Parallelization of the Kalman filter for multi-output systems on multicore platforms

Per Jonsson
Abstract

Parallelization of the Kalman filter for multi-output systems on multicore platforms

Per Jonsson

The Kalman filter is a very commonly used signal processing tool for estimating state variables from noisy observations in linear systems. Because of its cubic complexity, it is motivated to search for more computationally efficient Kalman filter implementations.

In this thesis work, previous attempts of parallelizing the Kalman filter have been investigated to determine whether any of them could run efficiently on modern multi-core computers. Two of the most interesting methods from a multi-core perspective have undergone further analysis to study how they perform in a multi-core environment. In the analysis, both state estimation accuracy and algorithm speedup have been considered.

The experiment results indicate that one of the evaluated algorithms, denoted the Fusion Gain method in this report, is faster on a quad-core CPU than a straight-forward implementation of the original Kalman filter when the number of output signals is large. It should be noted, however, that this algorithm is not identical to the true Kalman filter due to an approximation used in the derivation of the method. Despite this detail, it might still be of use in some applications where speed is more important than accurate state estimates.

The other evaluated method is based upon a fast Givens rotation. It was originally implemented on a so-called systolic array, which makes use of parallelism differently than multi-core computers. Unfortunately, this algorithm turned out to run very slow in the benchmarks even though the number of floating-point operations per second (FLOPS) should be far less than many of the other methods according to the theoretical analysis. More attention could be devoted to this implementation to locate possible bottlenecks.
Contents

1 Introduction ........................................... 5
   1.1 Background ........................................... 5
   1.2 The Thesis Work ..................................... 6

2 Theory .................................................. 6
   2.1 The Kalman Filter .................................. 6
   2.2 Multi-Core Computers ................................ 8

3 Existing Parallelization Approaches ..................... 10
   3.1 Full-Order Distributed Kalman Filters ............... 10
       3.1.1 A Globally Optimal Method ..................... 11
       3.1.2 A Globally Sub-Optimal Method ................. 13
   3.2 Reduced-Order Distributed Kalman Filters .......... 15
   3.3 The Kalman Filter on Systolic Arrays ............... 17
       3.3.1 SIMD Arrays .................................. 18
       3.3.2 Warp ......................................... 20
       3.3.3 Decoupling of Time-Update and Measurement-Update Equations ............ 21

4 Choice of System ...................................... 22
   4.1 A Temperature Bar .................................... 23

5 Simulations ............................................ 24
   5.1 Setup ............................................... 24
   5.2 Experiments and Results ............................. 24

6 Algorithm Analysis ................................... 28
   6.1 Fusion-Gain ......................................... 28
   6.2 Fast-Givens Rotation ................................. 30
   6.3 Summary ............................................. 36

7 Implementation ........................................ 38
   7.1 Introduction ......................................... 38
   7.2 The Algorithms ...................................... 38
       7.2.1 Fusion-Gain ................................... 38
       7.2.2 Fast Givens Rotation ............................ 40
   7.3 The Benchmark Software .............................. 40
8 Preparations for the Final Run

8.1 Setting up the Test Cases ........................................... 41
8.2 Computer Environment ............................................. 41

9 Final Results

9.1 Fusion Gain ................................................................. 42
9.2 Fast Givens ................................................................. 44
9.3 Experimental versus Predicted Results .......................... 44
9.4 Intel MKL with Thread Support Enabled ......................... 48

10 Discussion

10.1 Conclusions ............................................................... 50
10.2 Future Work .............................................................. 51
1 Introduction

1.1 Background

The Kalman Filter  In many technical applications there is a need to determine the state of some system of interest. For instance, in navigation or tracking applications it might be desirable to know the position and the speed of a vehicle. This can be accomplished by constructing a model of the system in terms of differential equations. The model reflects the dynamics of the system and how the inner state variables relate to the sensor measurements and the input to the system. Given this model, a well-known and widely used algorithm exists that fuses this information together to yield a state estimate, namely the Kalman filter.

The Kalman filter was invented by Rudolf E. Kálmán in 1960 (Kalman, 1960). It has since then been widely adopted, not only in technical applications, but also in other fields such as economics. The first known implementation of the filter was in the navigation system used by the lunar module in the Apollo 11 space mission, carried out by NASA in 1969. During the time when that project took place, many other techniques were also invented (Mcgee, 1985), e.g. the Extended Kalman filter, which allowed nonlinear models to be used, and the Square-root implementation which greatly improved the numerical stability of the implemented algorithm.

The filter consists of two parts. The first is the time-update step, in which the state vector is propagated one discrete time-step using a model of the true system. The second part is the measurement-update step, which involves solving a Riccati equation and fusing its solution with measurement data to improve the time-update prediction. The computational complexity of these steps is approximately of the order of $O(n_x^3)$ and $O(n_z^3)$ respectively, where $n_x$ denotes the number of state variables and $n_z$ is the number of measured output signals. Inevitably, the algorithm is quite demanding when it is applied to large systems and could therefore be hard to apply, especially in real-time filtering applications.

The Entry of Multi-Core Computers  Traditionally all central processing units (CPUs) on the consumer market have only had one core. Due to a number of factors (mainly related to issues with higher clock frequencies) it has become hard to achieve any significant performance gain using only a single core. The remedy was to incorporate additional CPU cores on the same die, which makes it possible to execute program code on multiple cores simultaneously. However, just adding an extra core does not automatically lead to a performance boost. To make the most of the multi-core architecture, the software has to be written in such way that it executes sections of code in parallel. How easy this is to achieve depends on the specific situation. In a desktop environment it might be sufficient to run just a small percentage of the work in parallel to make the computer seem more responsive. On the other hand, in the case with signal processing, it might be highly desirable to keep each CPU core fully utilized at all times.

The trend is that multi-core CPUs enter more and more areas and that the number of cores is increasing. Nowadays quad-core CPUs can even be found in handheld smart phone devices.

Previous Parallelization Attempts  Since the time the Kalman filter was invented, various attempts have been made to enhance its computational performance. One of the first attempts dates back to 1976 when (Willner/Chang/Dunn, 1976) proposed a method for computing the Kalman filter when the sensors observing the system are located at different sites. The idea was to process each sensor reading locally. If this was done simultaneously at each sensor computation time could hopefully be saved. Papers along the same path include (Chong, 1979; Hashemipour/Roy/Laub, 1988; Carlson, 1990; Paik/Oh, 2000). This category of Kalman filter parallelization techniques are referred to as “Distributed Kalman filter” and are covered in Section 3.1 and 3.2.  

