function gives bit-for-bit identical results in all cases gives us high confidence that code developed in friendly environments will behave as intended in unfriendly environments. This level of repeatability is available *only* because of functional decomposition, which minimizes the coupling between the accumulator function and the environment and makes it possible to deploy exactly the same code, without even recompilation, in all environments.

2 Kalman Folding in the Wolfram Language

In this series of papers, we use the Wolfram language[2] because it excels at concise expression of mathematical code. All examples in these papers can be directly transcribed to any modern mainstream language that supports closures. For example, it is easy to write them in C++11 and beyond, Python, any modern Lisp, not to mention Haskell, Scala, Erlang, and OCaml. Many can be written without full closures; function pointers will suffice, so they are easy to write in C. It's also not difficult to add extra arguments to simulate just enough closure-like support in C to write the rest of the examples in that language.

In *Kalman Folding*,[1] we found the following elegant formulation for the accumulator function of a fold that implements the static Kalman filter:

$$\text{kalmanStatic}\left(\mathbf{Z}\right)\left(\{\mathbf{x},\mathbf{P}\},\{\mathbf{A},z\}\right)=\{\mathbf{x}+\mathbf{K}\ \left(z-\mathbf{A}\,\mathbf{x}\right),\mathbf{P}-\mathbf{K}\,\mathbf{D}\,\mathbf{K}^{\intercal}\} \tag{1}$$

where

$$\mathbf{K}=\mathbf{P}\,\mathbf{A}^{\intercal}\,\mathbf{D}^{-1} \tag{2}$$

$$\mathbf{D}=\mathbf{Z}+\mathbf{A}\,\mathbf{P}\,\mathbf{A}^{\intercal} \tag{3}$$

and all quantities are matrices:

- z is a $b\times 1$ column vector containing one multidimensional observation
- $\mathbf{x}$ is an $n\times 1$ column vector of *model states*
- $\mathbf{Z}$ is a $b\times b$ matrix, the covariance of observation noise
- $\mathbf{P}$ is an $n\times n$ matrix, the theoretical covariance of $\mathbf{x}$
- $\mathbf{A}$ is a $b\times n$ matrix, the *observation partials*
- $\mathbf{D}$ is a $b\times b$ matrix, the Kalman denominator
- $\mathbf{K}$ is an $n\times b$ matrix, the Kalman gain

[2] `http://reference.wolfram.com/language/`

In physical or engineering applications, these quantities carry physical dimensions of units of measure in addition to their matrix dimensions as numbers of rows and columns. 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

$$\begin{array}{lcccc} [[\mathbf{Z}]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\ [[\mathbf{A}]] & = & (& \mathcal{Z}/\mathcal{X} & b \times n \ ) \\ [[\mathbf{P}]] & = & (& \mathcal{X}^2 & n \times n \ ) \\ [[\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\ [[\mathbf{D}]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\ [[\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{X}\mathcal{Z} & n \times b \ ) \\ [[\mathbf{K}]] & = & (& \mathcal{X}/\mathcal{Z} & n \times b \ ) \end{array} \tag{4}$$

Dimensional arguments, regarding both matrix dimensions and physical dimensions, are invaluable for checking code and derivations in this topic at-large.

2.1 A Test Example

In the following example, the observations $\mathbf{z}$ are 1×1 matrices, equivalent to scalars, so $b = 1$.

The function in equation 1 *lambda-lifts*[3] $\mathbf{Z}$, meaning that it is necessary to call *kalmanStatic* with a constant $\mathbf{Z}$ to get the actual accumulator function used in folds. This is desirable to reduce coupling between the accumulator function and its calling environment.

