User's Guide for UltimateKalman: a Library for Flexible Kalman Filtering and Smoothing Using Orthogonal Transformations

Sivan Toledo

May 1, 2024

UltimateKalman is a library that implements efficient Kalman filtering and smoothing algorithms using orthogonal transformations. The algorithms are based on an algorithm by Paige and Saunders [7], which to the best of our knowledge, has not been implemented before. This guide is part of an article in the journal ACM Transactions on Mathematical Software that describes UltimateKalman. The algorithms are described in that article; this user guide focuses on usage of the software library.

UltimateKalman is currently available in MATLAB, C, and Java. Each implementation is separate and does not rely on the others. 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 [3, 2] and on the QR and Cholesky factorizations from LAPACK [1]. 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 first describe 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. The guide ends with a discussion of the data structures that are used to represent the step sequence and a presentation of a mechanism for measuring the performance of the implementations.

The code has been tested with MATLAB R2021b (version 9.11) and with GNU OCTAVE 7.1.0, both running under Windows 11. The code has also been tested on Linux and on MacOS. Under Windows, MATLAB was configured to use Microsoft Visual C/C++ 2019 to compile C (mex) code. OCTAVE was configured to use mingw64 to compile C (mex) code. The code also compiles as a standalone C program under both GCC and Microsoft Visual C/C++ 2022, as well as under Java version 1.8 (also called version 8) and up.

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`.

State vectors are not always observable. This topic is explained in Section 3.2 in the companion article. This situation usually arises when there are not enough observations to estimate the state. The function calls and methods that return estimates of state vectors and the covariance matrices of the estimates return in such cases a vector of `NaN`s (not-a-number, a floating point value that indicates that the value is not available) and a diagonal matrix whose diagonal elements are `NaN`.

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 $W = \text{diag}(w)$ (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 defined in the implementation classes:

```
RealCovarianceMatrix.Representation.COVARIANCE_MATRIX
RealCovarianceMatrix.Representation.FACTOR
RealCovarianceMatrix.Representation.INVERSE_FACTOR
DiagonalCovarianceMatrix.Representation.COVARIANCE_MATRIX
DiagonalCovarianceMatrix.Representation.DIAGONAL_VARIANCES
DiagonalCovarianceMatrix.Representation.DIAGONAL_STANDARD_DEVIATIONS
DiagonalCovarianceMatrix.Representation.DIAGONAL_INVERSE_STANDARD_DEVIATIONS
```

The `RealCovarianceMatrix` class has a single constructor that takes a `RealMatrix` and a representation constant. The `DiagonalCovarianceMatrix` class several constructors that take either a `RealVector`, an array of `double` values, or a single `double` and a dimension (the covariance matrix is then a scaled identity); all also take as a second argument a representation constant.

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 (`C`, `W`, `I`, or `w`) that defines how the given 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.

3 The MATLAB programming interface

The MATLAB implementation resides in the `matlab` directory of the distribution archive. Add it to your MATLAB search path using MATLAB `addpath` command.

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,

```
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 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 indices; 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 indices 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.

To use the C or Java implementations from within MATLAB, create an object of one of the adapter classes, not an object of the `UltimateKalman` class:

```
kalman = UltimateKalmanNative()
kalman = UltimateKalmanJava()
```

To use the C implementation, you will first need to compile the C code into a MATLAB-callable dynamically-linked library that MATLAB uses through an interface called the `mex` interface. To perform this step, run the `compile.m` script in the `matlab` directory. To use the Java implementation, build the Java library using the instructions in the next section, and add both the resulting library (a `jar` file) and the library containing the Apache Commons Math library to the MATLAB search path using MATLAB `addpath` command. The script `replication.m` adds these libraries to the path and can serve as an example.

4 The Java programming interface

The programming interface to the Java implementation is nearly identical. The implementing class is `sivantoledo.kalman.UltimateKalman`. 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`.

Building and Running the Java Code

The Java implementation resides under the `java` directory of the distribution archive. The sources are in the `src` subdirectory. To use it, you first need to compile the source code and to assemble the compiled code into a library (a `jar` file). The scripts `build.bat` and `build.sh`, both in the `java` directory, perform these steps under Windows (`build.bat`) and Linux and MacOS (`build.sh`). To run the scripts, your computer must have a Java development kit (JDK) installed. We used successfully releases of OpenJDK on both Windows and Linux. The code is compiled so that the library can be used with any version of Java starting with version 8 (sometimes also referred to as 1.8).

The build scripts also compile and run a test program, `Rotation.java`. It performs the same computations and outputs the same values as the MATLAB example function `rotation.m` when executed with arguments `rotation(UltimateKalman,5,2)`. This program also serves as an example that shows how to write Java code that calls UltimateKalman.

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 $(W^TW)^{-1}$ 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.

Building and Running the C Outside Matlab

To help you integrate UltimateKalman into your own native code in C or C++, the software includes three C programs that call UltimateKalman and Windows and Linux scripts that builds executable versions of the three programs. The scripts compile one of the programs along with UltimateKalman, link it with BLAS and LAPACK libraries, and run it. You should be able to use these scripts as examples of how to compile and link UltimateKalman. In particular, the C implementation of UltimateKalman consists of a single C source file, so it is not necessary to build it into a library; the source code or a single object file can be used directly in your C or C++ programs or libraries.

These C programs, the C implementation of UltimateKalman, and the build scripts are all under the `native` directory of the distribution archive.

The three sample programs are `rotation.c`, a C implementation of one of the example MATLAB programs, `blastest.c`, a small program intended only to test the interface to the BLAS and LAPACK libraries, and `performance.c`, a program designed to measure the running time of UltimateKalman. The program `rotation.c` performs the same computation that the expression `rotation(UltimateKalman,5,2)` performs in MATLAB and should output the same numerical results, just like `Rotation.java` in Java. The program `performance.c` implements a Kalman smoother on problems with $n_i = m_i = 6$ or $n_i = m_i = 48$ using orthonormal matrices for F_i and G_i and with $H_i = I$. It measures and reports the running time of the smoother.

The Windows script, `build.bat`, uses the Microsoft C command-line compiler (`cl`) that comes with Microsoft's Visual Studio Community 2022 [6], a free integrated development environment, and the BLAS and LAPACK libraries that are part of Intel's free oneAPI Math Kernel Library (MKL) [5]. The script takes one argument, the name of the program that you want to build and run, `blastest` or `rotation`. Here what you should expect to see on the console when you build and run `blastest` (ellipsis stand for deleted output that is not particularly interesting; the output of the `blastest` program is shown in bold):

```
C:\Users\stoledo\github\ultimate-kalman\native>build.bat blastest
...
**********************************************************************
** Visual Studio 2022 Developer Command Prompt v17.9.2
** Copyright (c) 2022 Microsoft Corporation
**********************************************************************
[vcvarsall.bat] Environment initialized for: 'x64'
:: initializing oneAPI environment...
   Initializing Visual Studio command-line environment...
   Visual Studio version 17.9.2 environment configured.
   "C:\Program Files\Microsoft Visual Studio\2022\Community\"
:  compiler -- latest
:  mkl -- latest
:  tbb -- latest
:: oneAPI environment initialized ::
generating test program blastest.exe
ultimatekalman.c
ultimatekalman.c(39): warning C4005: 'blas_int_t': macro redefinition
ultimatekalman.c(28): note: see previous definition of 'blas_int_t'
blastest.c
Generating Code...
generated test program
BLAS test starting
A = matrix_print 2 3
```

```
1 2 3
4 5 6
B = matrix_print 3 2
7 8
9 10
12 13
C = matrix_print 2 2
125 137
293 323
Result should be:
  125  137
  293  323
BLAS test done
done running test
build script done
```

When you run the Windows executable program yourself, make sure that the directory where the MKL dynamically-linked libraries (`dll` files) are stored is on your `path`. The same is true, of course, for your own programs that use UltimateKalman. The installation of MKL does not modify the path to include this library, but it does provide a script that modifies the search path appropriately. In the version of MKL that I used, this script is `c:\Program Files (x86)\Intel\oneAPI\setvars.bat`.

Under Linux, the corresponding script is `build.sh`. It also takes a single argument, `blastest` or `rotation`. The script provides with the software uses the BLAS and LAPACK libraries that are installed in the Ubuntu Linux distribution using the commands