The other big category of parallel approaches are Kalman filters intended to run on so-called systolic arrays (Kung/Leiserson/of Computer Science, 1978). A systolic array is a signal processing resource
consisting of a number (often large) of simple processing elements connected together in a mesh-like structure. Data then “flows” through the array, one processing element at a time. This design turned out to be computationally efficient when performing common signal processing tasks, such as convolutions and matrix multiplications. Attempts have been made to come up with Kalman filter implementations that could run efficiently on systolic arrays (Gaston/Irwin, 1990). Many of them use some form of square-root implementation. Some of them use Faddeev’s algorithm to perform the matrix operations, others use fast Givens rotations (Itzkowitz/Baheti, 1989; Palis/Krecker, 1990) and some decouples the time-update equation from the measurement-update equation to process those simultaneously (Youmin et al., 1993).

1.2 The Thesis Work

The purpose of this thesis work is to investigate how ideas to parallelize the Kalman filter done in the past (1960-) can be made to work on a modern multi-core architecture.

The strategy is to divide the work into smaller pieces. These are,

- Search for, and study publications done in the past to find out what approaches that are available. Summarize the findings. (Section 3)
- Decide what system to use for the simulations and benchmarks. (Section 4)
- Pick out a few methods to put under test. (Section 5)
- Simulate the system in Matlab and estimate the state using each of the picked out methods. (Section 5)
- Make reflections based on the result and possibly discard one or more of the methods that were not good enough. (Section 5)
- Analyze the computation requirement of the remaining methods. (Section 6)
- Implement the methods in the C programming language and make them run on the supercomputers at UPPMAX. (Section 7)
- Set up and run the final benchmarks at UPPMAX. Then present and comment on the result. (Section 9)
- Compare experimental performance with the theoretical predictions. (Section 9)
- Discuss the outcome of the entire thesis work. How did it go? What would be the next steps? (Section 10)

2 Theory

2.1 The Kalman Filter

The System Model

Consider a model of a discrete dynamical system written on state-space form,

\[
x(k + 1) = F(k)x(k) + \Gamma(k)u(k) + G(k)w(k) \\
z(k) = H(k)x(k) + v(k)
\]  

(2.1)
(2.2)
In this model the upper equation (2.1) describes the dynamics of the system expressed in terms of the state variables contained in the state vector $x(k)$. If the model would be an exact description of the true system the state transition matrix $F(k)$ would completely describe the dynamics of the system. However, in most real-life applications this is not the case. This is why it is necessary to introduce the $w(k)$ term, which is often referred to as the process noise. $w(k)$ is a stochastic variable modeled as zero mean white Gaussian noise with the co-variance matrix $Q(k)$.

The lower equation (2.2) is the so-called observation equation which tells us how the output vector $z(k)$ measurements are related to the internal state vector $x(k)$. These readings are often affected by a measurement noise $v(k)$, which is here assumed to be zero mean white Gaussian noise with the co-variance matrix $R(k)$.

All this can be expressed in mathematical notation as,

$$
\begin{align*}
\dot{x}(k) & \in \mathbb{R}^{n_x} \\
u(k) & \in \mathbb{R}^{n_u} \\
z(k) & \in \mathbb{R}^{n_z} \\
w(k) & \in \mathbb{R}^{n_w} \\
v(k) & \in \mathbb{R}^{n_z} \\
F(k) & \in \mathbb{R}^{n_x \times n_x} \\
\Gamma(k) & \in \mathbb{R}^{n_x \times n_u} \\
G(k) & \in \mathbb{R}^{n_x \times n_w} \\
H(k) & \in \mathbb{R}^{n_z \times n_x} \\
w(k) & \sim \mathcal{N}(0,Q(k)) \\
v(k) & \sim \mathcal{N}(0,R(k)) \\
Q(k) & \in \mathbb{R}^{n_w \times n_w} \\
R(k) & \in \mathbb{R}^{n_z \times n_z}
\end{align*}
$$

The Discrete Kalman Filter  
What is needed to get hold of the state vector $x(k)$ is an observer, such as the Kalman filter. It provides an optimal way of estimating the state variables of linear dynamical systems affected by white noise ($w(k)$ and $v(k)$). An observer can be written as,

$$
\begin{align*}
\dot{x}(k+1|k) & = F(k)\dot{x}(k|k-1) + \Gamma(k)u(k) + K(k)v(k) \\
v(k) & = z(k) - H(k)\dot{x}(k|k-1)
\end{align*}
$$

Here $v(k)$ is a measure of how well the model prediction corresponds to the measurement $z(k)$, and is called the innovation or the residual. $\hat{x}(q|r)$ is the actual state estimate at time instant $q$ based on information from time sample 1 to $r$. The factor $K(k)$ that minimizes the state estimate error covariance matrix $P(k)$ is called the Kalman gain and can be expressed as,

$$
K(k) = P(k|k-1)H^T(k) \left[ H(k)P(k|k-1)H^T(k) + R(k) \right]^{-1}
$$

$P(k|k-1)$ denotes the so-called a priori state estimate error covariance for the a priori state estimate vector $x(k|k-1)$. $k - 1$ indicates that $P(k|k-1)$ and $x(k|k-1)$ are predictions based only on information available up to time sample $k-1$. This is in contrast to the a posteriori estimates $P(k|k)$ and $\dot{x}(k|k)$ which represent $P(k)$ and $x(k)$ when information from time sample $k$ is used as well.

$$
\begin{align*}
P(k|k-1) & = E[\dot{x}(k|k-1) \cdot \dot{x}(k|k-1)^T] \\
P(k) & = E[\dot{x}(k|k) \cdot \dot{x}(k|k)^T] \\
\dot{x}(k|k) & = x(k) - \dot{x}(k|k) \\
\dot{x}(k|k-1) & = x(k) - \dot{x}(k|k-1)
\end{align*}
$$
The discrete Kalman filtering algorithm can be separated into two parts, namely the time-update step and the measurement-update step. The time-update step is where the one-step prediction occurs, which tries to compute an \textit{a priori} state estimate of the system’s state vector, one step ahead in time. In this step information about the dynamics of the model as described in (2.1) is used.

The second step is the measurement-update step in which the \textit{a priori} estimate is “corrected” using information from the sensor measurements.

The complete algorithm is presented in Algorithm 1.