In Wolfram, this function is

```
kalmanStatic[Zeta_][{x_, P_}, {A_, z_}] :=
 Module[{D, K},
  D = Zeta + A.P.Transpose[A];
  K = P.Transpose[A].Inverse[D];
  {x2 + K.(z - A.x), P - K.D.Transpose[K]}]
```

We test it on a small case

```
Fold[kalmanStatic[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
~~>
```

[3] https://en.wikipedia.org/wiki/Lambda_lifting

$$\mathbf{x} = \begin{bmatrix} -2.97423 \\ 7.2624 \\ -4.21051 \\ -4.45378 \end{bmatrix} \tag{5}$$

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

expecting results within one or two standard deviations of the ground truth $\aleph = \begin{bmatrix} -3 & 9 & -4 & -5 \end{bmatrix}^\intercal$, where the standard deviations can be found as square roots of the diagonal elements of $\mathbf{P}$. For details about this test case, see the first paper in the series, *Kalman Folding, Part 1*.[1]

Below, we reproduce these values exactly, to the bit level, by running *kalmanStatic* over lazy streams and asynchronous observables.

3 Types for Kalman Folding

Kalman and all its variants are examples of *statistical function inversion.* We have models that predict outcomes from inputs; we observe outcomes and want estimates of the inputs. Structurally, all such incremental model inversions take a pair of a state estimate (with uncertainty) and an observation, and produce a new state estimate (with uncertainty). Such an inverted model has signature, using a type notation similar to that of Haskell or Scala

$$\text{inverted-model}\,[\mathsf{S},\mathsf{T}] :: (\mathsf{S} \to \mathsf{T} \to \mathsf{S})$$

where the return type is on the far right and the other types that appear before arrows are the types of input arguments. This function signature is exactly that required for the first argument of a functional fold (more precisely, a *left* fold). The signature of *fold* is as follows:

$$\text{fold}\,[\mathsf{S},\mathsf{T}] :: (\mathsf{S} \to \mathsf{T} \to \mathsf{S}) \to \mathsf{S} \to \text{Sequence}\,[\mathsf{T}] \to \mathsf{S}$$

Read this, abstractly, as follows

> *Fold* over types S and T is a function that takes three arguments:
>
> 1. another function (called the *accumulator function*)
> 2. an initial instance of type S
> 3. a sequence of instances of type T
>
> and produces an instance of type S. The accumulator function, in turn, is a binary function that takes an S and a T and produces an S.

More concretely, In the context of Kalman filtering:

$$\text{AccumulatorFunction} :: \text{Accumulation} \to \text{Observation} \to \text{Accumulation}$$

where the types *Accumulation* and *Observation* are arbitrary.

It's the job of *Fold* to pass the elements of the input sequence to the accumulator function one observation at a time, and to maintain and ultimately return the final accumulation. The second argument to *Fold* is the desired, initial value of the accumulation. The third and final argument to *Fold* is the sequence of observations, of type Sequence[Observation]

Fold looks like a trinary function of an accumulator function, an initial accumulation, and a sequence, yielding an accumulation. Folds thus have the following type:

$$\text{Fold} :: \text{AccumulatorFunction} \to \text{Accumulation} \to \text{Sequence}[\text{Observation}] \to \text{Accumulation}$$

where *Sequence* can be *List*, *Stream*, *Observable*, or any type that can be accessed sequentially.

4 Over Lazy Streams and Asynchronous Observables

The accumulator function knows nothing about the source of the observations. If we can figure out how to implement *Fold* and *FoldList* for things other than *List*, we will have Kalman filtering over those sources, too.

The following are research-grade sketches of implementations of *Fold* over lazy streams[4] and asynchronous observables.[5] They provide just enough to support the Kalman-folding examples.

4.1 Folding Over Lazy Streams

Represent a lazy stream as a pair of a value and a *thunk* (function of no arguments).[6] The thunk must produce another lazy stream when called. Such a stream can be infinite in abstract length because the elements of the stream are only concretized in memory when demanded by calling thunks.

Streams are a natural fit for integrals of differential equations. We see in other papers of this series how we an use them to deeply modularize filters over rich non-linear models. In this paper, we show only how to fold a linear Kalman filter over a stream.

By convention, a finite stream has a `Null` thunk at the end. Thus, the empty stream, obtained by invoking such a thunk, is `Null[]`, with square brackets denoting invocation with no arguments.

