Kalman Folding 5: Non-Linear Models and the EKF (REVIEWER'S DRAFT)

Extracting Models from Data, One Observation at a Time

Brian Beckman

<2016-05-03 Tue>

Contents

1 Abstract

We exhibit a foldable Extended Kalman Filter that internally integrates non-linear equations of motion with a nested fold of generic integrators over lazy streams in constant memory. Functional form allows us to switch integrators easily and to diagnose filter divergence accurately, achieving orders of magnitude better speed than the source example from the literature. As with all Kalman folds, we can move the vetted code verbatim, without even recompilation, from the lab to the field.

2 Background and Synopsis

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 linear models with time-evolving states.

Here, we extend that example to include aerodynamic drag, making the model nonlinear. We must change the Kalman filters themselves to handle such problems. The resulting filters are called *Extended Kalman Filters* or EKFs.

The particular EKF designed here features internal integration of the 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 Observables*,[3] where we used them to step a static Kalman filter over observations. Here, lazy streams afford two advantages:

1. constant-memory solutions required for EKFs in embedded systems in the field
2. easy switching of integrators, even at run time, allowing accurate diagnosis and tuning of the filter

We show these advantages at work by improving Zarchan and Musoff's[4] example of tracking the height of a falling object with drag. These authors exhibit a sequence of filter tunings to converge the filter, ending with a second-order Runge-Kutta stepped at 100 times the observation frequency. Integration dominates the run-time performance of their example. Because their integration code is enmeshed with the filter and with data delivery, they were not easily able to experiment with alternative integrators and missed the opportunity to try fourth-order Runge-Kutta, which we find converges the filter 100 times faster.

Other papers in this series feature applications of EKFs to a variety of problems including navigation and pursuit.

In this series of papers, we use the Wolfram language[5] 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.

[1] B. Beckman, *Kalman Folding, Part 1*, to appear.
[2] B. Beckman, *Kalman Folding 2: Tracking and System Dynamics*, to appear.
[3] B. Beckman, *Kalman Folding 4: Streams and Observables*, to appear.
[4] Zarchan and Musoff, *Fundamentals of Kalman Filtering, A Practical Approach, Fourth Edition*, Ch. 4
[5] `http://reference.wolfram.com/language/`

3 Linear Kalman Accumulator with Time Evolution

In *Kalman Folding 2: Tracking*,[2] we found the following accumulator function for a fold that implements the linear dynamic Kalman filter, that is, a filter that can track states that evolve with time[6] according to a linear transformation of the state:

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

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
- $\boldsymbol{\Xi}$ is an $n\times n$ integral of *process noise* $\boldsymbol{\xi}$, namely $\int_0^{\delta t}\boldsymbol{\Phi}(\tau)\cdot\left(\begin{array}{c|c}0 & 0\\ \hline 0 & \mathrm{E}\left[\boldsymbol{\xi}\boldsymbol{\xi}^\intercal\right]\end{array}\right)\cdot\boldsymbol{\Phi}(\tau)^\intercal\,\mathrm{d}\tau$
- $\boldsymbol{\Phi}$ is the non-dimensional $n\times n$ propagator for $\mathbf{F}$, namely $e^{\mathbf{F}\,\delta t}$
- $\boldsymbol{\Gamma}$ is an $n\times\dim(\mathbf{u})$ integral of *system response*, namely $\int_0^{\delta t}\boldsymbol{\Phi}(\tau)\cdot\mathbf{G}\,\mathrm{d}\tau$
- $\mathbf{u}$ 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.

[6] 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.

3.1 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 and code.

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

$$\begin{array}{lcccc}
[[\mathbf{Z}]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\
[[\mathbf{A}]] & = & (& \mathcal{Z}/\mathcal{X} & b \times n \ ) \\
[[\mathbf{P}]] & = & (& \mathcal{X}^2 & n \times n \ ) \\
[[\mathbf{A}\,\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\
[[\mathbf{D}]] & = & (& \mathcal{Z}^2 & b \times b \ ) \\
[[\mathbf{P}\,\mathbf{A}^\intercal]] & = & (& \mathcal{X}\mathcal{Z} & n \times b \ ) \\
[[\mathbf{K}]] & = & (& \mathcal{X}/\mathcal{Z} & n \times b \ ) \\
[[\mathbf{F}\,\mathbf{x}]] & = & (& \mathcal{X}/\mathcal{T} & n \times n \ ) \\
[[\mathbf{\Phi}\,\mathbf{x}]] & = & (& \mathcal{X} & n \times n \ ) \\
[[\mathbf{G}\,\mathbf{u}]] & = & (& \mathcal{X}/\mathcal{T} & n \times 1 \ ) \\
[[\mathbf{\Gamma}\,\mathbf{u}]] & = & (& \mathcal{X} & n \times 1 \ ) \\
[[\mathbf{\Xi}]] & = & (& \mathcal{X}^2 & n \times n \ )
\end{array} \tag{7}$$

