Kalman Folding 5: Non-Linear Models and the EKF (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 *Kalman Folding 2: Tracking*,[2] we reproduce a tracking example from the literature, showing that these advantages extend to time-dependent, linear models. Here, we extend that example further to include aerodynamic drag, making the model nonlinear. We must change the Kalman filter itself to handle such problems. The resulting class of filters are called Extended Kalman Filters or EKFs. Other papers in this series feature applications of EKFs to a variety of problems including navigation and pursuit.

The particular EKF designed here includes integration of non-linear equations of motion. We integrate these equations by folding over a lazy stream that generates, on demand, differential updates to the solution. Folds over lazy streams were introduced in *Kalman Folding 4: Streams and*

[1] B. Beckman, *Kalman Folding, Part 1*, to appear.
[2] B. Beckman, *Kalman Folding 2: Tracking*, to appear.

Observables[3] where we used them to step a static Kalman filter over observations. They also afford a constant-memory representation for solutions of differential equations, making them suitable for integration components in constant-memory filters.

2 Kalman Folding in the Wolfram Language

In this series of papers, we use the Wolfram language[4] 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 2: Tracking*,[2] we found the following formulation for the accumulator function of a fold that implements the linear dynamic Kalman filter, that is, a filter that can track states that evolve with time[5] according to a linear transformation.

$$\begin{gathered}\text{kalmanDynamic}\left(\{\mathbf{x},\mathbf{P}\},\{\mathbf{Z},\boldsymbol{\Xi},\boldsymbol{\Phi},\boldsymbol{\Gamma},\mathbf{u},\mathbf{A},\mathbf{z}\}\right)= \\ \left\{\mathbf{x}_2+\mathbf{K}\left(\mathbf{z}-\mathbf{A}\,\mathbf{x}_2\right),\quad \mathbf{P}_2-\mathbf{K}\,\mathbf{D}\,\mathbf{K}^{\intercal}\right\}\end{gathered} \tag{1}$$

where

$$\mathbf{x}_2=\boldsymbol{\Phi}\,\mathbf{x}+\boldsymbol{\Gamma}\,\mathbf{u} \tag{2}$$

$$\mathbf{P}_2=\boldsymbol{\Xi}+\boldsymbol{\Phi}\,\mathbf{P}\,\boldsymbol{\Phi}^{\intercal} \tag{3}$$

$$\mathbf{K}=\mathbf{P}\,\mathbf{A}^{\intercal}\,\mathbf{D}^{-1} \tag{4}$$

$$\mathbf{D}=\mathbf{Z}+\mathbf{A}\,\mathbf{P}\,\mathbf{A}^{\intercal} \tag{5}$$

and all quantities are matrices:

- $\mathbf{z}$ is a $b\times 1$ column vector containing one multidimensional observation
- $\mathbf{x}$ and $\mathbf{x}_2$ are $n\times 1$ column vectors of *model states*
- $\mathbf{Z}$ is a $b\times b$ matrix, the covariance of observation noise
- $\mathbf{P}$ and $\mathbf{P}_2$ are $n\times n$ matrices, the theoretical covariances of $\mathbf{x}$ and $\mathbf{x}_2$, respectively
- $\mathbf{A}$ 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

[3] B. Beckman, *Kalman Folding 3: Streams and Observable*, to appear.

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

[5] In most applications, the independent variable is physical time, however, it need not be. For convenience, we use the term *time* to mean *the independent variable of the problem* simply because it is shorter to write.

- $\boldsymbol{\Xi}$ is a non-dimensionalized $n \times n$ integral of *process noise* $\boldsymbol{\xi}$, namely $\int_0^{\delta t} \boldsymbol{\Phi}(\tau) \cdot \left(\begin{array}{c|c} 0 & 0 \\ \hline 0 & E\left[\boldsymbol{\xi}\boldsymbol{\xi}^T\right] \end{array} \right) \cdot \boldsymbol{\Phi}(\tau)^T \, d\tau$
- $\boldsymbol{\Phi}$ is a non-dimensionalized $n \times n$ propagator for $\mathbf{F}$, namely $e^{\mathbf{F}\,\delta t}$
- $\boldsymbol{\Gamma}$ is an $n \times \dim(\mathbf{u})$ integral of *system response*, namely $\int_0^{\delta t} \boldsymbol{\Phi}(\tau) \cdot \mathbf{G} \, d\tau$
- $\mathbf{u}$ is a vector of external *disturbances* or *control inputs*
- δt is an increment of time (or, more generally, the independent variable of the problem)

and the time-evolving states satisfy the following differential equation in *state-space form*:

$$\dot{\mathbf{x}} = \mathbf{F}\,\mathbf{x} + \mathbf{G}\,\mathbf{u} + \boldsymbol{\xi} \tag{6}$$