```
sudo apt install libblas-dev
sudo apt install liblapack-dev
```

These software packages install the libraries in a directory that is already on the search path, so no special configuration of the search path is required. These particular implementations of the BLAS and LAPACK are not high-performance implementations, but since UltimateKalman typically handles fairly small matrices, a high-performance library like Intel's MKL may not provide performance benefits. However, MKL is also available for Linux and you can certainly use it.

To successfully link UltimateKalman with other implementations of the BLAS and LAPACK, you may need to set some preprocessor variables that control how UltimateKalman calls these libraries. Most of the variations are due to the fact that these libraries were originally implemented in Fortran. There are a few other preprocessor variables that control the behavior of UltimateKalman. All of these are listed and explained in Table 1.

6 Testing and Code Examples

The software distribution of UltimateKalman includes five MATLAB functions that demonstrate how to use UltimateKalman:

- `rotation.m`, modeling a rotating point in the plane (this example is also implemented in C and Java, as described above).

variable name	requires a value?	explanation
`BUILD_MKL`	no	specifies that MKL is used; sets all the other BLAS and LAPACK variables (so you do not need to)
`BUILD_BLAS_INT`	yes	C data type that specifies a row or column index or a dimension of a matrix; usually either `int32_t` or `int64_t`
`HAS_BLAS_H`	no	specifies that the `blas.h` header is available
`HAS_LAPACK_H`	no	specifies that the `lapack.h` header is available
`BUILD_BLAS_UNDERSCORE`	no	add an underscore to names of BLAS and LAPACK functions
`BUILD_BLAS_STRLEN_END`	no	add string-length arguments to calls to the BLAS and LAPACK
`BUILD_DEBUG_PRINTOUTS`	no	generates run-time debug printouts
`NDEBUG`	no	suppresses run-time assertion checking
`BUILD_WIN32_GETTIMEOFDAY`	no	required under Windows; do not use under Linux or MacOS

Table 1: Preprocessor variables that control how UltimateKalman calls the BLAS and LAPACK, as well as several other aspects of its behavior.

- `constant.m`, modeling an fixed scalar or a scalar that increases linearly with time, and with observation covariance matrices that are identical in all steps except perhaps for one exceptional step.
- `add_remove.m`, demonstrating how to use H_i to add or remove parameters from a dynamic system.
- `projectile.m`, implementing the model and filter of a projectile described by Humpherys et al. [4].
- `clock_offsets.m`, implementing clock-offset estimation in a distributed sensor system. This example demonstrates how to handle parameters that appear only in one step.

The mathematical details of these models are described in the article that describes UltimateKalman.

The script `replication.m` runs all of these examples and optionally generates the graphs shown in the article. The script can run not only the MATLAB implementation, but also the C and Java implementations. This script assumes that the corresponding libraries have already been built. The version of the C that MATLAB uses is built using the `compile.m` script, as explained in Section 5 above. The Java version should be built outside MATLAB, as explained in Section 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 and 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.

8 Data Structures for the Step Sequence

The information in this section helps to understand the implementations, but is essentially irrelevant to users of UltimateKalman.

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.

References

[1] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, PA, USA, 3rd edition, 1999.

[2] J. J. Dongarra, Jermey Du Cruz, Sven Hammarling, and I. S. Duff. Algorithm 679: A set of level 3 basic linear algebra subprograms: Model implementation and test programs. *ACM Transactions on Mathematical Software*, 16(1), 1990. `doi:10.1145/77626.77627`.

[3] J. J. Dongarra, Jeremy Du Croz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Trans. Math. Softw.*, 16(1):1–17, 1990. `doi:10.1145/77626.79170`.

[4] Jeffrey Humpherys, Preston Redd, and Jeremy West. A fresh look at the Kalman filter. *SIAM Review*, 54(4):801–823, 2012. `doi:10.1137/100799666`.

[5] Intel. *oneAPI Math Kernel Library (oneMKL)*, 2024. downloaded version 2024.0.1 in April 2024. URL: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl-download.html`.

[6] Microsoft. *Visual Studio Community 2022*, 2024. downloaded in April 2024. URL: `https://visualstudio.microsoft.com/vs/community/`.

[7] Christopher C. Paige and Michael A. Saunders. Least squares estimation of discrete linear dynamic system using orthogonal transformations. *SIAM Journal on Numerical Analysis*, 14(2):180–193, 1977. `doi:10.1137/0714012`.