One of Wolfram's notations for a literal thunk is an expression with an ampersand in postfix position. An ampersand turns the expression to its left into a thunk. For instance, here's a function that returns an infinite stream of natural numbers starting at `n`:

```
integersFrom[n_Integer] := {n, integersFrom[n + 1] &}
```

Calling, say, `integersFrom[42]` produces `{42, integersFrom[42 + 1]&}`, a pair of an integer, 42, and another stream, `integersFrom[42+1]&`. We get the stream by extracting the second part of the pair *via* Wolfram's double-bracket notation

```
integersFrom[42][[2]] ~~> integersFrom[42 + 1]&
```

[4] `http://www1.cs.dartmouth.edu/~doug/music.ps.gz`

[5] `http://introtorx.com/`

[6] This is quite similar to the standard — not Wolfram's — definition of a list as a pair of a value and of another list.

and then call it with empty brackets (it's a thunk, and takes no arguments):

```
integersFrom[42][[2]][] ~~> {43, integersFrom[43 + 1]&}
```

and so on. We can get a few more by repeating the process

```
integersFrom[42][[2]][][[2]][][[2]][] ~~> {45, integersFrom[45 + 1]&}
```

but the best way to extract values from streams is to write recursive functions to demand any number of elements from the head. The variety of such functions, which include *map*, *select*, *fold*, is well known, large, and identical across lists, streams, observables, and, in fact, any collection that can support a *next* operator. A good, contemporary full-service library for collection types is LINQ's Standard Query Operators (SQO),[7]. If building up a library from the present prototype level into something of product grade, presentable to intolerant users, the SQO are an excellent framework to emulate.

As another example, the following function, when called with an appropriate input, say the 2×2 identity matrix, returns a lazy stream of matrices full of Fibonacci numbers:

```
fs[f_] := {f, fs[{{0, 1}, {1, 1}}.f] &}
```

Here is an explicit invocation a few values down:

```
fs[IdentityMatrix[2]][[2]][][[2]][][[2]][][[2]][][[2]][][[2]][][[
        2]][][[2]][][[2]][][[2]][]
~~>
```

$$\left\{\begin{pmatrix}34 & 55\\ 55 & 89\end{pmatrix}, \mathrm{fs}\left[\begin{pmatrix}34 & 55\\ 55 & 89\end{pmatrix}\cdot\begin{pmatrix}0 & 1\\ 1 & 1\end{pmatrix}\right]\ \&\right\}$$

the point being that lazy streams are versatile.

We now write bi-directional conversions between streams and lists so we can test an example, then we write *foldStream*.

4.1.1 Disperse :: List → Stream

We'll need a way to convert a list into a stream. There are three cases: an empty list, a singleton list, and the inductive or recursive case.

```
disperse[{}] := Null[]; (* empty list yields empty stream       *)
disperse[{x_}] := {x, Null}; (* the stream for a singleton list *)
disperse[{v_, xs__}] := {v, disperse[{xs}] &}; (* recursion     *)
```

4.1.2 Reify :: Stream → List

We need to go the other way, too; don't call this on a stream of infinite length:

```
reify[Null[]] := {};          (* produce empty list from empty stream *)
rify[{v_, Null}] := {v};      (* singleton list from singleton stream *)
reify[{v_, thunk_}] := Join[{v}, reify[thunk[]]]; (* recursion       *)
```

[7] LINQ's Standard Query Operators

4.1.3 foldStream

Our equivalent for Wolfram's *FoldList* is *foldStream*.[8] Its type is similar

$$\begin{aligned}\text{foldStream} :: \text{AccumulatorFunction} &\rightarrow \text{Accumulation} \\ \rightarrow \text{Stream}\,[\text{Observation}\,] &\rightarrow \text{Stream}\,[\text{Accumulation}\,]\end{aligned}$$

Here is an implementation:

```
foldStream[f_, s_, Null[]] := (* acting on an empty stream *)
  {s, Null}; (* produce a singleton stream containing 's'  *)
foldStream[f_, s_, {z_, thunk_}] :=
  (* pass in a new thunk that recurses on the old thunk    *)
  {s, foldStream[f, f[s, z], thunk[]] &};
```

4.1.4 Test

Test it over the *dispersion* of the example data:

```
foldStream[
  kalmanStatic[IdentityMatrix[1]], (* same 'kalmanStatic;' no changes *)
  {ColumnVector[{0, 0, 0, 0}], IdentityMatrix[4]*1000.0},
  disperse[{{{{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}}}]
  ] // reify
```

The only changes to the earlier fold over lists is the initial call of *disperse* to convert the test case into a stream, and the final postfix call `// reify` to turn the result back into a list for display. The final results are identical to those in equation 5, but we see all the intermediate results as well, confirming that Kalman folds over observations one at a time. We would have seen exactly the same output had we called *FoldList* instead of *Fold* over lists above.