$\mathbf{F}$, $\mathbf{G}$, and $\mathbf{u}$ may depend on time, but not on $\mathbf{x}$; that is the meaning of "linear dynamic" in this context. In this paper, we relieve those restrictions by explicitly integrating non-linear equations of motion and by using Taylor-series approximations for the gain $\mathbf{K}$ and denominator $\mathbf{D}$ matrices.

2.1 Dimensional Arguments

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. Both kinds of dimensions are aspects of the *type* of a quantity. Dimensional arguments, like type-arguments more generally, are invaluable for checking equations.

If the physical and matrix dimensions of $\mathbf{x}$ are $[[\mathbf{x}]] \stackrel{\text{def}}{=} (\mathcal{X}, n \times 1)$, of $\mathbf{z}$ are $[[\mathbf{z}]] \stackrel{\text{def}}{=} (\mathcal{Z}, b \times 1)$, of δt are $[[\delta t]] \stackrel{\text{def}}{=} (\mathcal{T}, \text{scalar})$, and of $\mathbf{u}$ are $[[\mathbf{u}]] \stackrel{\text{def}}{=} (\mathcal{U}, \dim(\mathbf{u}) \times 1)$, then

$$\begin{array}{lcl} [[\mathbf{Z}]] & = & (\quad \mathcal{Z}^2 \quad\quad b \times b \quad) \\ [[\mathbf{A}]] & = & (\quad \mathcal{Z}/\mathcal{X} \quad\quad b \times n \quad) \\ [[\mathbf{P}]] & = & (\quad \mathcal{X}^2 \quad\quad n \times n \quad) \\ [[\mathbf{A}\,\mathbf{P}\,\mathbf{A}^T]] & = & (\quad \mathcal{Z}^2 \quad\quad b \times b \quad) \\ [[\mathbf{D}]] & = & (\quad \mathcal{Z}^2 \quad\quad b \times b \quad) \\ [[\mathbf{P}\,\mathbf{A}^T]] & = & (\quad \mathcal{X}\,\mathcal{Z} \quad\quad n \times b \quad) \\ [[\mathbf{K}]] & = & (\quad \mathcal{X}/\mathcal{Z} \quad\quad n \times b \quad) \\ [[\mathbf{F}]] & = & (\quad \text{powers of } 1/\mathcal{T} \quad\quad n \times n \quad) \\ [[\boldsymbol{\Phi}]] & = & (\quad \text{dimensionless} \quad\quad n \times n \quad) \\ [[\mathbf{G}]] & = & (\quad \mathcal{X}/(\mathcal{U}\mathcal{T}) \quad\quad n \times \dim(\mathbf{u}) \quad) \\ [[\boldsymbol{\Gamma}]] & = & (\quad \mathcal{X}/\mathcal{U} \quad\quad n \times \dim(\mathbf{u}) \quad) \\ [[\boldsymbol{\Xi}]] & = & (\quad \mathcal{X}^2 \quad\quad n \times n \quad) \end{array} \tag{7}$$

The non-dimensionalization of $\mathbf{F}$ and $\boldsymbol{\Xi}$ requires careful analysis and is the subject of another paper in this series. For now, assume they have been implicitly non-dimensionalized.
In all examples in this paper, the observations $\mathbf{z}$ are 1×1 matrices, equivalent to scalars, so $b = 1$, but the theory and code carry over to multi-dimensional vector observations.

3 Tracking with Drag

Let us reproduce Zarchan and Musoff's[6] example of tracking a falling object in one position dimension, with aerodynamic drag. Throughout this series of papers, we refer to the examples in that reference because they provide solid benchmarks against which to check our contribution. The difference between our approach and typical presentations of Kalman-type filters is functional decomposition: writing code in functional style affords the ability to deploy it verbatim, even and especially at the binary level, in both laboratory and field. This ability can make the difference in a successful application because seemingly insignificant changes, even instruction order, can make qualitative differences in filter behavior due to floating-point issues.

