Kalman Folding 7: A Small Streams Library (Review Draft)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-05-03 Tue>

Contents

1 Abstract

In *Kalman Folding 5: Non-Linear Models and the EKF*,[1] we present an Extended Kalman Filter as a fold over a lazy stream of observations that uses a nested fold over a lazy stream of states to

[1]B. Beckman, *Kalman Folding 5: Non-Linear Models and the EKF*, to appear.

integrate non-linear equations of motion. In *Kalman Folding 4: Streams and Observables*,[2] we present a handful of stream operators, just enough to demonstrate Kalman folding over observables.

In this paper, we enrich the collection of operators, adding *takeUntil*, *last*, and *map*. We then show how to use them to integrate differential equations in state-space form in two different ways and to generate test cases for the non-linear EKF from paper 5.

2 Kalman Folding in the Wolfram Language

In this series of papers, we use the Wolfram language[3] because it supports functional programming and 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.

3 Review of Stream Basics

From paper 4, we briefly review the following basics:

Represent a lazy stream as a pair, in curly braces, of a value and a *thunk* (function of no arguments).[4] 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.

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 empty square brackets denoting invocation with no arguments.

A *finite stream* is one that eventually produces `Null[]`.

One of Wolfram's notations for a literal thunk is an expression with an ampersand at the end. An ampersand turns the expression to its left into a thunk.

Here is 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 stream consisting of an integer, 42, and another stream, `integersFrom[42+1]&`.

4 Stream Operators

The following are just sketches, suitable for research but not for release to third-party users, who will need packaging, namespacing, documentation, and error handling. In fact, it is easy to construct pathological cases that circumvent the stated semantics for these implementations, which are designed only to illustrate the concepts and to support the examples in this series of papers.

[2]B. Beckman, *Kalman Folding 4: Streams and Observables*, to appear.

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

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

Furthermore, this is just a representative set found most useful while developing Kalman folding, not anything like a complete set. It turns out that nearly identical sets of operators can be developed for any kind of sequential collection. Haskell[5] has operators for lists, Rx[6] has a large set for observables, and LINQ[7] has sets for many types including SQL tables. Haskell also includes mechanisms[8, 9] for abstracting the underlying collection, making it easy to implement a standardized set of operators for any new, user-defined collection type.

4.1 Extract :: Stream $\to$ Value

Extract pulls the n-th element, 1-indexed, from a given stream. Its run time is $O(n)$ and its space consumption is constant, $O(1)$.

First, a base case. Extracting any element from the empty stream `Null[]` should produce nothing, represented in Wolfram by `Null` without the brackets, which conveniently does not print in a Mathematica notebook.

```
extract[Null[], _] := Null;
```

Another base case is to get the first value from a stream. This overload *pattern-matches*[10] or *destructures* its stream input, instantiating the variables `v` and `thunk` to the two components of that input. It then returns the value `v`.

```
extract[{v_, thunk_}, 1] := v;
```

The final, recursive case extracts the $n-1$-th element from the tail of the input stream by discarding the current value `v` and invoking the `thunk`. This runs in constant memory when the programming language supports tail-recursion, as Wolfram does.[11] If the programming language does not, `extract` should be implemented with a loop.

The notation `/;` precedes a conditional expression in the scope of the argument list. In most other languages, this would be implemented by a conditional test in the body of the function.

```
extract[{v_, thunk_}, n_Integer /; n > 1] := extract[thunk[], n - 1];
```

Now we can get the 630000-th integer efficiently in space, if not terribly quickly:

```
Block[{$IterationLimit = Infinity},
  extract[integersFrom[1], 630000]] // AbsoluteTiming
~~>
{1.47735 second, 630000}
```

Without tail recursion, this would exceed the system's stack depth.

[5] `http://learnyouahaskell.com/higher-order-functions`
[6] `http://introtorx.com/`
[7] LINQ's Standard Query Operators
[8] `https://en.wikipedia.org/wiki/Monad`
[9] See Haskell's type classes
[10] `http://tinyurl.com/j5jzy69`
[11] `https://en.wikipedia.org/wiki/Conditional_term_rewriting`

4.2 Disperse :: List → Stream

We'll need a way to convert a whole finite 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     *)
```

4.3 Reify :: Stream → List

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

```
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       *)
```

Reify undoes *disperse*:

```
reify@disperse@{1, 2, 3}
~~> {1, 2, 3}
```

4.4 Take :: Stream → FiniteStream

Infinite streams are very important, but we frequently want finite subsets so that we don't have to explicitly extract values by index. *Take* takes a stream and an element count and produces another stream that eventually yields `Null[]`, that is, a finite stream. Because the streams are lazy, *take* doesn't actually run until elements are demanded, say by *extract*, *last*, or *reify*.

