Kalman Folding 2: Tracking and System Dynamics (Review 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. The examples in that paper are all static, meaning that the states of the model do not depend on the independent variable, often physical time.

Here, we present a dynamic Kalman filter in the same, functional form. This filter can handle many dynamic, time-evolving applications including some tracking and navigation problems, and is easilly extended to nonlinear and non-Gaussian forms, the Extended Kalman Filter (EKF) and Unscented Kalman Filter (UKF) respectively. Those are subjects of other papers in this Kalman-folding series. Here, we reproduce a tracking example from a well known reference, but in functional form, highlighting the advantages of that form.

[1]B. Beckman, *Kalman Folding Part 1*, `http://vixra.org/abs/1606.0328`.

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:

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

where

$$\begin{aligned}\mathbf{K}&=\mathbf{P}\,\mathbf{A}^{\intercal}\,\mathbf{D}^{-1} &(2)\\ \mathbf{D}&=\mathbf{Z}+\mathbf{A}\,\mathbf{P}\,\mathbf{A}^{\intercal} &(3)\end{aligned}$$

and all quantities are matrices:

- $\mathbf{z}$ 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

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 $[[\mathbf{x}]]\overset{\text{def}}{=}(\mathcal{X},n\times 1)$ and of $\mathbf{z}$ are $[[\mathbf{z}]]\overset{\text{def}}{=}(\mathcal{Z},b\times 1)$, then

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

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

In all examples in this paper, the observations z are 1×1 matrices, equivalent to scalars, so $b = 1$, but the theory and code carry over to multi-dimensional vector observations.

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. Lambda lifting is desirable when $\mathbf{Z}$ does not depend on the independent variable because it reduces coupling between the accumulator function and its calling environment. It is better to pass in an explicit constant than to implicitly close over[4] ambient constants, and it is good to keep the number of parameters in the observation packet $\{\mathbf{A}, z\}$ as small as possible. In other applications, $\mathbf{Z}$ can depend on the independent variable, in which case we pass it around in the observation packet along with $\mathbf{A}$ and z.

In Wolfram, this function is

```
kalman[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]}]
```

For details about this filter including walkthroughs of small test cases, see the first paper in the series, *Kalman Folding, Part 1*.[1] In another paper in this series, *Kalman Folding 3: Derivations*,[5] we present a full derivation of this static accumulator function.

3 A Tracking Example

Let us reproduce an example from Zarchan and Musoff,[6] to track the height of a falling object, with no aerodynamic drag. Handling drag requires an extended Kalman filter (EKF), subject part five of this series,[7] because a model with drag is nonlinear.

We will need a dynamic Kalman filter, which applies an additional, linear dynamic model to the states.

3.1 Time-Evolving States

Suppose the states $\mathbf{x}$ suffer time evolution by a linear transformation $\mathbf{F}$ and an additional *disturbance* or *control* input $\mathbf{u}$, linearly transformed by $\mathbf{G}$. These new quantities may be functions of time, but not of $\mathbf{x}$ lest the equations be non-linear. Write the time derivative of $\mathbf{x}$ as

$$\dot{\mathbf{x}}(t) = \mathbf{F}\,\mathbf{x}(t) + \mathbf{G}\,\mathbf{u}(t)$$

If the physical dimensions of $\mathbf{x}$ are $\mathcal{X}$ and the physical dimensions of t are $\mathcal{T}$, then the physical dimensions of $\mathbf{F}\,\mathbf{x}$ are $\mathcal{X}/\mathcal{T}$. The various elements of $\mathbf{F}$ have physical dimensions of various powers of $1/\mathcal{T}$, so $\mathbf{F}$ does not have a single physical dimension on its own.

We often leave off the explicit denotation of time dependence for improved readability:

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

[4]`https://en.wikipedia.org/wiki/Closure_(computer_programming)`

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

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

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

$$\dot{x} = \mathbf{F}x + \mathbf{G}\,u$$

Generalize by adding *random process* noise ξ to the state derivative:

$$\dot{x} = \mathbf{F}x + \mathbf{G}\,u + \xi \tag{5}$$

This is standard *state-space form*[8] for differential equations. Solving these equations is beyond the scope of this paper, but suffice it to say that we need certain time integrals of $\mathbf{F}$, $\mathbf{G}$, and ξ as inputs to the filter. We denote these integrals as $\mathbf{\Phi}$, $\mathbf{\Gamma}$, and Ξ. The first, $\mathbf{\Phi}$, is defined as follows:

$$\mathbf{\Phi}(\delta t) \stackrel{\text{def}}{=} e^{\mathbf{F}\,\delta t} = \mathbf{1} + \frac{\mathbf{F}^2\delta t^2}{2!} + \frac{\mathbf{F}^3\delta t^3}{3!} + \cdots \tag{6}$$

where δt is an increment of time used to advance the filter discretely.