\textbf{Algorithm 1} The Kalman filter

\textbf{Initialization:}
\[ \hat{x}(-1|1) = \hat{x}_{\text{initial guess}} \]
\[ \hat{P}(-1|1) = \hat{P}_{\text{initial guess}} \]

\textbf{Time-update (a priori estimation):}
\[ \hat{x}(k|k-1) = F(k-1)\hat{x}(k-1|k-1) + \Gamma(k-1)u(k-1) \]
\[ P(k|k-1) = F(k-1)P(k-1|k-1)F^T(k-1) + G(k-1)Q(k-1)G^T(k-1) \]

\textbf{Measurement-update (a posteriori estimation):}
\[ K(k) = P(k|k-1)H^T(k) \left[H(k)P(k|k-1)H^T(k) + R(k)\right]^{-1} \]
\[ \hat{x}(k) = \hat{x}(k|k-1) + K(k)(z(k) - H(k)\hat{x}(k|k-1)) \]
\[ P(k) = [I - K(k)H(k)]P(k|k-1) \]

\textbf{Numerical Stability} One problem with this straight-forward implementation (Algorithm 1) is the risk of numerical instability, which is prone to make the state error covariance matrix $P$ non-positive-definite, and thus violating the Kalman filter requirement of $P$ being positive-definite (Gustafsson, 2000). This phenomena often occurs when the process error $Q(k)$ is small. Other problems that may have bad influence on the numerical properties are e.g. the risk of $P$ becoming asymmetric or loosing numerical accuracy when the state variables are not scaled properly.

A way to overcome these issues is to work with a decomposition of the state error covariance $P$ instead of $P$ itself. For instance, $P$ could be decomposed as $P = SS^T$, where $S$ is a matrix square root of $P$. This assures that $P$ will not become non-positive-definite. To implement these ideas decomposition techniques such as QR or LDL\textsuperscript{T} are commonly used.

However, in this report Algorithm 1 is used when referring to the “Standard Kalman Filter” implementation.

\subsection*{2.2 Multi-Core Computers}

Parallel computing is about running two or more pieces of machine code simultaneously. The concept is not new in any way, but has been used for decades in various forms. In recent years more attention has been devoted to the field of parallel computing (Faxén et al., 2008). The main cause for this is the difficulties in gaining increased performance by increasing the CPU clock frequency. To overcome this performance issue, CPU manufacturers have begun to implement different parallel strategies into their CPUs. One strategy is the Instruction Level Parallelism (ILP), where (independent) machine instructions occurring close to each other in the program are run in parallel. This lowers the average time spent waiting for the CPU core when cache-misses occur.
Another approach is to put more than one core on the same chip, which is what multicore computing is all about. At this point there are basically two different paths to choose from. The first one is to have a large number of quite primitive cores. This choice is often most suitable for problems where one expects the performance of the algorithm to grow with the number of cores, and is sometimes referred to as manycore. The second approach is to place a couple of standard CPU cores on the same chip and connect them together. We will concentrate on this type of platform, because it is was what we have at our disposal for this thesis work.

The latter types of platforms are seen in the multicore CPUs found in today’s desktop and laptop computers, and recently even in smartphones. Similar kinds of CPUs are also found in the computation network SNIC-UPPMAX (“Uppsala Multidisciplinary Center for Advanced Computational Science”). One example is the system Kalkyl at UPPMAX which consists of 348 computer nodes, where each node has two Intel Xeon Quad-core 5520 CPUs. Figure 2.1 shows the cache configuration and the memory sizes.

Programs that make use of the multicore architecture often use the OpenMP API to specify what parts of the program should be run in parallel. OpenMP is fairly easy to use once you know how the program could be parallelized. Concluding how this could be made is the hard part.

There are a number of issues one should be aware of to succeed in the parallelization process. The goal when programming for multicores is to keep all the cores fully utilized. Things that will prevent you from accomplishing this are mainly slow memory, too much overhead and programming errors such as deadlocks.

**Cache** The CPU cores operate at a much higher rate than the RAM memory. This is why the cache memories are so important. The cache memory is a memory placed between the CPU and the RAM memory and acts as a fast, but small, buffer for the RAM. Without the cache memory the CPU core would spend unreasonable amount of time waiting for data to be read (written) from (to) the RAM memory.

On an ordinary single-core CPU there are most often one to three levels of cache memories between the CPU and the RAM memory. The L1 cache is the fastest, but smallest, cache memory and sits closest to the CPU. The L3 cache, on the other hand, is the slowest, but largest cache memory.

On multi-core CPUs some of the cache memories are shared and some are not. A common cache memory arrangement could e.g. be as depicted in Figure 2.1, where each core has its own L2 cache.

What makes it into the cache usually follows the principles behind temporal locality and spatial locality, i.e. data that has recently been accessed is likely to be used again very soon and the neighboring data as well. This also holds for single-core machines. What gets complicated in the multi-core case is to
maintain consistency among the individual caches. If one core alters the content of a memory location the rest of the caches holding the same bit of memory need to be updated to reflect the new state. Of course, all this is handled by the hardware, but it is nevertheless good to be aware of in order to write efficient code.

**Overhead** When attempting to parallelize an algorithm, the key is to find a section that is large enough to be worth parallelizing. The reason for this is that the overhead associated with entering and exiting a parallel section would otherwise become larger than the actual performance gain of parallelizing that piece of code in the first place. The overhead comes from the time it takes to split the current program flow into a series of new threads and then, when the threads are finished, synchronize everything again. This is also known as the fork-join model. Figure 2.2 illustrates how this works.

When the application is first started it is running in the master thread. Later, when the program execution encounters a parallel section one or more additional threads are spawned. The operating system can then choose to execute the threads in parallel on one or more available cores. When the threads finish, a synchronization step takes place, where it is made sure that the memory is consistent. During this phase all threads, except for the master thread, will terminate.

### 3 Existing Parallelization Approaches

The outcome of this section lays the basis for the remaining parts of the work. The plan has been to skim through a number of papers to get an idea of existing solutions to parallelizing the Kalman filter. While reading the articles short notes have been written down that summarize the papers.

Then, every note has been sorted so that similar methods get into the same category. These categories turned out to be,

- Distributed filters
- Filter used on large computer arrays
- Decoupling of time-update and measurement-update

#### 3.1 Full-Order Distributed Kalman Filters

The typical need for the distributed Kalman filter arises when several sensors, spread out geographically are used together to make observations on a common system. The sensor nodes are often connected to a central computer that computes the global estimate of the system using the information received from the sensor nodes.

Also fault detection and correction are integrated in these methods, which often implies that the central computer is able to get a state estimate even in situations where some of the local filter nodes for some
reason are unable to provide their local estimates to the global filter. Another possibility could be to exclude certain local state estimates from the global collating filter in cases where a sensor output reports estimates that are clearly erroneous.

This idea of distributed computing could most certainly be applicable to multicore computers if the sensors are not assumed to be spread out geographically, but rather be connected to the same machine. The local sensor processing can then be carried out on different cores on the same CPU instead of being computed at the spread out sensor nodes.