```
take[_, 0] := Null[];
take[Null[], _] := Null[];
take[{v_, thunk_}, 1] := {v, Null};
take[{v_, thunk_}, n_Integer /; n > 1] := {v, take[thunk[], n - 1] &};
```

Produce a finite stream of three integers; extract the first value:

```
extract[take[integersFrom[1], 3], 1]
~~> 1
```

and the last value:

```
extract[take[integersFrom[1], 3], 3]
~~> 3
```

If we extract too far into a finite stream, we get `Null`, which doesn't print to the notebook:

```
extract[take[integersFrom[1], 3], 4]
~~>
```

4.5 TakeUntil :: Stream → Predicate → Stream

TakeUntil produces a new stream that produces elements from the original stream, evaluating the predicate on them until it produces `True`, at which point it permanently produces the empty stream `Null[]`.

```
takeUntil[Null[], _] := Null[];
takeUntil[{v_, thunk_}, predicate_] /; predicate[v] := Null[];
takeUntil[{v_, thunk_}, predicate_] := {v, takeUntil[thunk[], predicate] &};
```

```
reify[takeUntil[integersFrom[1], # >= 3 &]]
~~> {1, 2, 3}
```

4.6 MapStream :: Stream → UnaryFunction → Stream

MapStream converts a stream into another stream of equal length by applying the given unary function to the elements one at a time. Because it converts a lazy stream to a lazy stream, it is safe to apply to infinite streams: nothing happens until elements are demanded.

```
ClearAll[mapStream];
mapStream[Null[], _] := Null[];
mapStream[{v_, thunk_}, f_] := {f[v], mapStream[thunk[], f] &};
```

Here we map the unary function `#^2 &`, which squares its single argument `#`, over a finite sub-stream of the integers.

```
reify@mapStream[take[integersFrom[1], 3]]
~~> {1, 4, 9}
```

4.7 Last :: Stream → Value

Last produces the last value in a finite stream without an explicit index. It requires tail recursion to run in constant memory.

```
last[Null[]] := Null;
last[{v_, thunk_} /; thunk[] === Null[]] := v;
last[{v_, thunk_}] := last[thunk[]];
```

Called on an empty stream, *last* produces `Null`, which does not print.

```
last@disperse[{}]
~~>
```

Otherwise, it produces the last element, even of a very long finite stream:

```
Block[{$IterationLimit = Infinity},
  last@take[integersFrom[1], 630000]] // AbsoluteTiming
~~> {4.72633 sec, 630000}
```

The at-sign `@` is Wolfram's prefix form for function invocation; `f@x` is the same as `f[x]`.

4.8 foldStream

Our equivalent for Wolfram's *FoldList* is *foldStream*.[12] Its type is similar

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

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[]] &};
```

and an example that produces the Fibonacci numbers in pairs:

```
allFibs = foldStream[
   Function[{s, z}, {s[[2]], s[[1]] + s[[2]]}],
   {0, 1},
   integersFrom[0]];

Transpose@reify@[take[allFibs, 11]]
~~>
```

$$\begin{pmatrix} 0 & 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 \\ 1 & 1 & 2 & 3 & 5 & 8 & 13 & 21 & 34 & 55 & 89 \end{pmatrix}$$

5 EKF

The EKF developed in paper 5 is

```
  EKFDrag[sigmaXi_, Zeta_, Phi_, Xi_, integrator_, fdt_, idt_]
   [{x_, P_}, {t_, A_, z_}] :=
   Module[{x2, P2, D, K},
    x2 = last[takeUntil[foldStream[integrator, {t, x},
         dragDStream[{idt, t, dragD}]],
        First[#] > t + fdt &]][[2]];
    P2 = sigmaXi^2 Xi[fdt, x] + Phi[fdt, x].P.Transpose[Phi[fdt, x]];
    D = Zeta + A.P2.Transpose[A];
    K = P2.Transpose[A].inv[D];
    {x2 + K.(z - A.x2), P2 - K.D.Transpose[K]}];
```

The EKF integrates the equations of state evolution, which can be arbitrarily nonlinear, by folding an integrator over a stream `dragDStream`. The integrator operates on a time increment `idt`,

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

which is often smaller than the overall update period `fdt` of the EKF. The last element of the integrated stream is collected and used as the state update for the filter.