[8] The initial uncial (lower-case) letter signifies that *we* wrote this function; it wasn't supplied by Wolfram.

$$
\left(\begin{array}{cc}
\begin{bmatrix}0\\0\\0\\0\end{bmatrix} & \begin{bmatrix}1000. & 0 & 0 & 0\\ 0 & 1000. & 0 & 0\\ 0 & 0 & 1000. & 0\\ 0 & 0 & 0 & 1000.\end{bmatrix}\\
\begin{bmatrix}-2.28214\\0\\0\\0\end{bmatrix} & \begin{bmatrix}0.999001 & 0 & 0 & 0\\ 0 & 1000. & 0 & 0\\ 0 & 0 & 1000. & 0\\ 0 & 0 & 0 & 1000.\end{bmatrix}\\
\begin{bmatrix}-2.28299\\-0.849281\\-0.849281\\-0.849281\end{bmatrix} & \begin{bmatrix}0.998669 & -0.332779 & -0.332779 & -0.332779\\ -0.332779 & 666.889 & -333.111 & -333.111\\ -0.332779 & -333.111 & 666.889 & -333.111\\ -0.332779 & -333.111 & -333.111 & 666.889\end{bmatrix}\\
\begin{bmatrix}-2.28749\\1.40675\\-5.35572\\1.40675\end{bmatrix} & \begin{bmatrix}0.998004 & 0 & -0.997506 & 0\\ 0 & 500.125 & 0 & -499.875\\ -0.997506 & 0 & 1.49676 & 0\\ 0 & -499.875 & 0 & 500.125\end{bmatrix}\\
\begin{bmatrix}-2.29399\\7.92347\\-5.34488\\-5.1154\end{bmatrix} & \begin{bmatrix}0.997508 & 0.49762 & -0.996678 & -0.498035\\ 0.49762 & 1.3855 & -0.829836 & -0.719881\\ -0.996678 & -0.829836 & 1.49538 & 0.830528\\ -0.498035 & -0.719881 & 0.830528 & 0.553787\end{bmatrix}\\
\begin{bmatrix}-2.97423\\7.2624\\-4.21051\\-4.45378\end{bmatrix} & \begin{bmatrix}0.485458 & 0 & -0.142778 & 0\\ 0 & 0.901908 & 0 & -0.235882\\ -0.142778 & 0 & 0.0714031 & 0\\ 0 & -0.235882 & 0 & 0.0693839\end{bmatrix}
\end{array}\right) \tag{6}
$$

4.2 Folding Over an Asynchronous Observable

Just as *FoldList* produces a list from a list, and *foldStream* produces a stream from a stream, *foldObservable* produces an observable from an observable. Its full signature is

$$
\begin{aligned}
&\text{foldObservable} :: \text{AccumulatorFunction} \to \text{Accumulation}\\
&\to \text{Observable}\,[\,\text{Observation}\,] \to \text{Observable}\,[\,\text{Accumulation}\,]
\end{aligned}
$$

Lists provide data elements distributed in space (memory). Lazy streams provide data in constant memory, but distributed in a kind of virtual time, delivered when demanded, the way a debugger fakes time. Observables provide data elements distributed asynchronously in real time. To consume elements of an observable, subscribe an observer to it. An observer has a callback function, and the observable will invoke the callback for each observation, asynchronously, as the observation arrives. The callback function takes a single argument that receives the observation.

We do not develop observables fully, here. For that, see a reference like Campbell's *Intro to Rx*.[5] Instead, we content ourselves with just enough to demonstrate Kalman folding over them and, as with lazy streams, a way to get back and forth from lists.