Over the years many different methods have been suggested that make use of the distributed approach, each with their own advantages and disadvantages. One of the first contributions was made in (Willner/Chang/Dunn, 1976). At each sensor node a local sequential Kalman filter is attempting to estimate the global state vector at each sample period using sensor data only from that sensor. However, the sensor nodes are not entirely autonomous. In addition to the variables required to run a standard Kalman filter it also requires the global estimate vector from the previous time step. When the local state estimation vector and the corresponding covariance matrix have been computed, the result is transmitted to the central processor. The central processor then combines the information received from its child nodes to yield the global state estimate (see Figure 3.1).

\[ x(k+1) = F(k)x(k) + \Gamma(k)u(k) + G(k)w(k) \]
\[ z_j(k) = H_j(k)x(k) + v_j(k) \]

3.1.1 A Globally Optimal Method

In the frequently cited paper (Hashemipour/Roy/Laub, 1988) a method similar to (Willner/Chang/Dunn, 1976) is proposed with the exception that the local filters do not need to know of the global estimate. This means that each sensor only has to communicate with the central processor over a one-way communication link (see Figure 3.2) which can be beneficial especially when communication is associated with significant performance costs.

\[ x(k+1) = F(k)x(k) + \Gamma(k)u(k) + G(k)w(k) \]
\[ z_j(k) = H_j(k)x(k) + v_j(k) \]
where,

\[
\begin{align*}
  z(k) &= [ z_1^T(k) \cdots z_N^T(k) ]^T \\
  H(k) &= [ H_1^T(k) \cdots H_N^T(k) ]^T \\
  v(k) &= [ v_1^T(k) \cdots v_N^T(k) ]^T \\
  v_j(k) &\in \mathbb{R}^{m_j}, \quad \sum_{j=1}^N m_j = m
\end{align*}
\]

Furthermore, it is assumed that there is no correlation between any two sensor measurement noises \( v_j(k) \) or between the measurement noises \( v_j(k) \) and the process noise \( w(k) \).

\[
E\{v_j(k)v_k(k)\} = 0, \quad j \neq k \\
E\{v(k)w(k)\} = 0 \\
E\{v(k)v^T(k)\} = \text{block diag}\{R_1, \cdots, R_N\} \\
E\{w(k)w^T(k)\} = Q
\]

The time-update and measurement-update equations to be computed at sensor node \( j \), will have the same form as a standard Kalman filter, see Algorithm 2.

**Algorithm 2** Local filter (full-order)

Initialization:

\[
\hat{x}_j(-1|1) = \hat{x}_{\text{initial guess}} \\
\hat{P}_j(-1|1) = \hat{P}_{\text{initial guess}}
\]

Local time-update:

\[
\begin{align*}
\hat{x}_j(k|k-1) &= F(k-1)\hat{x}_j(k-1|k-1) + \Gamma_j(k-1)u_j(k-1) \\
P_j(k|k-1) &= F(k-1)P_j(k-1|k-1)F^T(k-1) + G(k-1)Q(k-1)G^T(k-1) \\
\tilde{Q}_j(k-1) &= G(k-1)\left[ Q(k-1) - \frac{(N-1)}{N}Q(k-1) \right]G^T(k-1)
\end{align*}
\]

Local measurement-update:

\[
\begin{align*}
K_j(k) &= P_j(k|k-1)H_j^T(k)\left[ H_j(k)P_j(k|k-1)H_j^T(k) + R_j(k) \right]^{-1} \\
\tilde{x}_j(k|k) &= [I - K_j(k)H_j(k)]\hat{x}_j(k|k-1) + K_jz_j \\
P_j(k|k)^{-1} &= P_j(k|k-1)^{-1} + H_j^T(k)R_j^{-1}(k)H_j(k)
\end{align*}
\]

Through a number of substitutions (Hashemipour/Roy/Laub, 1988) the expressions for the global time-update and global measurement-update can finally be obtained as seen in Algorithm 3.
Algorithm 3 Global filter (full-order)

Initialization:
\[ \hat{x}(-1|1) = \hat{x}_{\text{initial guess}} \]
\[ \hat{P}(-1|1) = \hat{P}_{\text{initial guess}} \]

Global time-update:
\[
\begin{align*}
\hat{x}(k|k-1) &= F(k-1)\hat{x}(k-1|k-1) - (N-1)\Gamma(k-1)u(k-1) \\
P(k|k-1) &= F(k-1)P(k-1|k-1)F^T(k-1) + G(k-1)Q(k-1)G^T(k-1) = \\
&= F(k-1)P(k-1|k-1)F^T(k-1) + \sum_{j=1}^{N} \hat{Q}_j(k-1)
\end{align*}
\]

Global measurement-update:
\[
\begin{align*}
\hat{x}(k|k) &= P(k|k)\left( P(k|k-1)^{-1}\hat{x}(k|k-1) + \sum_{j=1}^{N} H_j^T(k)R_j^{-1}(k)z_j(k) \right) \\
&= P(k|k)\left[ P(k|k-1)^{-1}\hat{x}(k|k-1) + \\
&\quad \sum_{j=1}^{N} \left\{ P(k|k)_j^{-1}\hat{x}(k|k)_j - P_j(k|k-1)^{-1}\hat{x}(k|k-1) \right\} \right] \\
P(k|k)^{-1} &= P(k|k-1)^{-1} + \sum_{j=1}^{N} H_j^T(k)R_j^{-1}(k)H_j(k) = \\
&= P(k|k-1)^{-1} + \sum_{j=1}^{N} \left\{ P(k|k)_j^{-1} - P_j(k|k-1)^{-1} \right\}
\end{align*}
\]

Conclusions
This method could be a candidate for implementation on a multicore machine. Each of the \(N\) local filters could run on a separate core on a multicore computer. More than one filter could possibly be assigned to some of the cores if the number of cores is greater than \(N\).

3.1.2 A Globally Sub-Optimal Method

Based on the prior work of e.g. (Hashemipour/Roy/Laub, 1988) and (Carlson, 1990) a new, sub-optimal composition of the distributed Kalman filter was developed in (Paik/Oh, 2000). The principle behind the idea is to put an upper bound on the covariance of the residual of the global model, denoted \(S\) in the Kalman gain equation,
\[
\begin{align*}
K(k+1) &= P(k+1)H^T(k+1)S^{-1}(k+1) \\
S(k+1) &= H(k+1)P(k+1|k)H^T(k+1) + R(k+1)
\end{align*}
\]

It can be shown (e.g. (Carlson, 1990)) that \(S\) is upper-bounded by,
\[
S(k+1) \leq \text{block diag}\{\gamma_1 S_1(k+1), \cdots, \gamma_N S_N(k+1)\}
\]

where the local residuals are given by,
\[
S_j(k+1) = H_j(k+1)P_j(k+1|k)H_j^T(k+1) + (1/\gamma_j)R_j(k+1)
\]
and $\gamma_j \geq 1$ are weights (normally $\gamma_j = N$, as mentioned in (Carlson, 1990)) applied locally to allow fusion of the local results to yield the global estimate.