The matrices $\mathbf{F}$, $\mathbf{\Phi}$, $\mathbf{G}$, and $\mathbf{\Gamma}$ do not have single dimensions on their own, but their dimensions in various combinations with other matrices make sense. Elements of matrix expressions for $\mathbf{\Xi}$ include sufficient implicit physical dimensions to make its overall physical dimensions work out to $\mathcal{X}^2$. Detailed dimensional analysis of these matrices is the subject of another paper in this series. In all examples in this paper, the observations $\boldsymbol{z}$ are 1×1 matrices, equivalent to scalars, so $b = 1$, but the theory and code carry over to multi-dimensional vector observations.

4 Tracking with Drag

To accommodate nonlinear equations of state evolution, we replace equation 2 with explicit numerical integration. The rest of the EKF uses equations 3, 4, and 5: with a propagator $\mathbf{\Phi}$ derived from first-order linearization.

4.1 Equations of Motion

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.

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

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

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

where

- g is the acceleration of Earth's gravitation, about 32.2 ft/s^2
- $\rho(h)$ is the density of air in slug/ft^3; $\rho\,\dot{h}^2$ 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.

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[7] at $h = 0$.

4.2 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 `foldStream` from paper 4 of this series,[3] which produces all intermediate results as a new stream.

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]};
```

Like all accumulator functions, this is a binary function that takes two arguments. The first is an instance of accumulation type, in this case, a pair of a scalar time `t` and a vector 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 values of the form `{dt, t, Dx}` and, like all streams, also contain a thunk that produces the rest of the stream.[8]

[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.

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

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

This `dragDStream` contains nothing specific to our example, but just increments time and passes through the integration inputs. 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 the following:

```
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)}];
```

in which `x` stands for h and `v` for $\dot{h}$. It is generalized to handle differential equations that have explicit dependence on a time variable `t`, but that parameter is harmlessly ignored in this example. 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 *)
```

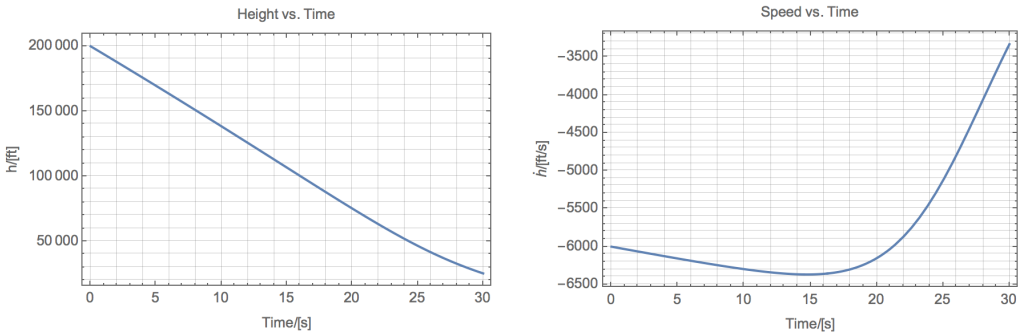

Figure 1: Trajectory of a falling object with drag

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, as in figure 1. These results are qualitatively indistinguishable from those in the reference and those produced by Wolfram's built-in numerical integrator `NDSolve`, giving us high confidence that we've got it right.

The arguments of `takeUntil` are a stream and a predicate, in our case, the literal function `First[#] > t1 &`. The result is a new stream that pulls values from the original stream, applying the predicate until it produces `True`.

The implementations of `foldStream`, `takeUntil` and other stream operators is the subject of another paper in this series.

