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

Sivan Toledo

March 20, 2025

UltimateKalman is a library that implements efficient and flexible Kalman filtering and smoothing algorithms, including parallel (multi-core) smoothers. The library contains MATLAB, C, and Java implementations (currently, the Java implementations does not contain all the algorithms). The library is robust: it includes mechanisms for testing and evaluating the performance of all the implementations, as well as with a set of well-documented examples. The library is available on GitHub at `https://github.com/sivantoledo/ultimate-kalman`. The release history of the library is as follows:

- **Release 1.2.0** contains the sequential UltimateKalman algorithm, which is documented carefully in an article in the ACM Transactions on Mathematical Software [13]. The algorithm is a slight extension an algorithm by Paige and Saunders [9], which to the best of our knowledge, has not been implemented before. The algorithm uses orthogonal transformations so it has good numerical stability. The algorithm is implemented monolithically in all 3 languages. Please cite this article when citing the sequential UltimateKalman algorithm or its implementation. To use that version, please use the user guide from that release, not this document.

- **Release 2.0.0** contains three additional algorithms in MATLAB and C, including two parallel-in-time smoothers, the Odd-Even smoother proposed in an article by Gargir and Toledo [4] and the smoother proposed by Särkkä and García-Fernández [11]. The code in this release was used to perform the experiments reported by Gargir and Toledo [4]; the release is meant mostly to document these experiments, not for adoption by users.

- **Release 2.1.0** is a significantly cleaned up version of the codes described by Gargir and Toledo [4]. It is meant to be adopted by users. This is the version described in this guide.
  The C codes in this release use the same API as the implementation in Release 1.2.0, but the code has been split into multiple files so the build is more complicated. The MATLAB also retains the same API, but class names have changed. The Java implementation is identical to the implementation in Release 1.2.0.

The implementation in each language 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 MATLABexample functions to invoke 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 C implementation of the parallel-in-time smoothers used the Threading Building Blocks (TBB) library to express shared-memory parallelism. TBB is a C++ library and the code uses a single C++ source file to expose the parallel primitives. The code can also be compiled without TBB, but this generates a sequential algorithm, not a multi-threaded (multi-core) one. 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 R2024a, 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.

The distribution archive contains a number of directories with scripts that build libraries, programs, and this document. The scripts for Windows are called `build.bat`. To run them, type `build` on the Windows command prompt. The scripts for Linux and MacOS are called `build.sh`. To run them, type `./build.sh` in a shell (terminal window). The `build.sh` files must have permissions that allows them to execute as scripts; unpacking the distribution archive normally gives them this permission, but if you receive a `permission denied` error message, give the file this premission using the command `chmod +x build.sh` and try again.

The rest of this guild is organized as follows. Section 1 explains how vectors

and matrices are represented in the three implementations. Section 2 explains how covariance matrices are represented. Section 3 presents the programming interface of the MATLAB implementation and how to add it to MATLAB's search path.

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 $W = \mathrm{diag}(w)$ (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 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.

The UltimateKalman algorithm always returns the covariance matrix of $\hat{u}_i$ as an upper triangular inverse factor W. The other algorithms return an explicit covariance matrix C. The MATLAB and Java implementations return covariance matrices as objects of the `CovarianceMatrix` type (always with an inverse-factor representation); the C implementation includes one function that returns the type of the covariance matrix (a single character) and another that returns the factor W or the explicit matrix C.

3 The MATLAB Programming Interface

The MATLAB implementation resides in the `matlab` directory of the distribution archive. To be able to use it, you must add this directory to your MATLAB search path using MATLAB's `addpath` command.