Then by substituting $S$ in (3.1) with the upper bound in (3.2), (3.1) can be parallelized through (3.3). This is also where the sub-optimality of this method is introduced. This makes it possible to write the Kalman gains as,

$$K(k+1) = P(k+1|k) \left[ \frac{1}{\gamma_1} H_1^T(k+1)S_1^{-1}(k+1), \ldots, \frac{1}{\gamma_N} H_N^T(k+1)S_N^{-1}(k+1) \right]$$

$$K_j(k+1) = \frac{1}{\gamma_j} P_j(k+1|k)H_j^T(k+1)S_j^{-1}(k+1)$$

The global error covariance ($P(k|k)$) and the global state estimate ($\hat{x}(k|k)$) are given as Step 4 in Algorithm 4. For expressing $P(k|k)$ the author has chosen to use the so-called “Joseph stabilized formula” to give a numerically more stable solution.

**Algorithm 4** A sub-optimal distributed variant of the Kalman filter

Initialization:

$$\hat{x}(-1|1) = \hat{x}_{\text{initial guess}}$$

$$\hat{P}(-1|1) = \hat{P}_{\text{initial guess}}$$

1. Perform Global Time-Update:

$$\hat{x}(k|k-1) = F(k-1)x(k-1|k-1) + \Gamma(k-1)u(k-1)$$

$$P(k|k-1) = F(k-1)P(k-1|k-1)F^T(k-1) + G(k-1)Q(k-1)G^T(k-1)$$

2. Assign the result to the local filters ($j = 1, \ldots, N$) as well:

$$\hat{x}_j(k|k-1) = \hat{x}(k|k-1)$$

$$P_j(k|k-1) = P(k|k-1)$$

3. On each local node compute the local Kalman gain and state estimates:

$$K_j(k) = \frac{1}{\gamma_j} P(k+1|k)H_j^T(k) \left[ H_j(k)P(k|k-1)H_j^T(k) + \frac{1}{\gamma_j} R_j(k) \right]^{-1}$$

$$\hat{x}_j(k|k) = \hat{x}(k|k-1) + K_j(k) [z_j(k) - H_j(k)\hat{x}(k|k-1)]$$

4. Perform the global fusion:

$$\hat{x}(k|k) = (1 - N)\hat{x}(k|k-1) + \sum_{j=1}^{N} \hat{x}_j(k|k)$$

$$P(k|k) = \left[ I - \sum_{j=1}^{N} K_j(k)H_j(k) \right] P(k|k-1) \left[ I - \sum_{j=1}^{N} K_j(k)H_j(k) \right]^T + \sum_{j=1}^{N} K_j(k)R_j(k)K_j^T(k)$$

**Conclusions** This is a sub-optimal filter, and hence not a “true” Kalman filter, with the advantage of possibly being more computationally efficient than the optimal variant. For instance, the absence
of computationally costly inversions in the global steps (Step 1 and Step 4 in Algorithm 4) might be favorable when being implemented on a multicore computer.

3.2 Reduced-Order Distributed Kalman Filters

In (Roy/Hashemi/Laub, 1991) a further developed variant of (Hashemipour/Roy/Laub, 1988) was published that uses local filters of reduced order. While the aim of this method is to achieve a globally optimal state estimation, reduction of order is not always possible for an arbitrary system. A requirement is that any state variable must not be part of more than one local filter. This implicates that the system matrix, $F$, needs to be block diagonal.

After performing the following derivation it turns out that the algorithm will look almost identical to the algorithm proposed in (Hashemipour/Roy/Laub, 1988) except that this method requires the extra “translation matrices” $D_j$ and $E_j$.

Again, consider the global system model,

$$
x(k+1) = F(k)x(k) + G(k)w(k)
$$

$$
z(k) = H(k)x(k) + v(k)
$$

Let $z$, $v$, $w$, $G$ and $H$ be partitioned into $N$ separate blocks as,

$$
z(k) = [z_1^T(k) \cdots z_N^T(k)]^T
$$

$$
w(k) = [w_1^T(k) \cdots w_N^T(k)]^T
$$

$$
v(k) = [v_1^T(k) \cdots v_N^T(k)]^T
$$

$$
H(k) = [H_1^T(k) \cdots H_N^T(k)]^T
$$

$$
G(k) = [G_1^T(k) \cdots G_N^T(k)]^T
$$

Then construct a local system model,

$$
x_j(k+1) = F_j(k)x_j(k) + G_j^{(r)}(k)w_j(k)
$$

$$
z_j(k) = H_j^{(r)}(k)x_j(k) + v_j(k)
$$

Here $G_j^{(r)}(k) \in \mathbb{R}^{n_j \times p_j}$ and $H_j^{(r)}(k) \in \mathbb{R}^{m_j \times n_j}$ denote the reduced-order process and measurement noise matrices respectively, with $\sum_{j=1}^{N} p_j = p$ and $\sum_{j=1}^{N} m_j = m$.

$F_j(k) \in \mathbb{R}^{n_j \times n_j}$ is the reduced-order state transition matrix operating on the reduced-order state vector $x_j(k) \in \mathbb{R}^{n_j}$.

It is obvious that Algorithm 3 is no longer valid under these new conditions, e.g. due to the possible dimension mismatch in the summation. However, if a matrix $D_j(k) \in \mathbb{R}^{n_j \times n}$ can be found such that,

$$
H_j(k) = H_j^{(r)}(k)D_j(k)
$$

and if there exists a matrix $E_j(k) \in \mathbb{R}^{n \times n_j}$ such that,

$$
G_j(k) = E_j(k)G_j^{(r)}(k)
$$

then Algorithm 3 can be rewritten as seen in Algorithm 5.
Algorithm 5 Global filter (reduced-order)

Initialization:
\[
\hat{x}(-1| -1) = \hat{x}_{\text{initial guess}} \\
\hat{P}(-1| -1) = \hat{P}_{\text{initial guess}}
\]

Global time-update:
\[
\hat{x}(k|k-1) = F(k-1)\hat{x}(k-1|k-1) \\
\hat{P}(k|k-1) = F(k-1) \hat{P}(k-1|k-1) F^T(k-1) + \sum_{j=1}^{N} E_j(k-1) \left[ G_j^{(r)}(k-1) Q_j(k-1) G_j^{(r)T}(k-1) \right] E_j^T(k-1)
\]

