UltimateKalman: Flexible Kalman Filtering and Smoothing Using Orthogonal Transformations

Sivan Toledo
Tel Aviv University

Abstract

UltimateKalman is a flexible linear Kalman filter and smoother implemented in three popular programming languages: MATLAB, C, and Java. **UltimateKalman** is a slight simplification and slight generalization of an elegant Kalman filter and smoother that was proposed in 1977 by Paige and Saunders. Their algorithm appears to be numerically superior and more flexible than other Kalman filters and smoothers, but curiously has never been implemented or used before. **UltimateKalman** is flexible: it can easily handle time-dependent problems, problems with state vectors whose dimensions vary from step to step, problems with varying number of observations in different steps (or no observations at all in some steps), and problems in which the expectation of the initial state is unknown. The programming interface of **UltimateKalman** is broken into simple building blocks that can be used to construct filters, single or multi-step predictors, multi-step or whole-track smoothers, and combinations. The paper describes the algorithm and its implementation as well as with a test suite of examples and tests.

Keywords: Kalman filter, Kalman smoother, QR factorization.

1. Introduction

The invention of the Kalman filter by Rudulph E. Kálmán in 1960 (Kalman 1960) is considered one of the major inventions of the 20th century. The filter efficiently and incrementally tracks the hidden state of a linear discrete dynamic system; each state estimate uses all the observations of the system up to that point in time. The filter can also predict future states and with suitable adaptations, to handle non-linear dynamic systems and to smooth entire state trajectories. The literature on Kalman filters and their applications is vast and the importance of Kalman filters is beyond doubt. We mention a few relatively recent and relatively comprehensive sources (Brown and Hwang 1997; Grewal and Andrews 2015; Humpherys, Redd, and West 2012), but there are numerous other authoritative sources on Kalman filtering.

Twelve years later, Duncan and Horn discovered that mathematically, the Kalman filter computes the solution to a generalized linear least squares problem (Duncan and Horn 1972). Algorithmically and numerically, however, the Kalman filter algorithm is far from state-of-the art algorithms for linear least squares problems, including key algorithms that were invented and published in the 1950s (Givens 1954; Householder 1958).

Numerically, the most stable algorithms for least squares minimization are based on orthogonal transformations, and more specifically on the QR factorization, the singular-value decomposition, or their variants (Björck 1996; Golub and Loan 2013; Higham 2002). Kalman filter algorithms are not, and many of them, including Kálmán's original algorithm, are based on algebraic building blocks, like explicit matrix inversion, that are prone to instability.

The situation was rectified in 1977, when Paige and Saunders discovered and published an elegant Kalman filter algorithm based on orthogonal transformations (Paige and Saunders 1977). Their algorithm is a specialized QR factorization and it can easily implement filtering, prediction, and smoothing.

Strangely, the Paige-Saunders algorithm appears to have had very limited impact, even though it was about as efficient as other Kalman filters and smoothers and more numerically stable. Their paper was not cited much, nobody approached the authors to discuss it (Saunders 2018), and to the best of our knowledge, it was never implemented (Paige and Saunders' paper describes the algorithm and analyzes it, but does not mention an implementation).

The present paper and the software that it describes, called **UltimateKalman**, aim to make the algorithm widely available and to highlight its advantages over other Kalman filtering and smoothing algorithms. Indeed, the Paige and Saunders algorithm is not only more stable numerically than other Kalman algorithms, but it is also more flexible in two important senses. First, unlike other Kalman filters, it does not need to know the expectation of the initial state of the system. Second, it can be easily generalized to handle quantities that are added or dropped from the state vector. These features make modeling easier, as we demonstrate with concrete examples in Sections 5.4 and 5.5. The algorithm can also easily handle problems with a varying number of observations and with missing observations, and it is equally good as a filter and smoother. Many other Kalman filter algorithms lack these two characteristics, but some do posses them, so they are not completely unique.

UltimateKalman is not an completely identical to the Paige-Saunders algorithm, but rather a variant that is simpler and more general at the same time. We explain later in the paper the differences from the original algorithm, but at the same time acknowledge that all the fundamental algorithmic ideas in **UltimateKalman** come from the Paige-Saunders algorithm and paper.

Our implementation is split into a collection of easy-to-understand building blocks from which a user can compose a variety of Kalman-based computations, including filters, predictors, and smoothers, and combinations of these.

The implementation is available[1] in three popular programming languages: MATLAB, Java, and C.

[1] https://github.com/sivantoledo/ultimate-kalman

The rest of the paper is organized as follows. Section 2 provides background material on discrete linear dynamic systems and Kalman filters and smoothers. Section 3 describes the details of **UltimateKalman**. Section 4 presents our implementations of the algorithm, and Section 5 describes a suite of examples and tests that come with **UltimateKalman**, demonstrate its correctness, and show how to use it in various cases, some nontrivial. We discuss the algorithm and the software and our conclusions from this project in Section 6.

This paper does not directly compare **UltimateKalman** to other Kalman filtering algorithms and does not prove its correctness and its numerical properties; the paper by Paige and Saunders addresses these issues thoroughly, and the analyses there are equally applicable to **UltimateKalman** (Paige and Saunders 1977).

2. Background

The discrete Kalman filter is a method to efficiently estimate the state of a discrete linear dynamic system from indirect observations. **UltimateKalman** can handle more general cases than Kalman filters, so we describe here the more general version.

2.1. Discrete linear dynamic systems

The instantaneous state of a discrete dynamic system at time t_i is represented by an n_i-dimensional *state vector* $u_i \in \mathbb{R}^{n_i}$. We assume that u_i satisfies a recurrence that we refer to as an *evolution equation* and possibly another equation that we refer to as an *observation equation*. Note that we do not require all the states to have the same dimension, although the uniform-dimension case is very common. The evolution equation has the form

$$H_i u_i = F_i u_{i-1} + c_i + \epsilon_i \ , \tag{1}$$

where $H_i \in \mathbb{R}^{\ell_i \times n_i}$ and $F_i \in \mathbb{R}^{\ell_i \times n_{i-1}}$ are known full-rank matrices, $c_i \in \mathbb{R}^{\ell_{i-1}}$ is a known vector, called a *control vector*, that represents external forces acting on the system, and ϵ_i is an unknown noise or error vector. The control vector is often assumed to be the product of a known matrix and a known vector, but this is irrelevant for the Kalman filter. The noise or error vector ϵ_i admits state vectors that do not satisfy the equation $H_i u_i = F_i u_{i-1} + c_i$ exactly. The matrix H_i is often assumed to be the identity matrix, but we do not require this (and do not require it to be square). When H_i and F_i are square and full rank, u_i is a function of u_{i-1} and c_i up to the error term. Obviously, the first state u_0 that we model is not defined by an evolution recurrence.

Some of the state vectors u_i (but perhaps not all) also satisfy an *observation equation* of the form

$$o_i = G_i u_i + \delta_i \ , \tag{2}$$

where $G_i \in \mathbb{R}^{m_i \times n_i}$ is a known full-rank matrix, $o_i \in \mathbb{R}^{m_i}$ is a known vector of observations (measurements), and δ_i represents unknown measurement errors or noise. The dimension m_i of the observation of u_i can vary; it can be smaller than n_i (including zero, meaning that there are not observations of u_i), equal to n_i, or greater than n_i.

We can write all of the evolution and observation equations up to step k as a single large block-matrix equation,

$$\begin{bmatrix} o_0 \\ c_1 \\ o_1 \\ c_2 \\ \vdots \\ \vdots \\ c_k \\ o_k \end{bmatrix} = \begin{bmatrix} G_0 & & & & & \\ -F_1 & H_1 & & & & \\ & G_1 & & & & \\ & -F_2 & H_2 & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -F_k & H_k \\ & & & & & G_k \end{bmatrix} \begin{bmatrix} u_0 \\ u_1 \\ u_2 \\ \vdots \\ u_{k-1} \\ u_k \end{bmatrix} + \begin{bmatrix} \delta_0 \\ \epsilon_1 \\ \delta_1 \\ \epsilon_2 \\ \vdots \\ \vdots \\ \epsilon_k \\ \delta_k \end{bmatrix} . \tag{3}$$

We denote this system by $b = Au + e$. The matrix A and the vector b are known. The noise or error terms e are not known, but we assume that they are small. Our task is to estimate u from A and b.