The MATLAB implementation is object oriented and is implemented as a collection of handle (reference) classes called `KalmanUltimate`, `KalmanConventional`, `KalmanOddevenSmoother`, `KalmanAssociativeSmoother`, and `KalmanSparse`. The first four classes implement, respectively, the UltimateKalman algorithm [13, 9], a conventional Kalman filter [7] and RTS smoother [10], the Gargir-Toledo parallel smoother [4], and the parallel smoother by Särkkä and García-Fernández [11]. The fifth implementation, `KalmanSparse`, uses an explicit sparse QR factorizations to filter and smooth; it is inefficient, especially when filtering, and is meant only

for testing the correctness of other implementations. It is particularly simple and therefore it is particularly easy to ensure that it is correct. The interface of all these classes is exactly the same. The constructor takes one optional argument, a structure that specifies algorithmic options.

```
kalman = KalmanUltimate(options)
```

or

```
kalman = KalmanUltimate()
```

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:

```
kalman = KalmanNative(options)
kalman = KalmanJava(options)
```

These adapter classes have exactly the same interface as the MATLAB implementations (including the fact that the `options` argument is optional).

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 `UltimateKalman_build_mex.m` script in the `matlab` directory. To use the Java implementation, build the Java library using the instructions in Section 4, and add both the resulting library (a `jar` file) and the library containing the Apache Commons Math library to MATLAB's Java search path using MATLAB `javaaddpath` command. The function `replication.m` in the `examples` sub-directory adds these libraries to the path and can serve as an example.

3.1 Options

The MATLAB implementations interpret a number of options, specified as fields of the `options` structure. Most of the implementations process only a subset of the options (some process none), as shown in Table 1.

The meaning of the options is as follows:

- The `estimateCovariance` field tells some of the algorithms that the covariance matrices of the state estimates are not required. Not computing them saves time in these algorithms. This is particularly useful in smoothers that are used as a building block of a non-linear smoother [12].

field name	type	values (default is 1st)	KalmanUltimate	KalmanConventional	KalmanOddevenSmoother	KalmanAssociativeSmoother	KalmanSparse	KalmanNative	KalmanJava
estimateCovariance	boolean	true, false	✓		✓				
covarianceEstimates	string	'PaigeSaunders', 'SelInv'	✓						
smoothOnly	boolean	false, true					✓		
algorithm	string	'Ultimate', 'Conventional', 'Oddeven', 'Associative'						✓	

Table 1: Options that affect the behavior of the MATLAB implementations.

- The `covarianceEstimates` field specifies which method to use to compute the covariance matrices of the state estimates, when more than one method is available.
- The `smoothOnly` field tells an implementation that filtered estimates are not required, only smoothed estimates. This can save time in some implementations (most notably, the `KalmanSparse` one).
- The `algorithm` field tells the interface to the C implementations which algorithm to use.

3.2 Implementation Details

All the implementations extend an abstract class called `KalmanBase`, which implements most of the methods that handle modifications of the sequence of steps (forgetting, rolling back, etc.). The `KalmanOddevenSmoother` and the `KalmanAssociativeSmoother` implementations do not return filtered estimated, only smoothed estimates. They both extend a second abstract class, `KalmanExplicitRepresentation`, in which the `evolve` and `observe` methods simply record the equations for later processing by the smoother. Both of these implementations are sequential but they exhibit the parallel algorithms in a clear way that is hopefully easy to implement in parallel programming environments.

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`.

4.1 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 an example program, `Rotation.java`. It performs the same computations as the MATLAB example function `rotation.m` when executed with arguments `rotation(UltimateKalman,5,2)`. This program serves as an example that shows how to write Java code that calls UltimateKalman.

5 The C programming interface

In the C interface, defined in `kalman.h`, 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` or `kalman_create_options`, which returns a pointer to `kalman_t`. The interface to the parallel smoothers is a little different and will be explained later.

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 the functions.

```
kalman_t*        kalman_create         ();
kalman_t*        kalman_create_options (kalman_options_t options);
void             kalman_free           (kalman_t* kalman);

void             kalman_evolve    (kalman_t* kalman, int32_t n_i,
                                   kalman_matrix_t* H_i,
```

```
                                kalman_matrix_t* F_i, kalman_matrix_t* c_i,
                                kalman_matrix_t* K_i, char K_i_type);
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);
int64_t         kalman_latest   (kalman_t* kalman);
void            kalman_forget   (kalman_t* kalman, int64_t i);
void            kalman_rollback (kalman_t* kalman, int64_t i);

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);
char            kalman_covariance_type (kalman_t* kalman, int64_t i);

kalman_matrix_t* kalman_perftest       (kalman_t* kalman, ...); // see later
```

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 $C = (W^T W)^{-1}$ or C itself, where C is the covariance matrix of the output of `kalman_estimate` on the same step. The function `kalman_covariance_type` specifies whether C is represented by itself or by its inverse factor W.