The integrated stream advances time and passes through to the integrator a function `Dx` that produces differential increments from the state and the time. In our example, `Dx` is `dragD`, which computes the height `x` of an object falling at speed `v` and experiencing aerodynamic drag. Don't confuse this `x` with the vector `x` that represents the state in the integrators. It's difficult not to run out of symbols.

```
  dragDStream[Delta : {dt_, t_, Dx_}] :=
    {Delta, dragDStream[{dt, t + dt, Dx}] &};
  dragD[{x_, v_}, t_] := {v, g (A Exp[-x/k] v^2/(2. beta) - 1)};
```

The additional functions `Phi` and `Xi` use linear approximations of the equations of state evolution to advance the gain `K` and the covariance `P`. They are explained in paper 5.

The simplest integrator is the Euler integrator, which updates a state with its derivative times a small interval of time. This is a binary function, like all accumulator functions for folds, that takes an accumulation and an observation and produces a new accumulation. In our case, the accumulation is a pair of a scalar time `t` and a vector state `x`, and the observation is a triple of a time increment `dt`, a time `t`, and the function `Dx` that produces differential increments.

```
eulerAccumulator[{t_, x_}, {dt_, t_, Dx_}] :=
  {t + dt, x + dt Dx[x, t]};
```

Much better numerics can be achieved with the Runge-Kutta integrators, which are drop-in replacements for the Euler integrator at the cost of calling `Dt` more often:

```
rk2Accumulator[{t_, x_}, {dt_, t_, Dx_}] :=
  With[{dx1 = dt Dx[x, t]},
   With[{dx2 = dt Dx[x + .5 dx1, t + .5 dt]},
    {t + dt, x + (dx1 + dx2)/2.}]];
rk4Accumulator[{t_, x_}, {dt_, t_, Dx_}] :=
  With[{dx1 = dt Dx[x, t]},
   With[{dx2 = dt Dx[x + .5 dx1, t + .5 dt]},
    With[{dx3 = dt Dx[x + .5 dx2, t + .5 dt]},
     With[{dx4 = dt Dx[x + dx3, t + dt]},
      {t + dt, x + (dx1 + 2. dx2 + 2. dx3 + dx4)/6.}]]]];
```

6 Testing the EKF

We test the EKF by folding it over another lazy stream — a stream of observation packets `{t, A, z}` of time `t`, model partial derivatives `A`, and observations *per-se* `z`. Unlike the filter itself, the test code does not run in constant memory. It doesn't have to — its purpose is to assist the verification of the filter by creating sample data, statistics, and plots. It does so by reifying some finite substreams of infinite streams.

First, we set up some constants

```
With[{nStates = 2, nIterations = 10},
 With[{sigmaZeta = 25., sigmaXi = 0.0, t0 = 0., t1 = 30.,
   filterDt = 0.1, integrationDt = 0.1},
  With[{x0 = 200000, v0 = -6000, Zeta = sigmaZeta^2 id[1],
    P0 = 1000000000000 id[nStates]},
```

We then build some fake data by building a lazy stream that integrates the equations of motion, producing an infinite stream of time-state pairs starting with `{t0, {x0, v0}}`, where `t0` is the initial scalar time and `{x0, v0}` is the initial vector state:

```
   Module[{fakes},
    fakes[] := foldStream[rk4Accumulator, {t0, {x0, v0}},
      dragDStream[{filterDt, t0, dragD}]];
```