4.3 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*[9] these aspects:

1. the integrator
2. potential manipulation of the time increment `dt` and derivative function `Dx`
3. the internals of the derivative function `Dx`

For example, should Euler integration prove inadequate, and it does, we can easily substitute second- or fourth-order Runge-Kutta integrators. This turns out to be crucial for a high-performance EKF in this example. The only requirement on an integrator is that it must match the function signature or type:

```
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 obvious, easier to memorize and to find on a page.

4.4 Gain and Covariance Updates

For gains and covariances, first-order linear approximations suffice. If we write the non-linear equations in state-space form as $\dot{\mathbf{x}} = \mathrm{f}(\mathbf{x})$, then a Taylor series, to first order, yields

$$\begin{aligned} \dot{\mathbf{x}} &= \mathrm{f}(\mathbf{x}_0) + \mathbf{F}(\mathbf{x}_0) \cdot (\mathbf{x} - \mathbf{x}_0) \\ \Leftrightarrow \dot{\mathbf{x}} &= \mathbf{F}(\mathbf{x}_0) \cdot \mathbf{x} + \dot{\mathbf{x}}_0 - \mathbf{F}(\mathbf{x}_0) \cdot \mathbf{x}_0 \end{aligned}$$

where $\mathbf{F}$ is the Jacobian matrix,

$$\mathbf{F}\left(\mathbf{x} = \begin{bmatrix} h \\ \dot{h} \end{bmatrix}\right) = \begin{bmatrix} \frac{\partial \dot{h}}{\partial h} & \frac{\partial \dot{h}}{\partial \dot{h}} \\ \frac{\partial \ddot{h}}{\partial h} & \frac{\partial \ddot{h}}{\partial \dot{h}} \end{bmatrix} \tag{10}$$

and clearly fills the role played by $\mathbf{F}$ in the linear state-space form, equation 6. Our linearized system-dynamics matrix is

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

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

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. Again, to first order,

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

We take $\mathbf{\Phi}(\delta t) = \mathbf{1} + \mathbf{F}\,\delta t$ for our propagator matrix and compute the gains and covariances as in equations 3, 4, and 5:

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

where Ξ, integral of the process noise, is

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

5 The EKF

Though the following code block is bigger than we have seen in earlier papers in this series, it is a straight implementation of the notions explained above, highly modularized. The block of code establishes one global symbol, `EKFDrag`, which we tune and analyze in the last section of this paper.

`With` establishes numerical constants for the equations of motion. `Module` establishes local variables to hold the differential-equation kernel and stream, and for the propagator matrix $\mathbf{\Phi}$ and process noise $\mathbf{\Xi}$.

```
With[{g = 32.2, A = 0.0034, k = 22000., beta = 500.},
 Module[{dragD, dragDStream, F21, F22, Phi, Xi},
```

The following lines furnish implementations of these functions:

```
  (* x stands for h, v for hdot *)
  dragD[{x_, v_}, t_] := {v, g (A Exp[-x/k] v^2/(2. beta) - 1)};
  dragDStream[Delta : {dt_, t_, Dx_}] :=
    {Delta, dragDStream[{dt, t + dt, Dx}] &};
  F21[x_, v_] := -A Exp[-x/k] g v^2/(2. k beta);
  F22[x_, v_] := A Exp[-x/k] g v/beta;
  Phi[dt_, {x_, v_}] := {{1, dt}, {dt*F21[x, v], 1 + dt*F22[x, v]}};
```

```
Xi[dt_, {x_, v_}] := With[{f = F22[x, v]},
  {{dt^3/3, (dt^2*(3 + 2*dt*f))/6}, {(dt^2*(3 + 2*dt*f))/6,
    dt + dt^2*f + (dt^3*f^2)/3}}];
```

The EKF itself is in the scope of these variables, and lambda lifts over

1. σ_ξ, the constant standard deviation of the process noise
2. $\mathbf{Z}$, the constant observation-noise matrix
3. the integrator function, for instance `eulerAccumulator` or either of the Runge-Kutta integrators
4. the filter period `fdt`
5. the internal integration period `idt`

allowing independent tuning of all these parameters. Its accumulation type is $\{\mathbf{x}, \mathbf{P}\}$, as usual. Its observation type includes time t because the integrators are all generalized to include it, even though, in our current example, the differential equations do not depend explicitly on the time variable. It could be optimized out. The other members of the observation packet are the usual partials matrix $\mathbf{A}$ and the observation itself $\mathbf{z}$. This is standard Kalman folding as explained in the first paper in this series.[10]

The stream operator `last` forces the lazy integration stream to execute, albeit in constant memory, until it picks up and returns the final value produced by `takeUntil`. This composition of `last`, `takeUntil`, and `foldStream` performs the EKF's high-precision replacement for the standard Kalman filter's update equation 2, operating at the integration period `idt`. The rest of the code implements equations 3, 4, and 5 with the linearized propagator `Phi` operating at the filter period `fdt`.