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 (the full set is declared in `matrix_ops.h`, included automatically by `kalman.h`).

```
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.

5.1 Options

The C takes an integer-type options argument that specifies which algorithm to use, as well as parameters for these algorithms (currently only one parameter is supported). Different options are ORed together. The option values defined in `kalman.h` are:

- `KALMAN_ALGORITHM_ULTIMATE`, `KALMAN_ALGORITHM_CONVENTIONAL`, `KALMAN_ALGORITHM_ODDEVEN`, and `KALMAN_ALGORITHM_ASSOCIATIVE`, which specify which algorithm to use. Specify exactly one of these bit values.

- KALMAN_NO_COVARIANCE, which tells the UltimateKalman and the Oddeven algorithms not to compute the covariance matrices of the estimates.

5.2 Direct Invocation of the Parallel Smoothers

The two parallel smoothers can be invoked for testing by first supplying the evolution and observation equations using a sequence of calls to kalman_evolve and kalman_observe and then calling kalman_smooth, but this normally not appropriate for high-performance parallel software, because the construction of the input to the smoother is completely sequential.

Instead, the parallel smoothers should be invoked directly on an array of equations,

```
void kalman_smooth_oddeven(kalman_options_t         options,
                           kalman_step_equations_t** equations,
                           kalman_step_index_t       k);
void kalman_smoother_associative(...); // same arguments
```

The first argument contains options, as in the call to kalman_create_options. The second, equations, is an array of k pointers to structures that each define an evolution equation and an observation equation, defined in kalman.h:

```
typedef struct kalman_step_equations_st {
  kalman_step_index_t step; // logical step number, start from 0
  int32_t dimension;

  kalman_matrix_t* H;
  kalman_matrix_t* F;
  kalman_matrix_t* c;
  kalman_matrix_t* K;
  char             K_type;

  kalman_matrix_t* G;
  kalman_matrix_t* o;
  kalman_matrix_t* C;
  char             C_type;

  kalman_matrix_t* state;
  kalman_matrix_t* covariance;
  char             covariance_type;
} kalman_step_equations_t;
```

Parallel codes should normally allocate two arrays of length k, an arrays of structures and an array of pointers to these structures, and ideally fill their elements using a multi-threaded code. The array of pointers should be filled with the addresses of the elements in the array of structures. Each structure should be filled with the matrices that define the evolution and observation equations of one step; the semantics are the same as in calls to kalman_evolve and kalman_observe. The dimension field specifies the dimension of the state vector, and the step field is a sequence number that should start from zero.

When the call to the smoother returns, the state fields contain the smoothed

state estimates and the `covariance` and `covariance_type` fields contain the covariance of the state, unless the `KALMAN_NO_COVARIANCE` bit was set in `options`.

The parallel-programming environment that the library uses, TBB, allows codes to control the number of operating-system threads that are used in a given computation, and a parameter called the block size. The first parameter controls how many physical cores are used. The second controls the overhead of parallelism: in parallel for loops and similar constructs, the environment performs blocks of iterations sequentially to reduce overhead. Our experiments indicate that a value of 16 is a good for the block size. Callers should specify these two value, using the following two functions defined in `parallel.h`, before calling a parallel smoother:

```
void parallel_set_thread_limit(int p);
void parallel_set_blocksize   (int b);
```

5.3 Building and Running the C Codes Outside MATLAB

To help you integrate UltimateKalman into your own native code in C or C++, the software includes several example programs that call UltimateKalman and Windows and Linux/MacOS scripts that build executable versions of the three programs. You should be able to use these scripts as examples of how to compile and link UltimateKalman.

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

Building Under Windows

The Windows build script `build.bat` uses the BLAS and LAPACK libraries from MKL [6]. Both MKL and TBB are installed as part of the oneAPI Base Kit, a set of free development tools from Intel. The kit includes additional components but for UltimateKalman, only thse components are required. We tested the library with the 2024.1 and 2025.0 versions of oneAPI, as well as some older ones. Upgrading oneMKL should not require any change in the build script.

The script also uses Microsoft's C/C++ compiler (`cl`) from the free Visual Studio Community Edition. We tested the library with versions 2019 and 2022 [8]. The build script is currently set up to use the 2022 version. If you installed a different version, you will need to update the path to the script that sets up the development tool in `build.bat`.

Assuming that these prerequisites have been installed, you can build the example programs by executing `build.bat`. The script will build the programs and it will tell you to execute another script so that Windows can find the MKL libraries at run time.