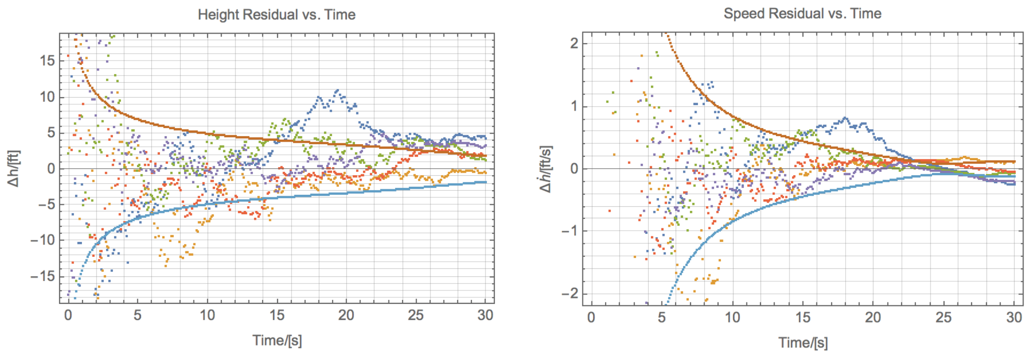

Figure 1: RK-2 integrator, `idt` $= 0.001$ sec, $\sigma_\zeta = 25$ ft; also RK-4 integrator, `idt` $= 0.1$ sec

We now set up some variables to hold the results of multiple iterations of the integration. We use these variables to produce the statistical plots in paper 5, one of which we reproduce here in figure 1. The variables are:

`ffs` a finite substream of the fakes, pairs of times and states

`rffs` an array: the reification of `ffs`

`ts` an array of times gotten by mapping the function `pick[1]` over `rffs` using Wolfram's built-in mapping operator `/@` for reified lists.

`txs` an array of ground-truth values for the first state component `x`, the height of the falling object, for computing residuals

`tvs` an array of ground-truth values for the second state component `v`, the speed of the falling object, for computing residuals

`xss, vss` parallel arrays of arrays of heights and speeds. The outer array has length `nIterations` and is built by mapping (using Wolfram's built-in `Map`) a function over `Range[nIterations]`, a list of the integers $1, 2, \ldots,$ `nIterations`. The inner arrays have the same length as `ts`, for plotting. These can be fed straight into Wolfram's plotting functions.

`xvs, ps` parallel arrays of vector states `{x, v}` and covariance matrices gotten by folding the EKF over a stream built by `mapStream`-ing a function over the finite fakes stream `ffs`. That function picks the times (from the first element of its argument `#` via `#[[1]]`) and the heights (from element 2, 1 its argument `#` via `#[[2, 1]]`) from the finite fakes `ffs` and builds a stream of observation packets with the constant, 1×2 matrix $\mathbf{A} =$ `{{1, 0}}`.

```
SeedRandom[44];
Module[{ffs, rffs, ts, txs, tvs, xss, vss, xvs,
  ps, sigmaxs, sigmavs},
 xss = ConstantArray[0, nIterations];
 vss = ConstantArray[0, nIterations];
 ffs = takeUntil[fakes[], First@# > t1 &];
 rffs = reify@ffs;
 ts = pick[1] /@ rffs;
 txs = pick[2, 1] /@ rffs;
 tvs = pick[2, 2] /@ rffs;
 {xss, vss} = Transpose@Map[
    ({xvs, ps} = Transpose@Rest@reify@foldStream[
          EKFDrag[sigmaXi, Zeta, Phi, Xi,
            rk4Accumulator, filterDt, integrationDt],
          {{0, 0}, P0},
          mapStream[ffs,
            {#[[1]], { {1, 0} }, #[[2, 1]] + gen[Zeta]} &]];
      sigmaxs = Sqrt[pick[1, 1] /@ ps;
      sigmavs = Sqrt[pick[2, 2] /@ ps;
      Transpose@xvs) &,
    Range[nIterations]];
```

Some minor manipulation of these arrays suffice to produce a plot like figure 1.

This test harness uses many of the stream operators in the little library, namely `takeUntil`, `reify`, `foldStream`, and `mapStream`, but the EKF does not know and cannot detect that it's being called through lazy streams. This is one of the secrets of Kalman folding that allows code to be tested in one environment and moved verbatim into other environments. It's even feasible to change integrators at run time through a functional shim. The only thing EKF knows is that it's internally stream-folding an integrator provided by its caller through a fixed interface contract.

7 Concluding Remarks

Lazy infinite streams are one of the kinds of collections supported by Kalman folding. They afford space-efficient integration of differential equations, but also concise and elegant test fixtures. As with all Kalman-folding scenarios, the code-under-test can be moved verbatim, without even recompilation, from the test environment to production and embedded environments.