If e is random, has zero expectation $\mathsf{E}(e) = 0$,[2] and has a known covariance matrix $\mathsf{COV}(e) = \mathsf{E}(ee^T)$ with $\mathsf{COV}(e)^{-1} = U^T U$, then the solution $\hat{u}$ of the *genearlized least squares* problem

$$\begin{aligned} \hat{u} &= \arg\min_u \|U(Au - b)\|_2^2 \\ &= \left(A^T U^T U A\right)^{-1} A^T U^T U b \end{aligned} \tag{4}$$

is the so-called best linear unbiased estimator (BLUE) of u (Aitken 1936). If we add the assumption that e has a Gaussian (normal) distribution, then the same minimizer is also the maximum-likelihood estimator of u.

If the structure of the G_is, F_is, and H_is guarantees that the rank of A always equals the number of columns, the system is called *observable*. This guarantees that (4) has a unique solution. This paper uses this term also in a more concrete sense: we say that a matrix or a block of a matrix is *observable* if its rank equal its column dimension. We also use the terms *flat* to describe a rectangular matrix or block with more columns than rows (a flat matrix cannot be observable) and *tall* to describe a rectangular matrix or block with more rows than columns.

Because our goal is to estimate the states of a *dynamic* system, we also denote the linear system (3) by $b^{(k)} = A^{(k)} u^{(k)} + e^{(k)}$, to provide a notation for the coefficient matrix A and for the right-hand b at a particular step k. Similarly, we denote $\mathsf{COV}(e^{(k)})^{-1} = (U^{(k)})^T U^{(k)}$.

[2] In this paper, capital letters and small letters denote unrelated objects. Here, E denotes the expectation and e denotes an error or noise vector.

2.2. Kalman filters and smoothers

Kalman filters and smoothers are a large family of efficient algorithms for solving problem (4) for some of the $\hat{u}_k$s when $\mathsf{COV}(e)$ is block diagonal with known blocks,

$$\mathsf{COV}(e) = \begin{bmatrix} C_0 & & & & & \\ & K_1 & & & & \\ & & C_1 & & & \\ & & & \ddots & & \\ & & & & K_k & \\ & & & & & C_k \end{bmatrix} .$$

That is, we assume that the matrices

$$\begin{aligned} K_i &= \mathsf{COV}\left(\epsilon_i\right) = \mathsf{E}\left(\epsilon_i \epsilon_i^T\right) \\ C_i &= \mathsf{COV}\left(\delta_i\right) = \mathsf{E}\left(\delta_i \delta_i^T\right) \end{aligned}$$

are known and that the offdiagonal blocks of $\mathsf{COV}(e)$ are all zero,

$$\begin{aligned} \mathsf{E}\left(\epsilon_i \delta_j^T\right) &= 0 \text{ for } i \text{ and } j \\ \mathsf{E}\left(\epsilon_i \epsilon_j^T\right) &= 0 \text{ for } i \neq j \\ \mathsf{E}\left(\delta_i \delta_j^T\right) &= 0 \text{ for } i \neq j\ . \end{aligned}$$

Some Kalman filtering and smoothing algorithms make additional assumptions, but **UltimateKalman** requires none. (For example, many Kalman filters assume that $G_0 = I$, which is equivalent to assuming that the expectation of u_0 is known.)

The generalized least-squares solution of Equation (3) estimates all the state vectors $u_{1,\ldots}, u_k$ using all the observations up to and including step k. We denote the vectors that make up of this solution by

$$\begin{bmatrix} \hat{u}_{0|k} \\ \hat{u}_{1|k} \\ \vdots \\ \hat{u}_{k|k} \end{bmatrix} . \tag{5}$$

The vector $\hat{u}_{k|k}$ is called the *filtered* estimate of step k. This estimate uses all the available observations of present and past states, but not of any future state. Vectors $\hat{u}_{0|k}, \ldots, \hat{u}_{k-1|k}$ are step-k *smoothed* estimates; they use observations of past, present, and future states. We can hope to compute filtered estimates almost in real time, whereas smoothed estimates can only be computed after a time lag. If we extend the system (3) with one or more block rows and columns that represent only evolution equations, the new vector components of the solutions are *predicted* estimates. For example, if we add a block row and a block column that contain $-F_{k+1}$, H_{k+1}, and c_{k+1}, but not G_{k+1} and not o_{k+1}, the last vector in the solution, denoted $\hat{u}_{k+1|k}$ is a prediction of u_{k+1} from the information we have up to step k. We can obviously continue to predict into the future by adding more evolution equations.

Kalman filters are efficient incremental algorithms that produce filtered and predicted estimates. Given H_k, F_k, G_k, c_k, o_k, C_k, K_k, and the compact data structure that was used to estimate $\hat{u}_{k-1|k-1}$, Kalman filters quickly compute $\hat{u}_{k|k}$ and updates the data structure (Brown and Hwang 1997; Grewal and Andrews 2015; Humpherys *et al.* 2012; Kalman 1960). The data structure itself is of size $\Theta((n_{k-1}+n_k)^2)$ and the number of operations required is $O((n_{k-1}+m_k+n_k)^3)$.

3. UltimateKalman and its heritage

This section describe the **UltimateKalman** algorithm. The algorithm is a slight simplification of the algorithm of Paige and Saunders (Paige and Saunders 1977) in that **UltimateKalman** uses block orthogonal transformations whereas the algorithm of Paige and Saunders uses Givens rotations. At the same time, the algorithm is also a generalization of the algorithm of Paige and Saunders, in that we allow the user to specify H_i and in that we allow the dimension of the state vector to change from step to step.

3.1. A specialized QR factorization

UltimateKalman computes estimates of the state vectors using a thin QR factorization of the weighted coefficient matrix

$$U^{(k)}A^{(k)} = \begin{bmatrix} W_0G_0 & & & & & \\ -V_1F_1 & V_1H_1 & & & & \\ & W_1G_1 & & & & \\ & -V_2F_2 & V_2H_2 & & & \\ & & \ddots & \ddots & & \\ & & & \ddots & \ddots & \\ & & & & -V_kF_k & V_kH_k \\ & & & & & W_kG_k \end{bmatrix} , \qquad (6)$$

where $W_i^TW_i = C_i^{-1}$ and $V_i^TV_i = K_i^{-1}$. The factorization is computed using a series of orthonormal transformations that are applied to block rows to reduce $U^{(k)}A^{(k)}$ to a block upper triangular form

$$R^{(k)} = \left(Q^{(k)}\right)^T \left(U^{(k)}A^{(k)}\right) ,$$

where

$$R^{(k)} = \begin{bmatrix} R_{0,0} & R_{0,1} & & & & \\ & R_{1,1} & R_{1,2} & & & \\ & & R_{2,2} & R_{2,3} & & \\ & & & \ddots & \ddots & \\ & & & & R_{k-1,k-1} & R_{k-1,k} \\ & & & & & \tilde{R}_{k,k} \end{bmatrix} . \tag{7}$$

The diagonal blocks $R_{i,i}$ are normally square and upper triangular, but are also allowed to be rectangular with more columns than rows. The superdiagonal blocks $R_{i-1,i}$ are are not identically zero. The same series of transformations is applied to the weighted right-hand side vector $U^{(k)}b^{(k)}$

$$U^{(k)}b^{(k)} = \begin{bmatrix} W_0 o_0 \\ V_1 c_1 \\ W_1 o_1 \\ V_2 c_2 \\ \vdots \\ \vdots \\ V_k c_k \\ W_k o_k \end{bmatrix} . \tag{8}$$

We denote the transformed right-hand by $y^{(k)} = \left(Q^{(k)}\right)^T \left(U^{(k)}b^{(k)}\right)$.

The transformations are discarded immediately after they are applied to the matrix and vector; no representation of $Q^{(k)}$ is stored.

3.2. Observability

Many Kalman algorithms rely on the assumption that A is observable, which guarantees that all the diagonal blocks $R_{i,i}$, as well as $\tilde{R}_{k,k}$, are square and upper triangular. For example, assuming that G_0 is square or tall (or is the identity) and that the F_is and H_i are square, along with the standard assumption that all of them are full rank, guarantees that A is observable. When all the diagonal blocks are square and triangular, we can compute the estimates using back substitution, starting from $\hat{u}_{k|k}$ and ending with $\hat{u}_{0|k}$.

However, **UltimateKalman** works and can provide useful estimates even when A is not always observable, and even when it is never observable. We explain the different cases in terms of the structure of the R factor and their meaning to the user.

If $\tilde{R}_{k,k}$ is flat but all the other $R_{i,i}$ blocks are square and triangular, the system is *not yet* observable, but it may become observable in a future state. We currently do not have enough observations to estimate any of the states. As the system evolves and additional observations are made, the system may become observable, allowing us to estimate the states that are currently unobservable.