```
C:\Users\stoledo\github\ultimate-kalman\c>build.bat
**********************************************************************
** 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\"
   Visual Studio command-line environment initialized for: 'x64'
:  advisor -- latest
:  compiler -- latest
:  dev-utilities -- latest
:  ipp -- latest
:  mkl -- latest
:  ocloc -- latest
:  pti -- latest
:  tbb -- latest
:  umf -- latest
:: oneAPI environment initialized ::
kalman_ultimate.c
kalman_conventional.c
kalman_oddeven_smoother.c
kalman_associative_smoother.c
...
generated test programs
To run the generated binaries, invoke
  "C:\Program Files (x86)\Intel\oneAPI\setvars.bat" intel64
on the command line, to ensure that Windows can find the required DLLs.
```

You can see in the output the setup of Visual Studio, then the setup of oneAPI, and then the compilation of the source files. Once you run `setvars.bat` so that Windows can find the libraries, you can run the test programs. The output of `blastest.exe`, the simplest example program, should look like this:

```
C:\Users\stoledo\github\ultimate-kalman\c>blastest
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
```

Building Under Linux

Under Linux, `build.sh` is set up to use high-performance BLAS and LAPACK libraries, normally libraries provides by the CPU vendor:

- On Intel CPUs, the script uses both MKL [6] and TBB from Intel's free oneAPI base development kit. Install using the instructions on Intel's web site. The

script contains a variable that must be set up to the path where oneAPI was installed. This path is normally `/opt/intel/oneAPI`, but if you installed it elsewhere, you will need to update this path in the script.

- On AMD CPUs, the script uses the BLAS and LAPACK from the AMD performance libraries, which you can download freely from the AMD web site. Again there is a path that might need to be modified. You will also need to install TBB separately. On Ubuntu Linux, you can install it using the command `sudo apt install libtbb-dev`.

- On ARM processors, the script uses the BLAS and LAPACK from the ARM performance libraries, which you can download freely from the ARM web site. Again there is a path that might need to be modified. You will also need to install TBB separately. On Ubuntu Linux, you can install it using the command `sudo apt install libtbb-dev`.

In principle, for testing you can also use the BLAS and LAPACK libraries that the Linux distribution provides. On Ubuntu, you can install them using the commands

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

For small state dimensions these libraries might perform similarly to the vendor's high-performance libraries, but for large dimensions the CPU-vendor's libraries typically perform better.

Building Under MacOS

The `build.sh` script assumes that on MacOS, XCode is installed (Apple's development environment), which provides a C/C++ compiler. The script also assumes that TBB has been installed using Homebrew, a popular package manager for MacOS. The compiler is invoked in the script using the `gcc` command, but the actual compiler is not the GNU C compiler (`gcc`) but `clang`. The BLAS and LAPACK libraries that we use in the MacOS build are from Apple's Accelerate framework, a collection of high-performance libraries.

To install TBB using Homebrew, first install Homebrew (if not already installed) and then issue the commands `brew update` and `brew install tbb`. To inspect the installed version, use the command `brew info tbb`.

Running Example Programs

The build script builds two binaries from each of the four example source files, a sequential binary and a multi-threaded binary compiled using TBB. The multi-threaded binaries have names that end with `_par`, such as `performance_par`. The examples are:

- `blastest`, a simple program that only tests that UltimateKalman is able to call BLAS and LAPACK functions correctly. If this program fails to compile, to link, or to run, there is a problem with the BLAS and LAPACK libraries.