```
EKFDrag[sigmaXi_, Zeta_, 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]}];
```

6 Tuning and Performance

Because we can tune five parameters of the filter independently, we can efficiently explore the tuning space. The first task is to reproduce the author's results, then to look for opportunities to improve them.

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

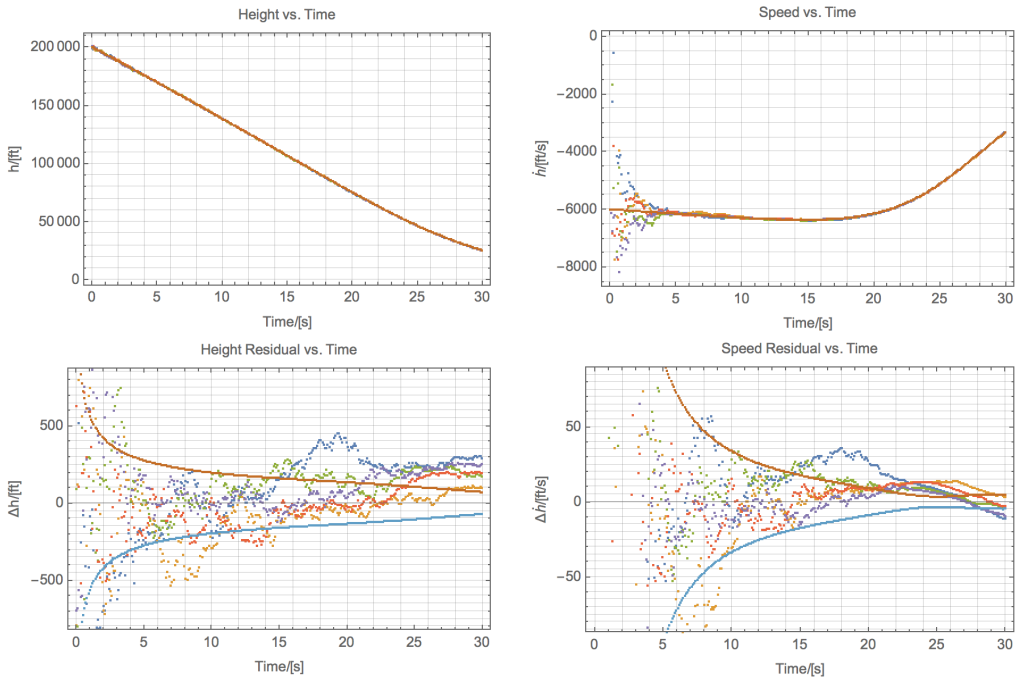

Figure 2: Euler integrator, `idt` $= 0.1$ sec, $\sigma_\zeta = 1000$ ft

Zarchan and Musoff report filter convergence and excellent speed for the Euler integrator operating at a period of 0.1 seconds, exactly the same as the filter period, and a standard deviation of $1,000$ ft for observation noise. We reproduce their results qualitatively in figure 2, by folding `EKFDrag` over a lazy stream of deterministically pseudorandom observations. The smoother lines represent one-sigma theoretical covariance envelopes and the noisy dots represent five iterations of the filter over faked data.

Figure 2 exhibits overall convergence, but there are signs of trouble for times beyond 20 sec. These are visually obvious, but would be difficult to detect statistically.

The authors report, and we concur, complete filter failure when the observation standard deviation is reduced to 25 feet, which forces the filter to rely much more on the integrator than on the observations at higher times because it has been told that the observations are reliable (low sigma). This interpretation is confirmed by the squeezing of the covariance envelopes in figure 3. The filter slavishly follows the integrator and seems to accumulate its floating-point errors into bad estimates. A detailed numerical analysis of this phenomenon is beyond the scope of this paper, but the authors gain evidence that this is the case, and we concur, by moving to a second-order Runge-Kutta integrator. They find, and we concur, that they must move to an integration period of 0.001 seconds, 100 times slower, to regain convergence. See figure 4.