To accommodate nonlinearity, we replace equation 2 for time-propagation of the state $\mathbf{x}$ with explicit numerical integration of the nonlinear equations of motion. We use an internal fold over a lazy stream of differential updates to the state, a kind of fold introduced in part 4 of this series.[3] This form runs in constant memory and allows easy change of the integrator, say from Euler to Runge-Kutta.

3.1 Equations of Motion

To establish a benchmark solution, we solve the differential equations of motion using Wolfram's built-in numerical integrator. We then introduce our own Euler and Runge-Kutta integrators and show they produce similar results when folded over lazy streams. These integrators do not use special features of the Wolfram language, so they are easy to implement in other languages.

Let $h(t)$ be the height of the falling object, and let the state vector $\mathbf{x}(t)$ contain $h(t)$ and its first derivative, $\dot{h}(t)$, the speed of descent.

$$\mathbf{x} = \begin{bmatrix} h(t) \\ \dot{h}(t) \end{bmatrix}$$

Omitting, for clarity's sake, explicit dependence of h and $\dot{h}$ on time, the equations of motion are elementary:

$$\begin{bmatrix} \dot{h} \\ \ddot{h} \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} h \\ \dot{h} \end{bmatrix} + \begin{bmatrix} 0 \\ -1 - \text{sign}(\dot{h})\, \rho(h)\, \dot{h}^2/(2\beta) \end{bmatrix} \begin{bmatrix} g \end{bmatrix} \tag{8}$$

where

- g is the acceleration of Earth's gravitation, about 32.2 ft/s^2
- $\rho(h)$ is the density of air[7] in slug/ft^2; $\rho\,\dot{h}^2$ has units of pressure, that is, slug/(ft $\cdot$ sec^2)
- $\beta = 500$ slug/(ft $\cdot$ sec^2) is a constant *ballistic coefficient* of the object in units of pressure (it is possible to estimate this coefficient in the filter; here, we treat it as a known constant).

The positive direction is up and we are only concerned with negative velocities where the object is approaching the ground. We may provisionally replace the factor sign($\dot{h}$) with -1 and keep our eye out for improper positive speeds.

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

[7] Zarchan and Musoff, on page 228, report 0.0034 for the density of air in slug/ft^3 at the surface; we believe the correct value is about 0.00242 but continue with 0.0034 for comparison's sake.

In scalar form, the equations are

$$\ddot{h} = g\left(\frac{\rho(h)\,\dot{h}^2}{2\beta} - 1\right)$$

or

$$\ddot{h} = g\left(\frac{Ae^{h/k}\,\dot{h}^2}{2\beta} - 1\right) \tag{9}$$

with $k = 22{,}000$ [ft], the e-folding height of the atmosphere, and $A = 0.0034$ [slug/ft^3] for the density of air at $h = 0$.

3.2 Built-in Solver

We integrate these equations for thirty seconds with the initial conditions $h(0) = 200{,}000$ [ft], $\dot{h} = -6{,}000$ [ft/s^2] with Wolfram's built-in `NDSolve`, the numerical integrator for differential equations, as follows:

```
With[{g = 32.2, A = 0.0034, k = 22000., beta = 500.},
 With[{x0 = 200000., v0 = -6000., t0 = 0., t1 = 30.},
   NDSolve[{h''[t] == -g + A g (h'[t])^2 Exp[-h[t]/k]/(2. beta),
       h[0] == x0, h'[0] == v0}, h, {t, t0, t1}]]]
```

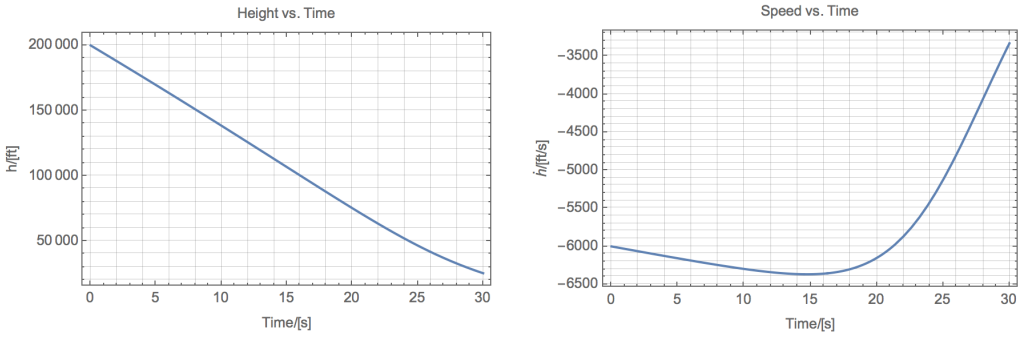

Figure 1: Trajectory of a falling object with drag

producing the results in figure 1. These results are indistinguishable from those in the reference.