Global measurement-update:
\[
\hat{x}(k|k) = \hat{P}(k|k) \left( \hat{P}(k|k-1)^{-1} \hat{x}(k|k-1) + \sum_{j=1}^{N} D_j^T(k) \left[ H_j^{(r)T}(k) R_j^{-1}(k) z_j(k) \right] \right) =
\]
\[
\hat{P}(k|k) \left[ \hat{P}(k|k-1)^{-1} \hat{x}(k|k-1) + \sum_{j=1}^{N} D_j^T(k) \left\{ P_j(k|k)^{-1} \hat{x}(k|k) - P_j(k|k-1)^{-1} \hat{x}(k|k-1) \right\} \right]
\]
\[
\hat{P}(k|k)^{-1} = \hat{P}(k|k-1)^{-1} + \sum_{j=1}^{N} D_j^T(k) \left[ H_j^{(r)T}(k) R_j^{-1}(k) H_j^{(r)}(k) \right] D_j(k) =
\]
\[
\hat{P}(k|k)^{-1} + \sum_{j=1}^{N} D_j^T(k) \left\{ P(k|k)^{-1} - P_j(k|k-1)^{-1} \right\} D_j(k)
\]

The algorithms to be executed locally will have the same form as before, as seen in Algorithm 6.
**Algorithm 6** Local filter (reduced-order)

**Initialization:**
\[
\hat{x}_j(-1| -1) = \hat{x}_{\text{initial guess}} \\
\hat{P}_j(-1| -1) = \hat{P}_{\text{initial guess}}
\]

**Global time-update:**
\[
\hat{x}_j(k| k - 1) = F_j(k - 1)\hat{x}_j(k - 1| k - 1) \\
P_j(k| k - 1) = F_j(k - 1)P_j(k - 1| k - 1)F_j^T(k - 1) + G_j^{(r)}(k - 1)Q_j(k - 1)G_j^{(r)T}(k - 1)
\]

**Global measurement-update:**
\[
K_j(k) = P_j(k| k - 1)H_j^{(r)T}(k) [H_j^{(r)}(k)P_j(k| k - 1)H_j^{(r)T}(k) + R_j(k)]^{-1} \\
\hat{x}_j(k| k) = [I - K_j(k)H_j^{(r)}(k)] \hat{x}_j(k| k - 1) + K_j(k)z_j(k) \\
P_j(k| k) = [I - K_j(k)H_j^{(r)}(k)] P_j(k| k - 1)
\]

**Conclusions** Having a reduced-order model helps reducing the computation burden on each sensor node since fewer operations have to be performed. It also provides a solution to the problem where not all local filters can observe all state variables.

### 3.3 The Kalman Filter on Systolic Arrays

Apart from the more general methods of parallelizing the Kalman filter there has also been an ongoing research on how the Kalman filter can be computed efficiently on customized hardware architectures. The architecture in this field which perhaps has got the most attention in the past is the so-called systolic array.

The systolic array (Kung/Leiserson/of Computer Science, 1978; Hoffmann/Devadas/Agarwal, 2010) consists of a big one- or two-dimensional array of simple data processing elements (PEs) (see Figure 3.3). At every clock cycle each PE grabs a data element from the input data stream, processes it and then passes it on to the next PE in the array. In addition to being able to perform simple operations each PE can also be designed to have internal registers where intermediate results can be stored.

The concept is quite different from how the multicore architecture is designed. A multicore computer typically do not have nearly as many cores as the systolic array has PEs. On the systolic array no extra computational cost is associated with letting a PE perform just one operation. It would not be beneficial to let a core in a multicore system perform one single operation due to the overhead associated with loading and storing of the data and the synchronization handling.

However, there exists at least one recent attempt to convert existing systolic implementations to the multicore architecture. A method presented in (Vinjamuri/Prasanna, 2009) divides the entire systolic array into larger groups of PEs. Those groups are named macro-nodes and the PEs that the macro-nodes are built up by are called micro-nodes.

To determine how the systolic array can be divided up into macro-nodes, it is necessary to know the dependency relations in the array. As mentioned earlier the data flows through the array, at the pace of one step per clock cycle. An important observation to make here is the case when some PEs do not alter the data before it is passed on to the neighboring PE. This means that the neighboring PE is not dependent on the result from the previous PE. This fact can be exploited when laying out the work on the different cores on the multicore computer.
3.3.1 SIMD Arrays

Another architecture is the SIMD (“Single Instruction Multiple Data”) array. Just like with systolic arrays the SIMD array contains a large number of processing elements (PEs). One important difference though is that the commands, that are to be executed synchronously on each PE, are issued by a Control unit instead of being “hardwired”. The data on which to perform the operations on is already stored in the local memories of the PEs on beforehand instead of being clocked in as in the case with the systolic array.

The fact that a Control unit determines the behavior of the array at each clock cycle means that the approach taken in (Vinjamuri/Prasanna, 2009), to apply a systolic array design on a multicore computer, would possibly become much harder since the behavior of the array is no longer static.

A suggestion of an implementation on a commercially available SIMD array called “Connection Machine” is presented in (Palis/Krecker, 1990). The method used is a Square-Root like variant of the Kalman filter. What follows here is a very brief explanation of the method to give an idea of how an implementation on a SIMD array could look like.

Start with the well-known state-space representation of a system,
\[ x(k+1) = F(k)x(k) + \Gamma(k)u(k) + G(k)w(k) \]
\[ z(k) = H(k)x(k) + v(k) \]

The matrices \( P(k) \) and \( R_c(k) \) can be written in the factorized form,
\[ P(k) = L_p(k)D_p(k)L_p^T(k) \]
\[ R_c(k) = L_c(k)D_c(k)L_c^T(k) \]

where,
\[ R_c(k) = H(k)P(k)H^T(k) + R(k) \]
\[ P(k+1) = F(k)P(k)F^T(k) + G(k)Q(k)G^T(k) - K(k)R_c(k)K^T(k) \]

Let \( S \) be the matrix,
\[ S(k) = \begin{bmatrix} I & H(k)P(k)F^T(k) \\ F(k)P(k)H^T(k) & F(k)P(k)F^T(k) + G(k)Q(k)G^T(k) \end{bmatrix} \]

Then it can be shown that \( S \) can be factored as,
\[ S(k) = A(k)D(k)A^T(k) = \]
\[ = \begin{bmatrix} I & H(k)L_p(k) \\ 0 & F(k)L_p(k) \end{bmatrix} \begin{bmatrix} R(k) & 0 & 0 \\ 0 & D_p(k) & 0 \end{bmatrix} \begin{bmatrix} I & 0 \\ 0 & G(k) \end{bmatrix} \]
\[ \begin{bmatrix} L_p^T(k)H^T(k) & L_p^T(k)F^T(k) \\ 0 & G^T(k) \end{bmatrix} \] (3.4)
Alternatively, $S$ can also be factored as,
\[
S = A'(k)D'(k)A^T(k) = \\
L_e(k) \begin{bmatrix} 0 & 0 \\ K(k) L_e(k) & L_p(k+1) \end{bmatrix} D_e(k) \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & D_a \end{bmatrix} \\
\begin{bmatrix} L_e^T(k) & L_p^T(k)K^T(k) \\ 0 & L_p^T(k+1) \end{bmatrix}
\] \tag{3.5}

To get from (3.4) to (3.5) one possibility is to use a triangularization algorithm like for instance the “fast Givens rotations”.