If for some $i < k$ the diagonal block $R_{i,i}$ is flat, then states $u_0, \ldots, u_i$ are not observable and will never become observable. For any assignment of states u_{i+1} and up, there is a nontrivial space of equally good (in the sense of (4)) estimates for $u_0, \ldots, u_i$. **UltimateKalman** tolerates this situation because even in this case, the observations of $u_0, \ldots, u_i$ do provide useful information on future states.

3.3. The Paige-Saunders factorization algorithm

UltimateKalman uses the technique of the Paige-Saunders algorithm to produce $R^{(k)}$ incrementally.

Step k starts by adding to $R^{(k-1)}$ a block row and a block column that express an evolution equation,

$$\begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & \tilde{R}_{k-1,k-1} & \\ & & -V_k F_k & V_k H_k \end{bmatrix} .$$

We now examine the block

$$\begin{bmatrix} \tilde{R}_{k-1,k-1} \\ -V_k F_k \end{bmatrix} . \tag{9}$$

If this block is flat (or more generally, if its rank is smaller than n_k, which implies that it can be orthogonally reduced to a flat block), the algorithm leaves the bottom 2-by-2 blocks as is, denoting

$$\begin{bmatrix} R_{k-1,k-1} & R_{k-1,k} \end{bmatrix} = \begin{bmatrix} \tilde{R}_{k-1,k-1} & \\ -V_k F_k & V_k H_k \end{bmatrix} .$$

Otherwise, the algorithm computes a QR factorization of the block (9), uses the resulting R factor as $R_{k-1,k-1}$, and applies the orthonormal transformation to the last block column and to the right-hand side y. This transforms the R factor into

$$\begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & R_{k-1,k-1} & R_{k-1,k} \\ & & & \bar{R}_{k,k} \end{bmatrix} .$$

Block row $k-1$ is now *sealed*; it will not change any more. If $\tilde{R}_{k-1,k-1}$ was square, then so is $R_{k-1,k-1}$. If $\tilde{R}_{k-1,k-1}$ was flat, then $R_{k-1,k-1}$ might be either square or flat, depending on $V_k F_k$.

The bottom right block $\bar{R}_{k,k}$ is not upper triangular and it might be completely missing, if (9) is square or flat. If $\bar{R}_{k,k}$ is square or tall and we now need to predict $\hat{u}_{k|k-1}$ (and perhaps additional future states), we compute the QR factorization of $\bar{R}_{k,k}$ and apply the transformation to y. We denote the R factor of $\bar{R}_{k,k}$ by $\check{R}_{k,k}$.

If there are no observations of u_k, then $\tilde{R}_{k,k} = \check{R}_{k,k}$ (or $\tilde{R}_{k,k} = \bar{R}_{k,k}$ if $\bar{R}_{k,k}$ is flat or missing) and we are done with step k.

If there are observations in step k, we add another block row to the R factor,

$$\begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & R_{k-1,k-1} & R_{k-1,k} \\ & & & \bar{R}_{k,k} \\ & & & W_k G_k \end{bmatrix} .$$

If we computed $\check{R}_{k,k}$, in principal we can use it instead of $\bar{R}_{k,k}$, but there is usually no significant benefit to this. If the block

$$\begin{bmatrix} \bar{R}_{k,k} \\ W_k G_k \end{bmatrix}$$

is flat, it becomes $\tilde{R}_{k,k}$ and we are done. Otherwise we compute the QR factorization of this block, use the R factor as $\tilde{R}_{k,k}$, and apply the transformation to y.

We now have $R^{(k)}$ and the $y^{(k)}$. Note that we have not used in this step block rows $1, \ldots, k-2$ of $R^{(k-1)}$ and $y^{(k-1)}$.

3.4. Forgetting and rolling back

Given $R^{(k)}$ and the $y^{(k)}$, the estimates $\hat{u}_{i|k}$ are computed by back substitution, from the bottom up.

Therefore, if we need only filtered estimates $\hat{u}_{k|k}$, we only need the last sealed block row, row $k-1$, and the incomplete block rows of step k. If we also need smoothed estimates but only for time step $i+1$ and higher, we need to store block rows $i+1$ and higher, but not earlier rows.

Dropping old rows that would not be used for smoothing in the future saves memory. **UltimateKalman** allows the user to *forget* the rows of steps $\leq i$ from $R^{(k)}$ and $y^{(k)}$.

Smoothing uses $R^{(k)}$ but does not modify it. **UltimateKalman** also retains $y^{(k)}$ when smoothing. This allows the algorithm to smooth again later if more observations are obtained, enabling easy implementation of strategies such as *fixed-lag smoothing* (Grewal and Andrews 2015), in which each state is estimated once from past observations and from observations of the next n steps for some fixed lag n.

In many cases it is useful to predict future state before any observations of them are available. In particular, by comparing the expectation of the observations of a predicted state $G_k \hat{u}_{k|k-1}$ with the actual observation vector o_k it is sometimes possible to detect and discard outlier observations.

UltimateKalman allows the user to predict future states while retaining the ability to provide observations later. To do so, **UltimateKalman** stores with sealed rows the incomplete diagonal block $\bar{R}_{k,k}$ and the associated right hand-side $\bar{y}_i$. If the user later asks

UltimateKalman to *roll back* to step k, the algorithm discards from memory sealed rows k and higher and restores $\bar{R}_{k,k}$ and $\bar{y}_i$ as the bottommost incomplete row. Obviously, it is not possible to roll back to a forgotten step.

3.5. Computing the covariance matrices of estimates

UltimateKalman computes representations of the covariance matrices $\mathsf{COV}(\hat{u}_{i|k})$ of estimates using orthogonal transformations of $R^{(k)}$. We assume here that $\tilde{R}_{k,k}$ is square and triangular (otherwise existing steps are not yet observable).

The covariance matrix of the filtered estimate satisfies

$$\mathsf{COV}(\hat{u}_{k|k})^{-1} = \tilde{R}_{k,k}^T \tilde{R}_{k,k} \ ,$$

so **UltimateKalman** simply returns $\tilde{R}_{k,k}$ as a representation of $\mathsf{COV}(\hat{u}_{k|k})$. We refer to this representation as an *inverse-factor* representation.

Producing inverse-factor representations of smoothed estimates requires a series of orthogonal transformations. The algorithm first computes the QR factorization of the bottom-right 2-by-1 block of

$$R^{(k)} = \begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & R_{k-1,k-1} & R_{k-1,k} \\ & & & \tilde{R}_{k,k} \end{bmatrix}$$

and applies the transformation to the entire two bottom block rows, to produce

$$\begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & S_{k-1,k-1} & S_{k-1,k} \\ & & S_{k,k-1} & 0 \end{bmatrix}$$

with $S_{k,k-1}$ square. This block satisfies

$$\mathsf{COV}(\hat{u}_{k-1|k})^{-1} = S_{k,k-1}^T S_{k,k-1} \ ,$$

so it is returned as a representation of the covariance matrix. The algorithm now permutes the last two block rows to obtain

$$\begin{bmatrix} \ddots & \ddots & & \\ & R_{k-2,k-2} & R_{k-2,k-1} & \\ & & S_{k,k-1} & 0 \\ & & S_{k-1,k-1} & S_{k-1,k} \end{bmatrix}$$

and continues in the same way. The process continues with the QR factorization of

$$\begin{bmatrix} R_{k-2,k-1} \\ S_{k,k-1} \end{bmatrix} ,$$

repeating the same procedure. The correctness of this algorithm is shown in (Paige and Saunders 1977) and in a perhaps slightly clearer way, in (Toledo 2020).

The computation proceeds upwards in $R^{(k)}$ and can produce the covariance matrices of all the steps that have not been forgotten.

4. Implementation

UltimateKalman is currently available in MATLAB, C, and Java. Each implementation is separate and does not rely on the other ones. The implementation includes MATLAB adapter classes that allow invocation of the C and Java implementations from MATLAB. This allows a single set of test functions to test all three implementations.

The programming interfaces of all three implementations are similar. They offer exactly the same functionality using the same abstractions, and each employs good programming practices of the respected language. For example, the MATLAB and Java implementations use overloading (using the same method name more than once, with different argument lists). Another example is a method that returns two values in the MATLAB implementation, but only one in the others; the second value is returned by a separate method or function in the Java and C implementations. The only differences are ones that are unavoidable due to the constraints of each programming language.

The MATLAB implementation does not rely on any MATLAB toolbox, only on functionality that is part of the core product. The implementation also works under **GNU Octave**. The C implementation relies on basic matrix and vector operations from the **BLAS** (Dongarra, Du Croz, Hammarling, and Duff 1990b; Dongarra, Cruz, Hammarling, and Duff 1990a) and on the QR and Cholesky factorizations from **LAPACK** (Anderson, Bai, Bischof, Blackford, Dongarra, Croz, Greenbaum, Hammarling, McKenney, and Sorensen 1999). The Java implementation uses the Apache Commons Math library for both basic matrix-vector operations and for the QR and Cholesky factorizations. The Cholesky factorization is used only to factor covariance matrices that are specified explicitly, as opposed to being specified by inverse factors or triangular factors.