3.3 Stream Solver

We can write the same differential equation as a lazy stream, which uses only constant memory. Thus, it is suitable for the internals of a Kalman filter. We implement the integrator as an accumulator function for a `foldStream` from paper 3[8] which produces all intermediate results as a new stream:

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

```
foldStream[f_, s_, Null[]] := (* acting on an empty stream *)
  {s, Null}; (* produces 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[]] &};
```

The simplest integrator is the Euler integrator, which updates a state with its derivative times a small interval of time:

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

This is a binary function that takes two compound arguments. The first is an instance of the accumulation type: a pair of a time `t` and a (usually compound) state `x`. The second is an element of the input stream, a triple of a time differential `dt`, the same time `t` that appears in the first argument, and a function `Dx` that computes the derivative of the state given the state and the time as `Dx[x,t]`.

Folding this integrator over the streamed differential equation produces a streamed solution. The input stream must produce elements of the form `{dt, t, Dx}` and, like all streams, contain a thunk that produces the rest of the stream.[9]

```
dragDStream[Delta : {dt_, t_, Dx_}] :=
  {Delta, dragDStream[{dt, t + dt, Dx}] &};
```

This bit contains nothing specific to our example, but just threads around the integration inputs and increments time. It could be much more rich, manipulating `dt` and `Dx` for speed or numerics (*adaptive integration*).

The kernel of our differential equation is the derivative function `Dx`, which, for our example, is

```
With[{g = 32.2, A = 0.0034, k = 22000., beta = 500.},
  dragD[{x_, v_}, t_] := {v, g (A Exp[-x/k] v^2/(2. beta) - 1)}];
```

Integrating the differential equation for thirty seconds looks like this:

```
(* constants and initial conditions *)
With[{x0 = 200000., v0 = -6000., t0 = 0., t1 = 30., dt = .1},
 takeUntil[
  foldStream[
   eulerAccumulator,
   {t0, {x0, v0}},
   dragDStream[{dt, t0, dragD}]
   ], First[#] > t1 &]] (* predicate on first elements of solution *)
```

The type of the result, here, is a lazy stream produced by `takeUntil` from the lazy stream produced by `foldStream`. Because these streams are lazy, nothing happens until we demand values for, say, plotting. The results are indistinguishable from those in figure 1.

[9] Wolfram's ampersand postfix operator can covert its operand into a thunk.

The arguments of `takeUntil` are a stream and a predicate. The result is a new stream that pulls values from the original stream, applying the predicate until it produces `True`. At that point, the rest of the stream returned by `takeUntil` is empty, represented by invocation of the null thunk, `Null[]`, The implementation of `takeUntil` is in three overloads:

Given an empty stream and any predicate, produce the empty stream:

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

Given a stream containing a value `v` and a tail `thunk`, return the empty stream if the predicate evaluates to `True`:

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

Otherwise, recurse by invoking the `thunk` in the stream:

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

3.4 What's the Point?

The point of this style of integration is that we can change three aspects of the integration independently of one another, leaving the others verbatim, without even recompilation, because we have un-nested and *decomplected*[10] these aspects:

1. the integrator
2. sophisticated adaptive treatments of the time increment `dt` and derivative function `Dx`
3. the internals of the derivative function `Dx`

For example, should Euler integration prove inadequate, we can easily substitute second- or fourth-order Runge-Kutta integrators. The only requirement is that an integrator must match the integrator's functional interface:

```
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.}]]]];
```

Decomplecting these bits also makes them easier to review and verify by hand because dependencies are lexically localized, making expressions smaller, easier to memorize and to find on a page.

[10] "Decomplecting" is a term coined by Rich Hickey for un-braiding and un-nesting bits of software.