We model observables as stateful thunks that produce new values every time they're invoked, then invoke the thunks inside asynchronous Wolfram tasks that start at the moment some observer subscribes.[9]

4.2.1 Subscribe :: Observable → Observer → Null

Wolfram supplies a primitive, *RunScheduledTask*, for evaluating expressions asynchronously, once per second by default. The expression that we pass to *RunScheduledTask*, just calls the observer on the evaluated observable:

```
subscribe[observable_, observer_] :=
  RunScheduledTask[observer[observable[]]];
```

4.2.2 Dispense :: List → Observable

The following is a specification of a task to run. Nothing happens till you subscribe something to it.

```
dispense[aList_List] :=
 Module[{state = aList},
  If[{} === state,
    Null, (* empty obs from empty list *)
    (state = Rest[state]; First[state]]);] &]
```

4.2.3 Harvest :: Observable → List

Set up a conventional, external variable, `r$`, so that we can interactively look at the results in a Wolfram `Dynamic[r$]` form. Our *harvest* subscribes an observer that appends observations to a list held in `r$`. Semicolon-separated expressions are sequenced, as with Scheme's `begin` or Lisp's `progn`.

```
harvest[obl_] :=
  (r$ = {};
   subscribe[obl, Function[v, If[v =!= Null, AppendTo[r$, v]]]]);
```

We must eventually clean up the tasks and the external variable.

```
cleanup[] := (ClearAll[r$];
              RemoveScheduledTask[ScheduledTasks[]];);
```

[9] This convention only models so-called *cold observables*, but it's enough to demonstrate Kalman's working over them.

4.2.4 foldObservable

The concrete type of *foldObservable* is obvious: just replace *Stream* with *Observable* in a copy of the type of *foldStream*.

$$\text{foldObservable} :: \text{AccumulatorFunction} \rightarrow \text{Accumulation}$$
$$\rightarrow \text{Observable}[\text{Observation}] \rightarrow \text{Observable}[\text{Accumulation}]$$

One might ask about the appropriate generalization of higher-order types like this, where we could go up a level, parameterize on types like *Stream* and *Observable*, and make the concrete types of *foldStream* and *foldObservable* instances of that higher, parameterized type. This is a sensible question, and the answer leads to category theory and monads,[10] out of scope for this paper.

This implementation isn't hygeinic: it uses global variables (suffixed with `$` signs). It's just enough to test Kalman folding over observables.

```
foldObservable[f_, s_, obl_] :=
 Module[{newObl, s$ = s},
  newObl[] := With[{result = s$},
    s$ = f[s$, obl[]];
    result];
  newObl] (* return new observable *)
```

4.2.5 Test

The following call has the same shape as our call of *foldStream* above, except calling *dispense* instead of *disperse* and *harvest* instead of *reify*.

```
Dynamic[r$]
foldObservable[
  kalmanStatic[IdentityMatrix[1]],
  {ColumnVector[{0, 0, 0, 0}], IdentityMatrix[4]*1000.0},
  dispense[{{{{1,  0., 0.,  0.}}, { -2.28442}},
            {{{1,  1., 1.,  1.}}, { -4.83168}},
            {{{1, -1., 1., -1.}}, {-10.46010}},
            {{{1, -2., 4., -8.}}, {  1.40488}},
            {{{1,  2., 4.,  8.}}, {-40.8079}}}]
  ] // harvest;
r$
```

The results are exactly the same as in equation 6.

5 Concluding Remarks

With prototypes for *foldStream* and *foldObservable*, we have demonstrated Kalman folding with exactly the same accumulator function over wildly different data-delivery environments. This

[10] `https://en.wikipedia.org/wiki/Monad`

demonstrates the primary thesis of this series of papers: that writing filters as functional folds enables verbatim deployment of code in both friendly, synchronous environments with all data in memory, and unfriendly asynchronous environments using only constant memory. Verbatim means with no changes at all, not even recompilation.

We have tested these prototypes against bigger examples like the tracking example[11] and the accelerometer example,[1] and there are no surprises. Emacs 24.5.1 (Org mode 8.3.4)

[11] B. Beckman, *Kalman Folding 2: Tracking and System Dynamics*, To appear.