We begin by describing how the different implementations represent matrices, vectors, and covariance matrices. Then we describe in detail the MATLAB programming interface and implementation and then comment on the differences between them and those of the other two implementations. We end the section with a discussion of the data structures that are used to represent the step sequence and of a mechanism for measuring the performance of the implementations.

4.1. The representation of vectors and matrices

The MATLAB implementation uses native MATLAB matrices and vectors. The Java implementation uses the types `RealMatrix` and `RealVector` from the **Apache Commons Math** library (both are interface types with multiple implementations).

The C implementation defines a type called `matrix_t` to represent matrices and vectors. The implementation defines functions that implement basic operations of matrices and vectors of this type. The type is implemented using a structure that contains a pointer to an array of double-precision elements, which are stored columnwise, as in the **BLAS** and **LAPACK**, and integers that describe the number of rows and columns in the matrix and the stride along rows (the so-called leading dimension in the **BLAS** and **LAPACK** interfaces). To avoid name-space pollution, in client code this type is called `kalman_matrix_t`.

4.2. The representation of covariance matrices

Like all Kalman filters, **UltimateKalman** consumes covariance matrices that describe the distribution of the error terms and produces covariance matrices that describe the uncertainty in the state estimates $\hat{u}_i$. The input covariance matrices are not used explicitly; instead, the inverse factor W of a covariance matrix $C = (W^T W)^{-1}$ is multiplied, not necessarily explicitly, by matrices or by a vector.

Therefore, the programming interface of **UltimateKalman** expects input covariance matrices to be represented as objects belonging to a type with a method `weigh` that multiplies the factor W by a matrix A or a vector v. In the MATLAB and Java implementations, this type is called `CovarianceMatrix`. The constructors of these classes accept many representations of a covariance matrix:

- An explicit covariance matrix C; the constructor computes an upper triangular Cholesky factor U of $C = U^T U$ and implements `X=C.weigh(A)` by solving $UX = A$.
- An inverse factor W such that $W^T W = C^{-1}$; this factor is stored and multiplied by the argument of `weigh`.
- An inverse covariance matrix C^{-1}; the constructor computes its Cholesky factorization and stores the lower-triangular factor as W.
- A diagonal covariance matrix represented by a vector w such that $W = \mathrm{diag}(w)$ (the elements of w are inverses of standard deviations).
- A few other, less important, variants.

In the MATLAB implementation, the way that the argument to the constructor represents C is defined by a single-character argument (with values `C`, `W`, `I`, and `w`, respectively). In the Java implementation, `CovarianceMatrix` is an interface with two implementing classes, `DiagonalCovarianceMatrix` and `RealCovarianceMatrix`; the way that the numeric argument represents C is specified using `enum` constants.

Covariance input matrices are passed to the C implementation in a similar manner, but without a class; each input covariance matrix is represented using two arguments, a matrix and a single character that defines how the matrix is related to C.

UltimateKalman always returns the covariance matrix of $\hat{u}_i$ as an upper-triangular inverse factor W. The MATLAB and Java implementations return covariance matrices as objects of the CovarianceMatrix type (always with an inverse-factor representation); the C implementation simply returns the inverse factor as a matrix.

4.3. The MATLAB programming interface

The MATLAB implementation is object oriented and is implemented as a handle (reference) class. The constructor takes no arguments.

```
kalman = UltimateKalman()
```

The (overloaded) methods that advance the filter through a sequence of steps are evolve and observe. Each of them must be called exactly once at each step, in this order. The evolve method declares the dimension of the state of the next step and provides all the known quantities of the evolution equation (1),

```
kalman.evolve(n_i, H_i, F_i, c_i, K_i)
```

where n_i is an integer, the dimension of the state, H_i and F_i are matrices, c_i is a vector, and K_i is a CovarianceMatrix object. The number of rows in H_i, F_i, and c_i must be the same and must be equal to the order of K_i; this is the number ℓ_i of scalar evolution equations. The number of columns in H_i must be n_i and the number of columns in F_i must be equal to the dimension of the previous step. A simplified overloaded version defines H_i internally as an n_i-by-n_{i-1} identity matrix, possibly padded with zero columns

```
kalman.evolve(n_i, F_i, c_i, K_i)
```

If $n_i > \ell_i$, this overloaded version adds the new parameters are added to the end of the state vector.

If $n_i < \ell_i$, the first version must be used; this forces the user to specify how parameters in u_{i-1} are mapped to the parameters in u_i. The evolve method must be called even in the first step; this design decision was taken mostly to keep the implementation of all the steps in client code uniform. In the first step, there is no evolution equation, so the user can pass empty matrices to the method, or call another simplified overloaded version:

```
kalman.evolve(n_i)
```

The observe method comes in two overloaded versions. One of them must be called to complete the definition of a step. The first version describes the observation equation and the second tells **UltimateKalman** that there are no observations of this step.

```
kalman.observe(G_i, o_i, C_i)
kalman.observe()
```

Steps are named using zero-based integer indexes; the first step that is defined is step $i = 0$, the next is step 1, and so on. The `estimate` methods return the estimate of the state at step `i` and optionally the covariance matrix of that estimate, or the estimate and covariance of the latest step that is still in memory (normally the last step that was observed):

```
[estimate, covariance] = kalman.estimate(i)
[estimate, covariance] = kalman.estimate()
```

If a step is not observable, `estimate` returns a vector of n_i `NaN`s (not-a-number, an IEEE-754 floating point representation of an unknown quantity).

The `forget` methods delete from memory the representation of all the steps up to and including i, or all the steps except for the latest one that is still in memory.

```
kalman.forget(i)
kalman.forget()
```

The `rollback` methods return the filter to its state just after the invocation of `evolve` in step `i`, or just after the invocation of `evolve` in the latest step still in memory.

```
kalman.rollback(i)
kalman.rollback()
```

The methods `earliest` and `latest` are queries that take no arguments and return the indexes of the earliest and latest steps that are still in memory.

The `smooth` method, which also takes no arguments, computes the smoothed estimates of all the states still in memory, along with their covariance matrices. After this method is called, `estimate` returns the smoothed estimates. A single step can be smoothed many times; each smoothed estimate will use the information from all past steps and the information from future steps that are in memory when `smooth` is called.

4.4. The Java programming interface

The programming interface to the Java implementation is nearly identical. It also uses overloaded methods to express default values. It differs from the MATLAB interface only in that the `estimate` methods return only one value, the state estimate. To obtain the matching covariance matrix, client code must call a separate method, `covariance`.

4.5. The C programming interface

In the C interface, the filter is represented by a pointer to a structure of the `kalman_t` type; to client code, this structure is opaque (there is no need to directly access its fields). The filter is constructed by a call to `kalman_create`, which returns a pointer to `kalman_t`.

In general, the memory management principle of the interface (and the internal implementation) is that client code is responsible for freeing memory that was allocated by a call to any function whose name includes the word `create`. Therefore, when client code no longer needs a filter, it must call `kalman_free` and pass the pointer as an argument.

The functionality of the filter is exposed through functions that correspond to methods in the MATLAB and Java implementations. These functions expect a pointer to `kalman_t` as their first argument. The functions are not overloaded because C does not support overloading. Missing matrices and vectors (e.g., to `evolve` and `observe`) are represented by a `NULL` pointer and default step numbers (to `forget` , `estimate`, and so on) by -1. Here is the declaration of some of the functions.

```
kalman_t*         kalman_create     ();
void              kalman_free       (kalman_t* kalman);
void              kalman_observe    (kalman_t* kalman,
                                     kalman_matrix_t* G_i, kalman_matrix_t* o_i,
                                     kalman_matrix_t* C_i, char C_i_type);
int64_t           kalman_earliest   (kalman_t* kalman);
void              kalman_smooth     (kalman_t* kalman);
kalman_matrix_t* kalman_estimate   (kalman_t* kalman, int64_t i);
kalman_matrix_t* kalman_covariance(kalman_t* kalman, int64_t i);
void              kalman_forget     (kalman_t* kalman, int64_t i);
...
```

Note that input covariance matrices are represented by a `kalman_matrix_t` and a representation code (a single character). The output of `kalman_covariance` is a matrix W such that $(W^T W)^{-1}$ is the covariance matrix of the output of `kalman_estimate` on the same step.