We were able to restore the speed of the filter and produce results visually indistinguishable from figure 4 with the fourth-order Runge-Kutta integrator simply by feeding those parameters into `EKFDrag`. Now suitably tuned, the filter can be deployed verbatim, without even recompilation, in the field. We emphasize the importance of verbatim deployment, as in the first paper in

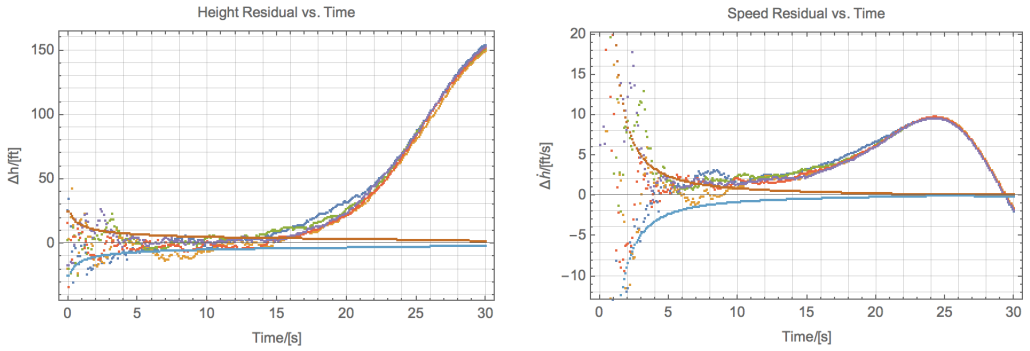

Figure 3: Euler integrator, `idt` $= 0.1$ sec, $\sigma_\zeta = 25$ ft

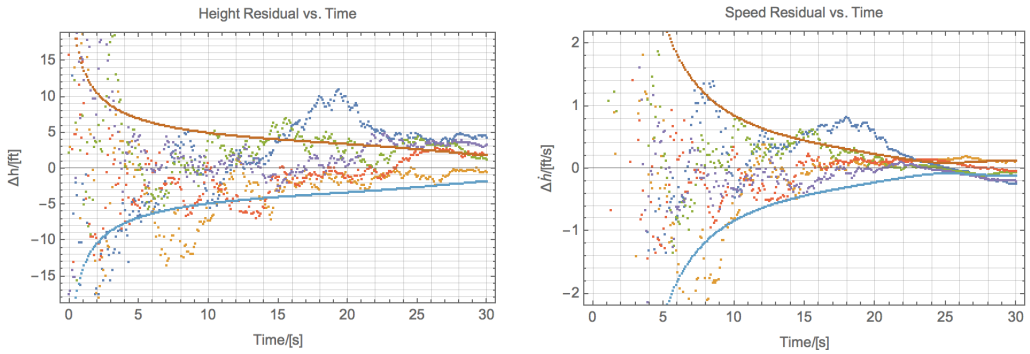

Figure 4: RK-2 integrator, `idt` $= 0.001$ sec, $\sigma_\zeta = 25$ ft; also RK-4 integrator, `idt` $= 0.1$ sec

this series, because floating-point issues are extremely sensitive to the slightest change. We have seen many cases where even changing the order of operations by compiler optimizer flags or by targeting different versions of the same processor family produce qualitatively different results due to non-associativity of floating point math and accumulation phenomena.

We note in passing that Zarchan and Musoff also find, and we concur, that increasing the order of the Taylor series for computing $\mathbf{\Phi}$ and Ξ does not qualitatively improve the filter. That option might become relevant and important at longer filter periods `fdt` or in other applications.

7 Concluding Remarks

The Extended Kalman Filter (EKF) typically handles non-linear problems by propagating states with high-precision integration and propagating Kalman gain and state covariance by linear approximations. The benefits of writing EKFs as folds over lazy streams include high modularity, allowing efficient and accurate tuning and diagnosis, and the flexibility to deploy fragile floating-point code verbatim, without even recompilation, from the lab to the field. Emacs 24.5.1 (Org mode 8.3.4)