3.5 Gain and Covariance Updates

For gains and covariances, we need the best linear approximation of the equations of motion so that we have an expression that structurally resembles equation 6. When there are no disturbances, $\mathbf{G}\,\mathbf{u} = \mathbf{0}$ and the solution of the linear equation $\dot{\mathbf{x}} = \mathbf{F}\,\mathbf{x}$ also satisfies $\Delta\dot{\mathbf{x}} = \mathbf{F}\,\Delta\mathbf{x}$ for small differences $\Delta\dot{\mathbf{x}}$ and $\Delta\mathbf{x}$. We seek a similar form for our nonlinear equations of motion because we can linearize them around small differences Δh and $\Delta\dot{h}$:

$$\begin{bmatrix}\Delta\dot{h}\\ \Delta\ddot{h}\end{bmatrix} = \begin{bmatrix}\frac{\partial\dot{h}}{\partial h} & \frac{\partial\dot{h}}{\partial\dot{h}}\\ \frac{\partial\ddot{h}}{\partial h} & \frac{\partial\ddot{h}}{\partial\dot{h}}\end{bmatrix}\begin{bmatrix}\Delta h\\ \Delta\dot{h}\end{bmatrix} = \mathbf{F}(\mathbf{x} = [h\;\dot{h}]^{\mathsf{T}})\cdot\begin{bmatrix}\Delta h\\ \Delta\dot{h}\end{bmatrix} \tag{10}$$

Thus, with $k = 22,000$ [ft], the e-folding height of the atmosphere, and $A = 0.0034$ [slug/ft^3] for the density of air[7] at $h = 0$, our linearized system-dynamics matrix is

$$\mathbf{F}(\mathbf{x}) = \begin{bmatrix}0 & 1\\ \frac{-Ag\dot{h}^2e^{h/k}}{2\beta k} & \frac{Ag\dot{h}e^{h/k}}{\beta}\end{bmatrix} \tag{11}$$

We need $\mathbf{\Phi} = e^{\mathbf{F}t}$ to propagate solutions forward, because, if $\dot{\mathbf{x}} = \mathbf{F}\,\mathbf{x}$, then $e^{\mathbf{F}t}\,\mathbf{x}(t)$ effects a Taylor series. To first order,

$$\begin{aligned}\mathbf{x}(t+\delta t) &= e^{\mathbf{F}\,\delta t}\,\mathbf{x}(t)\\ &\approx (\mathbf{1} + \mathbf{F}\,\delta t)\;\mathbf{x}(t)\\ &= \mathbf{x}(t) + \mathbf{F}\mathbf{x}(t)\,\delta t\\ &\approx \mathbf{x}(t) + \dot{\mathbf{x}}(t)\,\delta t\end{aligned} \tag{12}$$

First-order expansions turn out to be enough, so we take $\mathbf{\Phi}(\delta t) = \mathbf{1} + \mathbf{F}\,\delta t$ for our propagator matrix.

We compute the gains and covariances as in equations 3, 4, and 5:

$$\mathbf{P} \leftarrow \mathbf{\Xi} + \mathbf{\Phi}\,\mathbf{P}\,\mathbf{\Phi}^{\mathsf{T}} \tag{13}$$

where Ξ, integral of the process noise, is

$$(\sigma_\xi)^2\cdot\begin{bmatrix}\frac{\delta t^3}{3} & \mathbf{F}_{22}\delta t^3/3 + \frac{\delta t^2}{2}\\ \mathbf{F}_{22}\delta t^3/3 + \frac{\delta t^2}{2} & {\mathbf{F}_{22}}^2\delta t^3/3 + \mathbf{F}_{22}\delta t^2 + \delta t\end{bmatrix} \tag{14}$$

with matrix element $\mathbf{F}_{22}$ evaluated at the current state $\mathbf{x}$.

4 Concluding Remarks

It's easy to add system dynamics to a static Kalman filter. Expressed as the accumulator function for a fold, the filter is decoupled from the environment in which it runs. We can run exactly the same code, even and especially the same binary, over arrays in memory, lazy streams,

asynchronous observables, any data source that can support a *fold* operator. Such flexibility of deployment allows us to address the difficult issues of modeling, statistics, and numerics in friendly environments where we have large memories and powerful debugging tools, then to deploy with confidence in unfriendly, real-world environments where we have small memories, asynchronous, real-time data delivery, and seldom more than logging for forensics. Emacs 24.5.1 (Org mode 8.3.4)