A small set of helper functions allows client code to construct input matrices in the required format, to set their elements, and to read and use matrices returned by **UltimateKalman**. Here are the declarations of some of them.

```
kalman_matrix_t* matrix_create(int32_t rows, int32_t cols);
void              matrix_free(kalman_matrix_t* A);

void   matrix_set(kalman_matrix_t* A, int32_t i, int32_t j, double v);
double matrix_get(kalman_matrix_t* A, int32_t i, int32_t j);

int32_t matrix_rows(kalman_matrix_t* A);
int32_t matrix_cols(kalman_matrix_t* A);
...
```

Client code is responsible for freeing matrices returned by `kalman_estimate` and `kalman_covariance` by calling `matrix_free` when they are no longer needed.

4.6. Data structures for the step sequence

The Java implementation uses an `ArrayList` data structure to represent the sequence of steps that have not been forgotten or rolled back, along with an integer that specifies the step number of the first step in the `ArrayList`. The data structure allows **UltimateKalman** to add steps, to trim the sequence from both sides, and to access a particular step, all in constant time or amortized constant time.

The C implementation uses a specialized data structure with similar capabilities. This data structure, called in the code `farray_t`, is part of **UltimateKalman**. The sequence is stored in an array. When necessary, the size of the array is doubled. The active part of the array is not necessarily in the beginning, if steps have been forgotten. When a step is added and there is no room at the end of the physical array, then either the array is reallocated at double its current size, or the active part is shifted to the beginning. This allows the data structure to support appending, trimming from both sides, and direct access to a step with a given index, again in constant or amortized constant time.

The Java implementation stores the steps in a cell array. The implementation is simple, but not as efficient as the data structure that is used by the C version.

4.7. Support for performance testing

All the implementations include a method, called `perftest`, designed for testing the performance of the filter. This method accepts as arguments all the matrices and vectors that are part of the evolution and observation equations, a step count, and an integer d that tells the method how often to take a wall-clock timestamp. The method assumes that the filter has not been used yet and executes the filter for the given number of steps. In each step, the state is evolved, observed, the state estimate is requested, and the previous step (if there was one) is forgotten. The same fixed matrices and vectors are used in all steps.

This method allows us to measure the performance of all the implementations without the overheads associated with calling C or Java from MATLAB. That is, the C functions are called in a loop from C, the Java methods are called from Java, and the MATLAB methods from MATLAB.

The method takes a timestamp every d steps and returns a vector with the average wall-clock running time per step in each nonoverlapping group of d steps.

5. Examples and tests

We implemented an extensive set of tests for **UltimateKalman**. The individual tests are implemented by several MATLAB functions. Most of the functions receive as a first argument a handle to a function that serves as a factory of **UltimateKalman** filters. These functions are invoked by a top-level function called `replication` that defines the factory function and performs the tests. The factory function can produce objects of either the MATLAB implementation or objects of adapter classes that invoke the C or

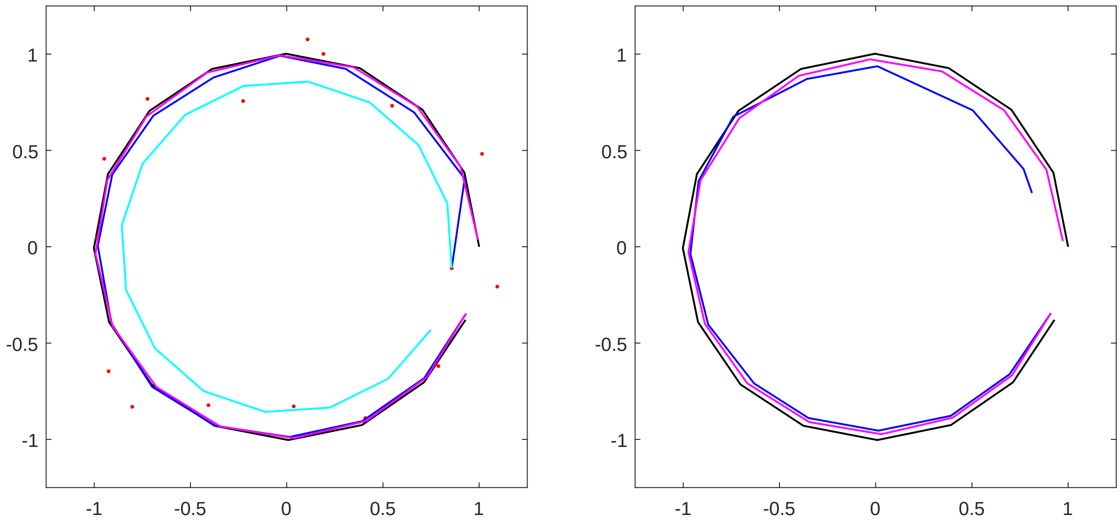

Figure 1: Kalman filtering, prediction, and smoothing of the trajectory of a rotating point in the plane. On the left $G = I$ and observations are indicated by red dots. The black curve represents a simulated system; the blue curve represents filtered estimates, the magenta whole-trajectory smoothed estimates, and the cyan predictions from one observation. On the right $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$; we cannot predict the estate from one such observation, and we cannot produced a filtered estimate of the first state.

Java implementation.

The function that performs performance testing is a little different: it includes a factory function and it can test multiple implementations, to enable plotting their performance on one graph.

The tests generate graphs similar to the ones presented below. The user can inspect the output visually, to ensure that the results are similar to those presented here. The tests do not produce pass/fail flags.

The tests were run on a laptop with an Intel i7 processor running Windows 11 using MATLAB version R2021b. We also verified that the MATLAB implementation works correctly under **GNU Octave**. The C version is compiled into a MATLAB-callable dynamic link library (a so-called `mex` file) by MATLAB itself using a script, `compile.m`. In our tests, MATLAB used the C compiler from Microsoft's **Visual Studio** 2019. The Java version is compiled using Eclipse so that it can be used by Java 1.8 and up (this is the version that MATLAB R2021b uses) and is packaged into a `jar` file by a simple shell script, `build.bat`.

5.1. Basic tests

We demonstrate and test the basic features of **UltimateKalman** using a simple model of a point in the plane that rotates around the origin. The initial state is $\begin{bmatrix} 1 & 0 \end{bmatrix}^T$. The evolution matrix is

$$F = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{bmatrix}$$

for $\alpha = 2\pi/16$, the observation matrix G has between 1 and 6 rows, with an identity in the first two rows, $K = 0.001^2 I$ and $C = 0.1^2 I$. With these parameters, the rotation is very accurate, but the observations are not. The system is simulated for 16 steps, just short of a complete rotation.

Figure 1 shows the results of simulation, predicting, filtering, and smoothing with $G = I$ and with $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$. The code, called `rotation`, first simulates the system and produces the ground-truth $u_1, \ldots, u_{15}$ and the observations $o_0, \ldots, o_{15}$. Then the code creates an **UltimateKalman** filter and runs it for 16 steps while providing only the first observation o_0. This attempts to predict $u_1, \ldots, u_{15}$. Then the code rolls back to step 1 and runs the filter again, providing all the observations. Finally, the code smooths the trajectory and collects the smoothed observations.

The first observation in the case of $G = I$ is quite inaccurate, so predictions from it are far from the real track, but they do follow nicely the system dynamics of exact rotation. The filtered estimates, on the other hand, improve quickly. The smooth estimates are nearly perfect, at least visually.

When $G = \begin{bmatrix} 1 & 0 \end{bmatrix}$ the block $\bar{R}_{0,0}$ is flat, so the algorithm fails to produce a filtered estimate of u_0, but it does produce filtered estimates of $u_1, \ldots, u_{15}$. Predicting from

This example is also used to test the code with overdetermined Gs.

5.2. Variance variations

The next example demonstrates and tests the evaluation of covariance matrices, and it also demonstrates the effectiveness of Kalman filtering even when the model of the system is not perfect.

The code simulates a scalar that evolves either according to $u_i = u_{i-1} + \epsilon_i$ with $\epsilon_1 \sim N(0,1)$ (the errors are distributed normally with expectation 0 and standard deviation 1) or according to $u_i = u_{i-1} + 0.2$. However, even in the latter case, the Kalman filter uses the $u_i = u_{i-1} + \epsilon_i$ evolution equations, which do not reflect the true dynamics of the system. The observations are direct, $o_i = u_i + \delta_i$, with δ_i having a standard deviation of 10 in almost all cases.

Figure 2 shows results for the case $u_i = u_{i-1} + 0.2$. The graphs show both the estimates and their standard deviations. We see that the variance of the filtered estimates drops quickly in the first few steps and then stabilizes. If the standard deviation of o_{50} is much smaller than the rest, 0.25, the variance of the filtered estimate also drops at that steps, but it climbs back up. The variance of the smoothed estimates is impacted in both direction around step 50. It also increases towards iteration 0 and 100, but not dramatically.