The complete scheme proposed in (Palis/Krecker, 1990) for performing the Kalman filtering is replicated in Algorithm 7.

**Algorithm 7 Square-Root Kalman with fast Givens rotation**

**Input:** $\hat{x}_0$, $L_{p,0}$, $D_{p,0}$, $F(k)$, $H(k)$, $R(k)$, $Q(k)$, $G(k)$, $z(k)$ for $k \geq 0$

**Output:** $\hat{x}(k)$ for $k > 0$

**Steps:** For $k = 0, 1, 2, \ldots$, do the following:

1. Set up matrices $A$ and $D$ such that:
   \[
   A = \begin{bmatrix} I & H(k)L_p(k) & 0 \\ 0 & F(k)L_p(k) & G(k) \end{bmatrix} \text{ and } D = \begin{bmatrix} R(k) & 0 & 0 \\ 0 & D_p(k) & 0 \\ 0 & 0 & Q(k) \end{bmatrix}
   \]

2. Triangularize $A$ to obtain:
   \[
   A' = \begin{bmatrix} L_e(k) & 0 & 0 \\ K(k) L_e(k) & L_p(k+1) & 0 \\ D_e(k) & 0 & 0 \end{bmatrix} \text{ and } D' = \begin{bmatrix} 0 & 0 & 0 \\ 0 & D_p(k+1) & 0 \\ 0 & 0 & D_a \end{bmatrix}
   \]

   such that $S = ADAT = A'D'A^T$.

3. Compute $K(k)[z(k) - H(k)\hat{x}(k)]$ as follows:
   (a) Compute $c(k) = z(k) - H(k)\hat{x}(k)$
   (b) Solve $L_e(k)s(k) = c(k)$ for $s(k)$
   (c) Compute $K(k)[z(k) - H(k)\hat{x}(k)] = [K(k)L_e(k)]s(k)$

4. Compute and output:
   \[
   \hat{x}(k+1) = F(k)\hat{x}(k) + \Gamma(k)u(k) + K(k)[z(k) - H(k)\hat{x}(k)]
   \]

In the paper focus is devoted to Step 2 in Algorithm 7 because it is concluded that it stands for $3/4$ of the total number of floating-point operations. Step 2 can be written in a pseudo programming language as,
\[
\text{for } q = (1 \text{ to } M) \text{ do} \\
\quad \text{for } j = (q \text{ to } N) \text{ do} \\
\quad \quad (A,D) = T_{jq}(A,D)
\]
end
end

The parallelization strategy presented in the paper is then to parallelize the inner loop using a constant number of operations. Let $A^{(q)}$ and $D^{(q)}$ denote the matrices at the end of iteration $q$.

Then it can be shown that the elements in $A$ and $D$ can be computed as,
\[
\alpha^{(q)} = \frac{s^{(q-1)}}{s^{(q)}} \quad \beta^{(q)} = \frac{s^{(q-1)}}{s^{(q)}} \alpha^{(q-1)}, \quad q < j \leq N
\]
\[ a_{iq}^{(q)} = \frac{s_{iq}^{(q-1)}}{s_{NN}^{(q-1)}} \]

and \[ a_{ij}^{(q)} = a_{ij}^{(q-1)} - \frac{s_{ij}^{(q-1)}}{s_{NN}^{(q-1)}} a_{qj}^{(q-1)}, \quad q \leq i \leq M \text{ and } q < j \leq N \]

where,

\[ s_{ij}^{(q-1)} = \sum_{l=q}^{j} a_{ql}^{(q-1)} a_{jl}^{(q-1)} d_{l}^{(q-1)}. \]

Thus the first \((q-1)\) diagonal elements in \(D^{(q)}\) will be the same as in \(D^{(q-1)}\) and the first \((q-1)\) rows and the first \((q-1)\) columns of \(A^{(q)}\) will be the same as in \(A^{(q-1)}\).

The complete scheme for implementing the fast Givens rotation step is presented in Algorithm 8. Because this algorithm is aimed to run on a SIMD computer, which can be seen on the algorithm structure, it is not clear if it could be useful for multicore computers.

**Algorithm 8** Fast Givens rotation on a “Connection Machine”

Initialization: See the original paper (Palis/Krecker, 1990)

For \(q = 1, \ldots, M\):

1. For all columns \(j, q \leq j \leq N\), spread-with-copy \(a(q, j)\) down the column and store it in \(b(i, j)\), 
   \(q \leq i \leq M\). (Thus, \(b(i, j) = a_{qj}^{(q-1)}\))

2. For \(q \leq i \leq M\) and \(q \leq j \leq N\), compute \(t1(i, j) = a(i, j) \cdot b(i, j) \cdot d(i, j)\) (Thus, \(t1(i, j) = a_{ij}^{(q-1)} a_{qj}^{(q-1)} d_{j}^{(q-1)}\))

3. For all rows \(i, q \leq i \leq M\), perform an upward, exclusive, scan-with-add over the set of values
   \(\{t1(i, j) | q \leq j \leq N+1\}\). (Thus, \(t1(i, j) = s_{ij}^{(q-1)}\))

4. For all columns \(j, q \leq j \leq N\), spread-with-copy \(t1(q, j)\) down the column and store it in \(t2(i, j)\), 
   \(q \leq i \leq M\). (Thus, \(t2(i, j) = s_{qj}^{(q-1)}\))

5. For \(q \leq i \leq M\) and \(q < j \leq N+1\), compute \(t1(i, j) = t1(i, j)/t2(i, j)\). (Thus, \(t1(i, j) = s_{ij}^{(q-1)} a_{qj}^{(q)}\) and \(t2(i, j) = s_{qj}^{(q-1)} d_{q}^{(q)}\))

6. For \(q \leq i \leq M\), send \(t1(i, N+1)\) to \(a(i, q)\) and send \(t2(i, N+1)\) to \(d(i, q)\). (Thus, \(a(i, q) = a_{iq}^{(q)}\) and \(d(i, q) = d_{q}^{(q)}\))

7. For \(q \leq i \leq M\) and \(q < j \leq N\), compute \(a(i, j) = a(i, j) - b(i, j) \cdot t1(i, j)\). (Thus, \(a(i, j) = a_{ij}^{(q)}\))

8. For \(q \leq i \leq M\) and \(q < j \leq N\), get \(t1(i, j)\) from \(t2(i, j+1)\). (Thus, \(t1(i, j) = s_{qj}^{(q-1)}\) and \(t2(i, j) = s_{qj}^{(q-1)}\))

9. For \(q \leq i \leq M\) and \(q < j \leq N\), compute \(d(i, j) = d(i, j) \cdot t2(i, j)/t1(i, j)\). (Thus, \(d(i, j) = d_{j}^{(q)}\))

### 3.3.2 Warp

Another method which is based on the same Square-Root Kalman filter, using the same decomposition technique, is found in (Itzkowitz/Baheti, 1989). In the paper an algorithm designed to run on a Warp computer (Kung, 1984) is proposed.