Like $\mathbf{F}$, $\mathbf{\Phi}$ does not have a single physical dimension. Only applications of $\mathbf{\Phi}$ to quantities including physical dimension $\mathcal{X}$ make sense. For instance, in the application $\mathbf{\Phi}\,x$, $\mathbf{\Phi}$ is dimensionless.

The second integral, $\mathbf{\Gamma}$, is defined as follows:

$$\mathbf{\Gamma}(\delta t) \stackrel{\text{def}}{=} \int_0^{\delta t} \mathbf{\Phi}(\tau)\cdot\mathbf{G}\,d\tau \tag{7}$$

The physical dimensions of $\mathbf{\Gamma}$ are defined only in combination with u: the product $\mathbf{\Gamma}\cdot u$ has physical dimensions $\mathcal{X}$.

The last integral, Ξ, is defined as follows:

$$\Xi(\delta t) \stackrel{\text{def}}{=} \int_0^{\delta t} \mathbf{\Phi}(\tau)\cdot\begin{pmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & E\left[\xi\xi^{\mathsf{T}}\right]\end{pmatrix}\cdot\mathbf{\Phi}(\tau)^{\mathsf{T}}\,d\tau \tag{8}$$

The physical dimensions of Ξ must be $\mathcal{X}^2$, and we accomplish this by accompanying the various zeros in the matrix in the integral with implicit dimensions so that the overall dimensions work out properly. Making these dimensions explicit would needlessly clutter the expressions.

Detailed dimensional analysis of these matrices is the subject of another paper in this series.

3.2 Recurrences for Dynamics

The transitions of a state (and its covariance) from time t to the next state (and covariance) at time $t + \delta t$ follow these recurrences:

$$x \leftarrow \mathbf{\Phi}\,x + \mathbf{\Gamma}\,u \tag{9}$$
$$\mathbf{P} \leftarrow \Xi + \mathbf{\Phi}\,\mathbf{P}\,\mathbf{\Phi}^{\mathsf{T}} \tag{10}$$

These equations appear plausible on inspection, and equation 9 has a particularly intuitive explanation. If $\mathbf{F}$ does not depend on time and if $\mathbf{G}\,u$ is zero, then the state space form $\dot{x} = \mathbf{F}x$ has

[8]https://en.wikipedia.org/wiki/State-space_representation

a trivial solution: $\mathbf{x}(t) = e^{\mathbf{F}\,t}\mathbf{x}_0 = \mathbf{\Phi}(t)\,\mathbf{x}_0$. We can use Φ to propagate the solution at any time $\mathbf{x}(t_1)$ forward to another time $\mathbf{x}(t_2)$ as follows:

$$\begin{aligned} \mathbf{x}(t_2) &= \mathbf{\Phi}(t_2 - t_1)\,\mathbf{x}(t_1) \\ &= e^{\mathbf{F}\times(t_2-t_1)}\,e^{\mathbf{F}\,t_1}\mathbf{x}_0 = e^{\mathbf{F}\,t_2}\mathbf{x}_0 \end{aligned} \tag{11}$$

This is the first step in verifying that the recurrences satisfy equation 5. It also explains why we call $\mathbf{\Phi}$ the *propagator matrix*.

3.3 The Foldable Filter

These tiny changes are all that is needed to add linear state evolution to the Kalman filter:

```
kalman[Zeta_][{x_, P_}, {Xi_, Phi_, Gamma_, u_, A_, z_}] :=
 Module[{x2, P2, D, K},
  x2 = Phi.x + Gamma.u;
  P2 = Xi + Phi.P.Transpose[Phi];
  (* after this, it's identical to the static filter *)
  D = Zeta + A.P2.Transpose[A];
  K = P2.Transpose[A].inv[D];
  {x2 + K.(z - A.x2), P2 - K.D.Transpose[K]}]
```

3.4 Dynamics of a Falling Object

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.[9]

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

The system dynamics are elementary:

$$\begin{bmatrix} \dot{h}(t) \\ \ddot{h}(t) \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} h(t) \\ \dot{h}(t) \end{bmatrix} + \begin{bmatrix} 0 \\ 1 \end{bmatrix} \begin{bmatrix} g \end{bmatrix}$$