All the estimates track the trajectory nicely, even though the Kalman filter and smoother

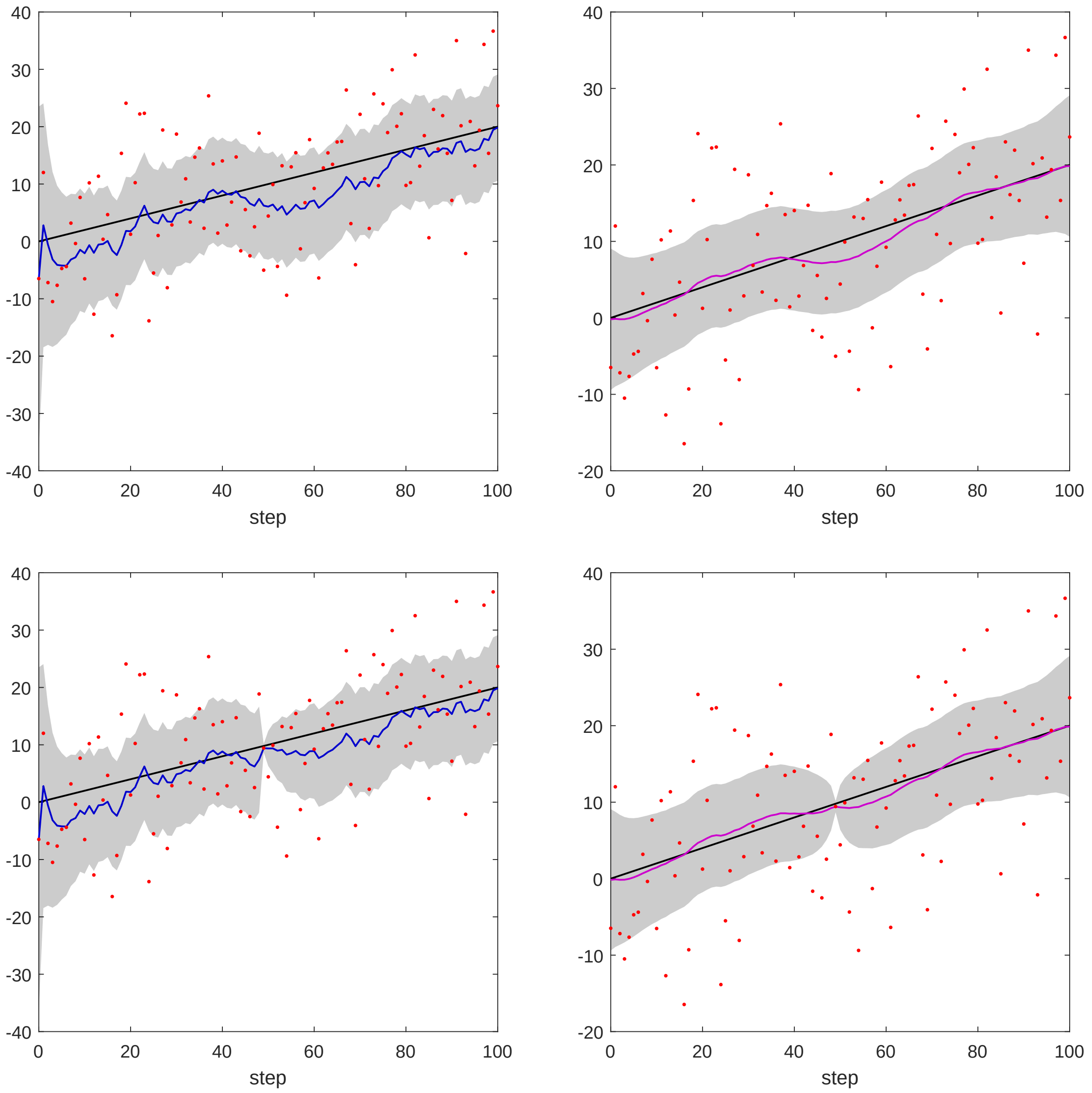

Figure 2: Kalman filtering and smoothing of a scalar trajectory. The black lines show the actual trajectory, the red dots show the observations, the blue lines show the filtered estimates (left column), the magenta lines show smoothed estimates (right column), and the gray areas represents values that are $\pm 3\hat{\sigma}_i$ from the estimate $\hat{u}_i$, where $\hat{\sigma}_i$ is the estimated standard deviation of $\hat{u}_i$. In the top row, the standard deviation of δ_i is always 10; in the bottom row, it drops down to 0.25 at iteration 50 and then goes back up to 10.

use evolution equations that are different from those of the actual dynamic system. This is an important reason that Kalman filtering is so useful in practice: it often works well even when it models the dynamic system only approximately.

The results for a simulation with $u_i = u_{i-1} + \epsilon_i$ are similar and not shown in the paper.

5.3. Adding and removing parameters

The `add_remove` example shows how to add and remove parameters from the state vector. In the first two steps, the filter tracks the constant 1 using the evolution equation $u_i = u_{i-1} + \epsilon_i$ and observation equation $o_i = u_i + \delta_i$ with both error terms having a standard deviation of 0.1. In the third step, we add a dimension to the state; the first argument to `evolve` is now 2. This causes **UltimateKalman** to construct and use a matrix

$$H_i = \begin{bmatrix} 1 & 0 \end{bmatrix} ,$$

which represent a normal evolution of the first component of the state while not using any historical information about the second, new component. The evolution matrix F_2 remains a 1-by-1 identity in this step, but G_2 is now a 2-by-2 identity; o_2 becomes 2-dimensional as well. The actual value of the second parameter is 2 and the observations reflect that. In the next iteration, F_3 grows to 2-by-2. After two iterations with a 2-dimensional state vector, we drop the first (original) component of the state vector by calling `evolve` with a first argument 1 and by providing an explicit matrix

$$H_4 = \begin{bmatrix} 0 \\ 1 \end{bmatrix} .$$

This matrix causes the filter to retain the second component of the state and to drop the first. In this step F_4 remains 2-by-2 but G_4 is 1-by-1. In the next iteration F_5 shrinks back to a 1-by-1 identity.

This demonstrates how to handle addition and removal of parameters and tests that **UltimateKalman** handles these cases correctly. The evolution and observation equations are very simple and track the two parameters separately, but this is immaterial for the addition and removal procedures.

5.4. A Projectile problem

The next example that we consider is taken almost as is from Humpherys et al. (Humpherys *et al.* 2012). The problem uses a linear dynamic system to model a projectile. The states are four-dimensional; they model the horizontal and vertical displacements and the horizontal and vertical velocities. The model accounts for gravity, reducing the vertical velocity by a constant every time step, and for drag, which scales both components of the velocity by a constant in every step. The matrices and vectors associated the system

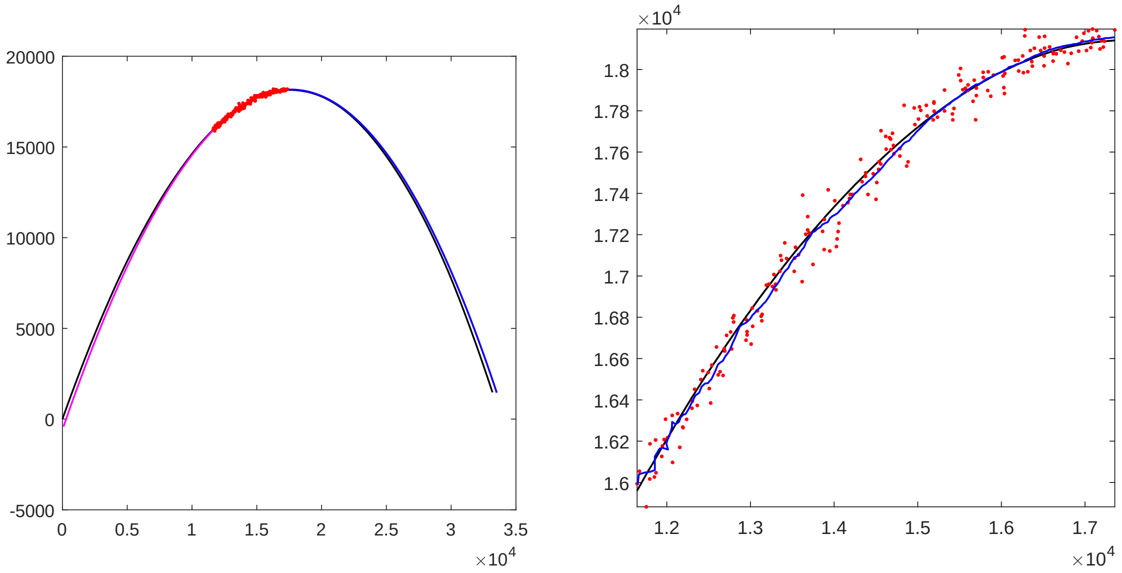

Figure 3: Kalman filtering and smoothing of the trajectory of a projectile affected by gravity and drag. Again, the black line show the actual trajectory, the red dots show the observations, the blue line shows the filtered estimates of the trajectory, and the magenta line shows the smoothed estimates. Smoothing was performed on the entire trajectory, to estimate the point of departure and the point where the projectile hits the ground. The graph on the left shows the entire trajectory and the graph on the right only the part in which observations are available.