- `rotation` performs the same computation that the expression `rotation(KalmanUltimate,5,` performs in the MATLAB version and should output the same numerical results, just like `Rotation.java`. This example shows how to use UltimateKalman to filter and smooth, and it shows how the APIs in the three languages are related to each other.
  The program accepts up to 3 parameters, specified on the command-line as key-value pairs: `nthreads`, `blocksize`, and `algorithm`. The first takes an integer value (e.g., `nthreads=4`) and limits TBB to use this number of operating-system threads, and hence this number of cores. If you omit this parameter, TBB will use all the cores available. The second controls TBB's block size, the number of elements that are processed sequentially in parallel loops. The default value is 16. The third tells the program which algorithm to use, with possible values `Ultimate`, `Conventional`, `Oddeven`, and `Associative`.

- `performance` implements a Kalman smoother on problems with $n_i = m_i = n$ and k steps. The matrices F_i and G_i are orthonormal and $H_i = I$. These choices avoid the risk of numerical problems, including overflows or underflows. The program measures and reports the running time it takes to set up the problem and to perform filtering (unless the algorithm selected is a smoother), and the time it takes to perform smoothing. You specify n and k on the command line. Setting $n = 6$ or $n = 48$ results in exactly the same numerical input as the MATLAB version (with seed 1 for the random-number generator).
  The program accepts all the parameters that `rotation` accepts, but also `n`, `k`, and `nocov`. The last is a binary parameter that tells the program, if set, to not compute the covariance matrices of the state estimates.

Modifying Build Parameters

Certain aspects of UltimateKalman can be modified by defining preprocessor macros, listed in Table 2:

- The BLAS and LAPACK were originally Fortran libraries and different implementations of them must be called from C in different ways. Several preprocessor macros select the calling convention. The build scripts set these up for several popular libraries.

- Step numbers (indices of states) can be represented by 32-bit or 64-bit numbers. Three preprocessor macros, which should be set consistently, control this choice: `KALMAN_STEP_INDEX_TYPE_INT32`, `FARRAY_INDEX_TYPE_INT32`, and `PARALLEL_INDEX_TYPE_INT32` (to use 64-bit integers, replace `INT32` by `INT64`; it is also possible to use unsigned integer typed, but we advise against this).

- Two preprocessor macros listed at the bottom of the table control debug outputs. They should normally not be set.

6 MATLAB Tests and Code Examples

The software distribution of UltimateKalman includes several MATLAB example and testing functions, stored in the `examples` sub-directory. They demonstrate how

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
KALMAN_STEP_INDEX_TYPE_INT32	exactly one	Specifies 32-bit step numbers
KALMAN_STEP_INDEX_TYPE_INT64	exactly one	Speficies 64-bit step numbers
FARRAY_INDEX_TYPE_INT32/64	yes	Must be consistent with KALMAN_STEP_INDEX_TYPE
PARALLEL_INDEX_TYPE_INT32/64	yes	Must be consistent with KALMAN_STEP_INDEX_TYPE
BUILD_DEBUG_PRINTOUTS	no	generates run-time debug printouts
NDEBUG	no	suppresses run-time assertion checking

Table 2: Preprocessor variables that control how UltimateKalman calls the BLAS and LAPACK, as well as several other aspects of its behavior.

to use the library and help test it. The directory also contains a function that builds the MATLAB-callable version of the C libray. All of these MATLAB files are documented with comments that can be accessed through the MATLAB `help` command (e.g., `help rotation`). Not all the examples work with all the Kalman filtering and smoothing algorithms.

The script `UltimateKalman_build_mex` builds the MATLAB-callable version of the C (native) library. The produced native library is a file in a format called a mex file.

The examples take as input a Kalman filter factory function, which allows them to instantiate one or more instances of the same algorithm. You construct these factory functions by calling `kalmanFactory` function, which accepts the name of a class and optionally a structure with options for its constructor. For example:

```
ultimateFactory = kalmanFactory('UltimateKalman');
nativeFactory   = kalmanFactory('KalmanNative',struct('algorithm','Conventional'));
rotation( ultimateFactory, 5, 2 );
rotation( nativeFactory,   5, 2 );
```

The examples are:

- `rotation.m`, modeling a rotating point in the plane (this example is also implemented in C and Java, as described above). This example should work with all the Kalman algorithms, but in implementations that only smooth (do not filter), it will obviously not produce filtered stated.

- `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. This example does not work with the conventional Kalman filter-smoother and does not work with the associative smoother, which both do not support non-identity H_is.

- `projectile.m`, implementing the model and filter of a projectile described by Humpherys et al. [5].

- `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 and it too depends on non-identity H_is.

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

The file `performance.m` contains a function that tests and plots the performance of Kalman filters and smoothers. Its behavior is described in the next section.

The file `compare.m` contains a function that runs two Kalman filters and/or smoothers on the same synthetic problem and compares their result. Its main function is to test the correctness of one algorithm or implementation to another known to be correct. It does not currently provide a complete coverage of all the code in all the implementation, but it is still very effective in detecting bugs. The gold standard for the known-correct filter/smoother is the `KalmanSparse` implementation, which is the simplest so its correctness is easiest to validate analytically,

but it is slow, especially when filtering. A reasonable strategy is to use it to validate the correctness of another filter/smoother once, say `KalmanUltimate`, and then to use this filter/smoother to test the other implementations and algorithms. The main current limitation of this function is that it always uses $H_i = I$.

The script `replication.m`, also in the `examples` sub-directory, runs all of these examples and optionally generates the figures shown in Section 5 of the article on UltimateKalman [13]. Comparing the generated figures to those in the article provides visual evidence that the code runs correctly. 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 library that MATLAB uses is built using the `UltimateKalman_build_mex.m` script, as explained in Section 5 above. The Java version should be built outside MATLAB, as explained in Section 4. To run the script to generate the graphs inn [13], run it with no arguments or with the arguments

```
replication(kalmanFactory('KalmanUltimate'), ...
    false, ...
    { kalmanFactory('KalmanUltimate'), ...
      kalmanFactory('KalmanJava'),     ...
      kalmanFactory('KalmanNative')    ...
    });
```

To test another implementation, run it with another factory as the first argument, as in

```
replication(kalmanFactory('KalmanNative',struct('algorithm','Ultimate')));
```

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.

The function `performance` uses the `perftest` method to measure and plot the performance of one or more filters/smoothers. You invoke it on a cell array of factories and provide several parameters: a seed for the random-number generator, an array of state dimensions to test, the number of state in each test, and the number of time steps between timestamps. Graphs produced by this function are presented in [13].

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 MATLAB 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, Jeremy 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):18–28, 1990. doi:10.1145/77626.77627.

[3] J. J. Dongarra, Jeremy Du Cruz, Sven Hammarling, and I. S. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16(1):1–17, 1990. doi:10.1145/77626.79170.

[4] Shahaf Gargir and Sivan Toledo. Parallel-in-time Kalman smoothing using orthogonal transformations. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2025. to appear, also available as https://arxiv.org/abs/2502.11686.

[5] 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.

[6] 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.

[7] R. E. Kalman. A new approach to linear filtering and prediction problems. *Journal of Basic Engineering*, 82(1):35–45, 1960. doi:10.1115/1.3662552.

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

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

[10] H. E. Rauch, F. Tung, and C. T. Striebel. Maximum likelihood estimates of linear dynamic systems. *AIAA Journal*, 3(8):1445–1450, 1965. doi:10.2514/3.3166.

[11] Simo Särkkä and Ángel F. García-Fernández. Temporal parallelization of Bayesian smoothers. *IEEE Transactions on Automatic Control*, 66(1):299–306, 2021. doi:10.1109/TAC.2020.2976316.

[12] Simo Särkkä and Lennart Svensson. Levenberg-Marquardt and line-search extended Kalman smoothers. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5875–5879, 2020. doi:10.1109/ICASSP40776.2020.9054686.

[13] Sivan Toledo. Algorithm 1051: UltimateKalman, flexible Kalman filtering and smoothing using orthogonal transformations. *ACM Transactions on Mathematical Software*, 50(4):1–19, 2024. doi:10.1145/3699958.