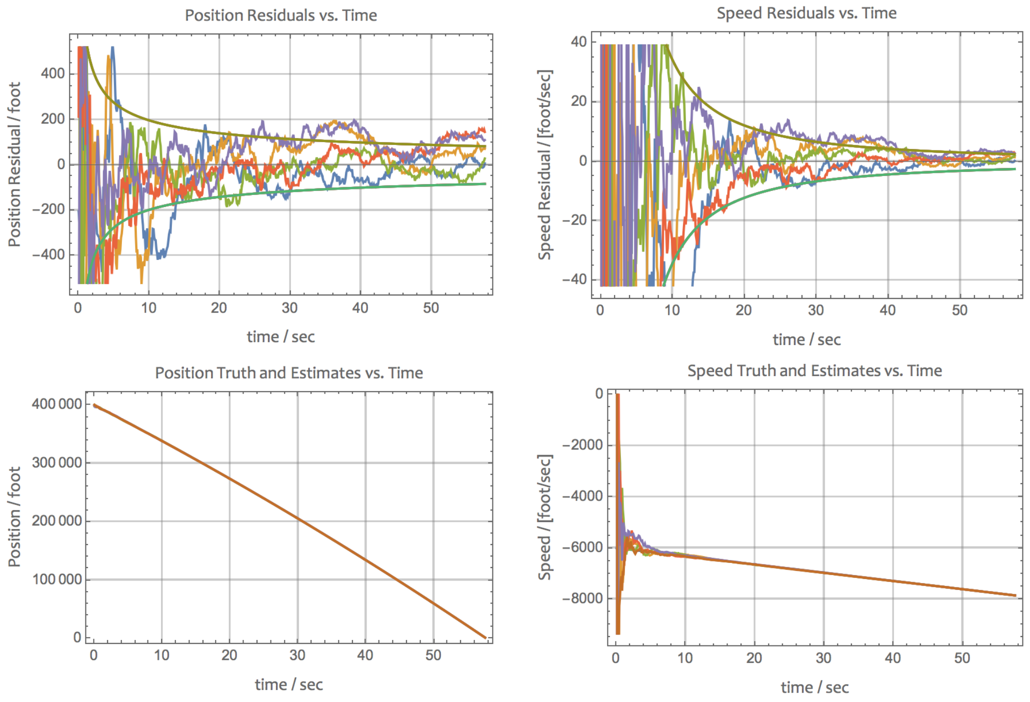
where g is the acceleration of Earth's gravitation, about -32.2ft/s^2 (note the minus sign). We read out the dynamics matrices:

$$\mathbf{F} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, \quad \mathbf{u} = \begin{bmatrix} g \end{bmatrix}$$

and their integrals from equations 6, 7, and 8

$$\mathbf{\Phi} = \begin{bmatrix} 1 & \delta t \\ 0 & 1 \end{bmatrix}, \quad \mathbf{\Gamma} = \begin{bmatrix} \delta t^2/2 \\ \delta t \end{bmatrix}, \quad \mathbf{\Xi} = E\left[\xi\xi^{\intercal}\right] \begin{bmatrix} \delta t^3/3 & \delta t^2/2 \\ \delta t^2/2 & \delta t \end{bmatrix}$$

[9]A state-space form containing a position and derivative is commonplace in second-order dynamics like Newton's Second Law. We usually employ state-space form to reduce n-th-order differential equations to first-order differential equations by stacking the dependent variable on $n-1$ of its derivatives in the state vector.



% position resids < σ (E >= 68)	79.3056
position resids >= σ	596.
position resids < σ	2284.
average position resid	-5.3446
theoretical std dev	83.2249
std dev of position resids	174.874

% speed resids < σ (E >= 68)	79.3056
speed resids >= σ	596.
speed resids < σ	2284.
average speed resid	-90.6453
theoretical std dev	2.50586
std dev of speed resids	1993.8

Figure 1: Simulated tracking of a falling object

We test this filter over a sequence of fake observations tracking an object from an initial height of 400, 000 ft and initial speed of $-6,000$ ft/s and from time $t = 0$s to $t = 57.5$ sec, just before impact at $h = 0$ ft. We take one observation every tenth of a second, so $\delta t = 0.10$ s. We compare the two states $h(t)$ and $\dot{h}(t)$ with ground truth and their residuals with the theoretical sum of squared residuals in the matrix $\mathbf{P}$. The results are shown in figure 1, showing good statistics over five consecutive runs and qualitatively matching the results in the reference.

The ground truth is

$$h(t) = h_0 + \dot{h}_0\, t + g\, t^2/2$$

where

$$h_0 = 400,000\,\text{ft}, \quad \dot{h}_0 = -6,000\,\text{ft/sec}$$

and we generate fake noisy observations by sampling a Gaussian distribution of zero mean and standard deviation 1, 000 ft. We do not need process noise for this example. It's often added during debugging of a Kalman filter to compensate for underfitting or overfitting an inappropriate model. It's also appropriate when we know that the process is stochastic or noisy and we have an estimate of its covariance.

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.