are

$$
\begin{aligned}
F_i = F &= \begin{bmatrix} 1 & 0 & \Delta t & 0 \\ 0 & 1 & 0 & \Delta t \\ 0 & 0 & 1-b & 0 \\ 0 & 0 & 0 & 1-b \end{bmatrix} \quad \text{and} \quad c_i = c = \begin{bmatrix} 0 \\ 0 \\ 0 \\ -9.8\Delta t \end{bmatrix} \\
G_i = G &= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} ,
\end{aligned}
$$

where $\Delta t = 0.1$ and $b = 10^{-4}$. The example in. (Humpherys *et al.* 2012) invites the reader to simulate the dynamic system for 1200 steps of 0.1 s each, starting from a known state, to generate noisy observations of the displacements (not the velocities) in steps 400 to 600, and to estimate the trajectory using a Kalman filter. They also ask the reader to use the filter to predict when and where the projectile will fall back to its original altitude, and to estimate the point of departure by reversing the dynamic system.

We have successfully applied **UltimateKalman** to this problem. The example code, `projectile`, generated the plots in Figure 3, which are similar to Figures 7.1a and 7.1b in (Humpherys *et al.* 2012) (it seems that the measurement noise shown in their Figure 7.1b has variance larger than the value 500 specified in (Humpherys *et al.* 2012)).

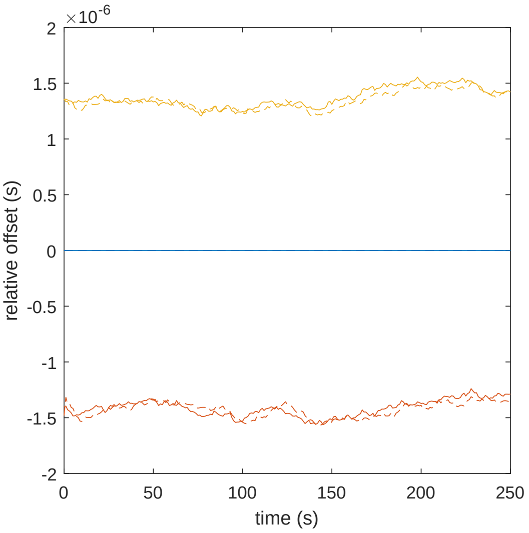

Figure 4: Kalman tracking of the relative offsets of three clocks from times of arrival of radio packets. The graphs show the relative offsets relative to the offset of the first (blue) clock. The solid lines are the simulated offsets and the dashed lines are the filtered estimates.

UltimateKalman also allows the user to easily extrapolate the trajectory by evolving the filter without providing additional observations, and to estimate the point of departure by smoothing time steps 0 to 600. Reversing the dynamic system is not required (and in particular, it is not necessary to invert the evolution matrix).

5.5. Clock offsets in a distributed system

The `clock_offsets` example highlights the utility of the H_i matrices. The aim is to estimate the relative offsets of a set of clocks in a distributed system, such as a wireless sensor network. Each clock is associated with a receiver and we assume that at time τ, clock j shows $t_j = \tau + f_{ij}$ where f_{ij} is the offset of clock j from real time at the time in which it displays the value t_j. The receivers receive radio packets from a beacon transmitter. The locations of the transmitter and receivers are known, so the line-of-sight propagation delays d_j between the transmitter and receiver j are also known. The receivers estimate the time of arrival of the packets using their local imperfect clocks. The observation equation for the time of arrival of packet i at receiver j is

$$t_{ij} = \tau_i + d_j + f_{ij} + \delta_{ij} \ ,$$

where t_{ij} is the time-of-arrival estimate, as represented by imperfect clock j, τ_i is the unknown time of departure of the ith packet, f_{ij} is the offset of clock j at (local) time t_{ij}, d_j is the known delay to receiver j, and δ_{ij} is the time-of-arrival estimation error.

The evolution equations are very simple:

$$f_{ij} = f_{i-1,j} + \epsilon_{ij} \ .$$

They express the belief that the offsets change slowly. Note that the number of evolution equations is equal to the number of clocks, so is smaller than the dimension of the state vectors by one. The code uses the following matrices and vectors, including an explicit fixed H_i:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} f_{i1} \\ f_{i2} \\ \vdots \\ f_{im} \\ \tau_i \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ & & \ddots & & \\ 0 & 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} f_{i-1,1} \\ f_{i-1,2} \\ \vdots \\ f_{i-1,m} \\ \tau_{i-1} \end{bmatrix} + \begin{bmatrix} \epsilon_{i1} \\ \epsilon_{i2} \\ \vdots \\ \epsilon_{1\ell} \end{bmatrix}$$

$$\begin{bmatrix} t_{i1} - d_1 \\ t_{i2} - d_2 \\ \vdots \\ t_{im} - d_m \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 & 1 \\ 0 & 1 & \cdots & 0 & 1 \\ \vdots & & & & \\ 0 & 0 & \cdots & 1 & 1 \end{bmatrix} \begin{bmatrix} f_{i-1,1} \\ f_{i-1,2} \\ \vdots \\ f_{i-1,m} \\ \tau_{i-1} \end{bmatrix} + \begin{bmatrix} \delta_{i1} \\ \delta_{i2} \\ \vdots \\ \delta_{i\ell} \end{bmatrix} .$$

The structure of H_i and F_i reflects the removal of τ_{i-1} and the introduction of τ_i in every step.

The problem as presented up to now is clearly rank deficient, because the residual is invariant under an addition of a constant T to all the offsets and a subtraction of T from all the depsarture times. Nothing anchors the solution relative to absolute time; indeed, we only want to estimate the relative offsets. To address this issue, we add a pseudo-observation of one offset in the first step. This removes the rank deficiency.

This model can be easily extended to allow for packets that are not received by all receivers, for multiple transmitters, for slowly-changing rate-errors instead of slowly-changing offsets, and so on. the model can also allow receivers to join and leave the system without restarting the estimation process, as we have done in Section 5.3.

The results are shown in Figure 4.

5.6. Performance testing

Figure 5 shows result of testing the performance of the three implementations using their `perftest` method. The experiments were carried out on a laptop with an Intel i7 processor. The test uses random square unitary matrices for F_i and G_i are fixed, identities for C_i and K_i, $c_i = 0$, and a random Gaussian vector for o_i. The matrices H_i are identities created by the algorithm itself; they are not passed as arguments. The use of unitary matrices avoids overflows and underflows.

The graphs in the figure show the average running times per step, averaging over groups of 1000 steps.

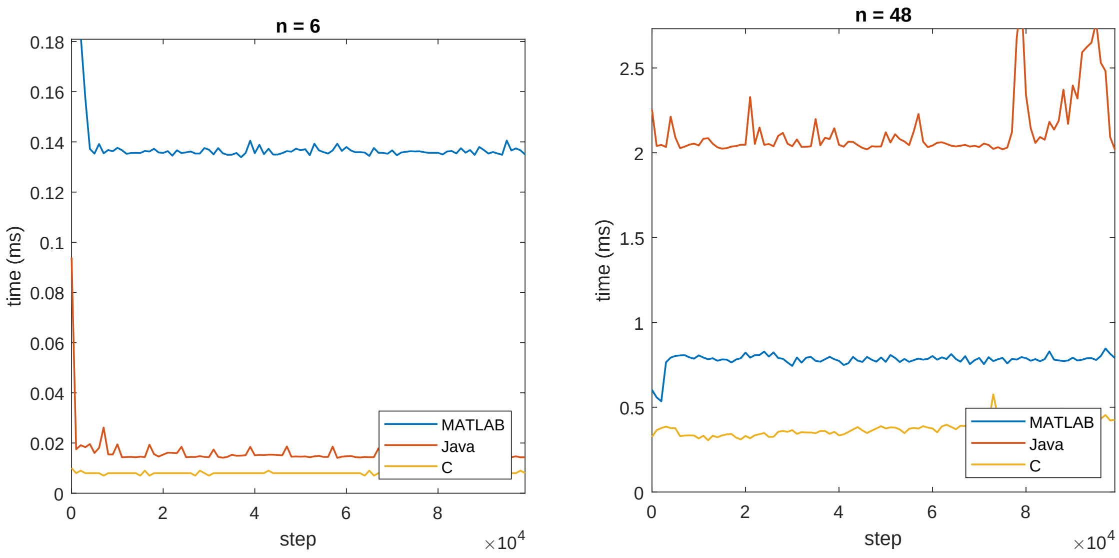

Figure 5: Performance testing of the three **UltimateKalman** implementations. The graphs show the time per step (window averages of nonoverlapping groups of steps) when the algorithm filtered systems with random square unitary matrices F_i and G_i.

The results show that on small problems ($n_i = 6$), the C implementation is the fastest, taking about $8\,\mu$s per step. The Java implementation is about a factor of two slower, with periodic jumps that are most likely caused by the garbage collector. The MATLAB implementation is much slower, taking about $140\,\mu$s per step. Both the Java and MATLAB implementations are initially even slower, probably due to just-in-time compilation of the code.

On larger problems ($n_i = 48$), the C implementation is still the fastest. The MATLAB implementation is now only about two times slower than the C implementation. The improved ratio is most likely the result of $\Theta(n^3)$ dense-matrix operations (matrix multiplications and QR factorizations) taking a significant fraction of the running times. The data structure and method invocation overheads in C are much smaller than in MATLAB, but dense-matrix operations are performed at the same rate. On these larger problems, the Java implementation is the slowest, most likely because of the relative poor performance of the **Apache Commons Math** library relative to the **BLAS** and **LAPACK** implementations that come with MATLAB.

Some of the changes in the running times that are visible in some of the plots are likely due to the computer, a laptop, slowing down to avoid overheating.

6. Discussion

Orthogonal transformations are the bedrock of numerical linear algebra. The numerical stability of algorithms that rely solely or mostly on orthogonal transformations is often both superior and easier to analyze than the stability of algorithms that use non-orthogonal transformations or explicit matrix inversion. From this standpoint, the fact that the Paige-Saunders algorithm did not become the standard linear Kalman filter and smoother is both puzzling and unfortunate. There is no good reason not to use it.

UltimateKalman aims to rectify this defect. It is a simplified version of the Paige-Saunders algorithm that is easier to implement, hopefully also easier to understand, and is probably just as efficient. **UltimateKalman** also generalizes the Paige-Saunders algorithm, making it more flexible. While there are certainly many high-quality and well-documented Kalman filter implementations, for example (Torres 2010; Tusell 2011), we are not aware of any other linear Kalman filter algorithm that is as flexible and that can handle problems with varying state-vector dimensions, does not require the expectation of the initial state, can handle missing observations, and can easily filter, predict, and smooth.

The MATLAB implementation has been optimized for clarity and conciseness, not for computational efficiency. In particular, the sequence of steps in memory is represented by a dynamically-resized cell array. We made this design choice in order to make **UltimateKalman** easy to port to other languages and easy to specialize.

The C implementation was optimized for computational and memory efficiency, but while retaining full generality (e.g., it can handle changes in the dimension of the state vector, and it can smooth, like the other implementations) and without a specialized memory manager. The main cost of these design decision is the dynamic allocation and deallocation of several matrices in every step. Specializing the algorithm to simple special cases, say only filtering and no changes in the state-vector dimension, would have allowed the algorithm to avoid dynamic memory allocation, making it faster and perhaps better suited to tiny embedded systems.

We hope that the numerical stability, the flexibility, the convenient programming interfaces, and the availability of the algorithm in multiple languages will make **UltimateKalman** the standard linear Kalman filter and smoother in new software. We also hope that authors and maintainers of statistical software packages of wider scope (Peng and Aston 2011; Villegas and Pedregal 2018), which often includes nonlinear Kalman filters and other state-space models, will also incorporate **UltimateKalman** into their packages, ideally exposing some of this unique features through their own programming and user interfaces. Finally, we hope that authors of code generators that automatically generate Kalman filter code optimized for specific cases (Whittle and Schumann 2004) will also incorporate the algorithm into their generators; the code clarity that we strived to achieve should simplify such efforts.

Acknowledgements

This research was supported in part by grant 1919/19 from the Israel Science Foundation and by a grant from the Israeli Ministry of Science.

References

Aitken AC (1936). "On Least Squares and Linear Combinations of Observations." *Proceedings of the Royal Society of Edinburgh*, **55**, 42–48. doi:10.1017/S0370164600014346.

Anderson E, Bai Z, Bischof C, Blackford S, Dongarra JDJ, Croz JD, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999). *LAPACK Users' Guide*. 3rd edition. SIAM, Philadelphia, PA, USA.

Björck Å (1996). *Numerical Methods for Least Squares Problems*. SIAM, Philadelphia, PA, USA. ISBN 0-89871-360-9.

Brown RG, Hwang PYC (1997). *Introduction to Random Signals and Applied Kalman Filtering*. 3rd edition. Wiley.

Dongarra JJ, Cruz JD, Hammarling S, Duff IS (1990a). "Algorithm 679: A Set of Level 3 Basic Linear Algebra Subprograms: Model Implementation and Test Programs." *ACM Transactions on Mathematical Software*, **16**(1). doi:10.1145/77626.77627.

Dongarra JJ, Du Croz J, Hammarling S, Duff IS (1990b). "A Set of Level 3 Basic Linear Algebra Subprograms." *ACM Trans. Math. Softw.*, **16**(1), 1–17. doi:10.1145/77626.79170.

Duncan DB, Horn SD (1972). "Linear Dynamic Recursive Estimation from the Viewpoint of Regression Analysis." *Journal of the American Statistical Association*, **67**(340), 815–821. doi:10.1080/01621459.1972.10481299.

Givens W (1954). "Numerical Computation of the Characteristic Values of a Real Symmetric Matrix." *Technical Report 1574*, Oak Ridge National Laboratory. doi:10.2172/4412175.

Golub GH, Loan CFV (2013). *Matrix Computations*. 4th edition. Johns Hopkins University Press. ISBN 1-4214-0794-9 (hardcover), 1-4214-0859-7 (ebook).

Grewal MS, Andrews AP (2015). *Kalman Filtering: Theory and Practice with MATLAB*. 4th edition. Wiley.

Higham NJ (2002). *Accuracy and Stability of Numerical Algorithms*. 2nd edition. SIAM, Philadelphia, PA, USA. ISBN 0-89871-521-0.

Householder AS (1958). "Unitary Triangularization of a Nonsymmetric Matrix." *Journal of the ACM*, **5**(4), 339–342. doi:10.1145/320941.320947.

Humpherys J, Redd P, West J (2012). "A Fresh Look at the Kalman Filter." *SIAM Review*, **54**(4), 801–823. doi:10.1137/100799666.

Kalman RE (1960). "A New Approach to Linear Filtering and Prediction Problems." *Journal of Basic Engineering*, **82**(1), 35–45. doi:10.1115/1.3662552.

Paige CC, Saunders MA (1977). "Least Squares Estimation of Discrete Linear Dynamic System Using Orthogonal Transformations." *SIAM Journal on Numerical Analysis*, **14**(2), 180–193. doi:10.1137/0714012.

Peng JY, Aston JAD (2011). "The State Space Models Toolbox for MATLAB." *Journal of Statistical Software*, **41**(6), 1–26. doi:10.18637/jss.v041.i06.

Saunders M (2018). "Private Communication." email.

Toledo S (2020). *Location Estimation from the Ground Up*. Society for Industrial and Applied Mathematics, Philadelphia, PA. doi:10.1137/1.9781611976298.

Torres GA (2010). "Algorithm 900: A Discrete Time Kalman Filter Package for Large Scale Problems." *ACM Transactions on Mathemathical Software*, **37**(1), 11:1–11:16. doi:10.1145/1644001.1644012.

Tusell F (2011). "Kalman Filtering in R." *Journal of Statistical Software*, **39**(2), 1–27. doi:10.18637/jss.v039.i02.

Villegas MA, Pedregal DJ (2018). "SSpace: A Toolbox for State Space Modeling." *Journal of Statistical Software*, **87**(5), 1–26. doi:10.18637/jss.v087.i05.

Whittle J, Schumann J (2004). "Automating the Implementation of Kalman Filter Algorithms." *ACM Transactions on Mathemathical Software*, **30**(4), 434–453. doi:10.1145/1039813.1039816.

Affiliation:

Sivan Toledo
Blavatnik School of Computer Science
Tel Aviv University
Tel Aviv 69978, Israel
URL: https://www.tau.ac.il/~stoledo