The Warp architecture is a hardware implementation of a systolic array originating from 1984. Each PE has its own pipelined multiplier and ALU (“Arithmetic Logic Unit”), and has also access to a small private RAM. This makes it slightly more advanced and versatile than many other systolic arrays. The PEs are interconnected via a 1D array.

In (Itzkowitz/Baheti, 1989) the implementation of the triangularization step (Step 2 in Algorithm 7) is instead described by Algorithm 9.


\begin{algorithm}
\caption{A Fast Givens rotation}
\begin{algorithmic}
\State For each row $A_i$ in $A$ ($i = 1, \cdots, M$)
\For {each element $j$ in $A_i$ ($j = N, \cdots, i+1$)}
\State Calculate $D'_i$
\State Zero out row $A_i$ using the rotation matrix $\Psi_{i,j}$ to calculate $A'_i$
\EndFor
\EndFor
\end{algorithmic}
\end{algorithm}

It is noticed here that the loop modifying the rows below the current row $A_i$ can be parallelized, since the iterations in this operation are not dependent on each other. It is therefore suggested in the article to dedicate $M$ cells to perform this task.

Another fact that can be noticed here is that during the triangularization the $L_e(k)$ and $K(k)L_e(k)$ matrices become available immediately after the first $m$ rows in the $A$ matrix have been zeroed out. This opens up for the opportunity of computing the Kalman gain, $K(k)$, and hence also the state update (time and measurement) while the processing of the remaining parts of the triangularization is still in progress.

### 3.3.3 Decoupling of Time-Update and Measurement-Update Equations

In the standard Kalman filter the measurement-update equation depends on the predicted state and covariance estimates computed using the time-update equations. By introducing a time lag of one time step this dependency can be broken, making it possible to perform the time-update and measurement-update steps on two cores simultaneously. This decoupling technique is proposed in (Youmin et al., 1993).

Given the usual state space model,

$$
x(k+1) = F(k)x(k) + \Gamma(k)u(k) + G(k)w(k)
$$

$$
z(k) = H(k)x(k) + v(k)
$$

Consider the state time-update and state measurement-update equations respectively (Youmin et al., 1993),

$$
\hat{x}(k+1|k) = F(k)\hat{x}(k|k) + \Gamma(k)u(k) = F(k)F(k-1)\hat{x}(k-1|k-1) + F(k)\Gamma(k-1)u(k-1) + \Gamma(k)u(k) 
$$

$$
\hat{x}(k|k) = \hat{x}(k|k-1) + K(k)[z(k) - H(k)\hat{x}(k|k-1)] 
$$

Expression (3.6) predicts the state vector two time-steps ahead in time. By this, (3.6) and (3.7) can be executed in parallel since the measurement-update expression (3.7) no longer depends on the result from the time-update expression (3.6).

After a number of equation rearrangements, substitutions and by concluding that $\hat{x}(k+1|k) - x(k+1|k)$ and $w(k)$ are uncorrelated, the state estimate error covariance matrix,

$$
P(k+1|k) = E \left[ (\hat{x}(k+1|k) - x(k+1|k))(\hat{x}(k+1|k) - x(k+1|k))^T \right]
$$
can be expressed as,

\[
P(k + 1|k) = F(k)F(k - 1)P(k - 1|k - 1)F^T(k - 1)F^T(k) + 
F(k)G(k - 1)Q(k - 1)G^T(k - 1)F^T(k) + G(k)Q(k)G^T(k)
\]

The state estimate error covariance matrix corresponding to the measurement-update phase will remain the same. The complete algorithm is summarized in Algorithm 10 (the entire time-update equation has been shifted one time-step to fit the measurement-update equation).

**Algorithm 10** A decoupled Kalman filter algorithm

**Initialization:** See the original paper (Youmin et al., 1993)

**Time-update equation:**

\[
\hat{x}(k|k - 1) = F(k - 1)F(k - 2)\hat{x}(k - 2|k - 2) + F(k - 1)\Gamma(k - 2)u(k - 2) + 
\Gamma(k - 1)u(k - 1)
\]

\[
P(k|k - 1) = F(k - 1)F(k - 2)P(k - 2|k - 2)F^T(k - 2)F^T(k - 1) + 
F(k - 1)G(k - 2)Q(k - 2)G^T(k - 2)F^T(k - 1) + 
G(k - 1)Q(k - 1)G^T(k - 1)
\]

**Measurement-update equation:**

\[
K(k) = P(k|k - 1)H^T(k)\left[H(k)P(k|k - 1)H^T(k) + R(k)\right]^{-1}
\]

\[
\hat{x}(k|k) = \hat{x}(k|k - 1) + K(k)\left[z(k) - H(k)\hat{x}(k|k - 1)\right]
\]

\[
P(k|k) = [I - K(k)H(k)]P(k|k - 1)
\]

**Conclusions** According to the authors this algorithm can, when it is implemented on a linear systolic array, achieve a speed-up of 1.8 compared to when the conventional Kalman filter is used. A performance gain can hopefully also be achieved on a multi-core computer. However, this method has two major disadvantages. First of all, the filter is not identical to the original Kalman filter and might therefore suffer from worse accuracy. Secondly, the method is not scalable in the way that is takes advantage of more than precisely two CPU cores.

4 Choice of System

All methods covered in this report assume a linear model of the system of interest. In tracking systems radars are often used to measure the position of the target in angular coordinates. However, if angular coordinates are chosen as the state vector, the equations describing the dynamics of the system will be nonlinear. On the other hand, if the state vector is based on Cartesian coordinates the relation between the measurement vector and the state vector is nonlinear. So in either way, nonlinearities will be present.

Another common real-life example is where a number of tanks containing liquids are connected in cascade. The aim is then to keep the height of the liquids at constant levels. However, just as in the case with the tracking system this leads to nonlinear models. This could be avoided by linearizing the model around some desired working points, but this requires a regulator.

To keep things simple it would be desirable to have a model which is simple, linear, stable, scalable and does not require a regulator.
4.1 A Temperature Bar

Consider a one-dimensional bar where the two outer ends are placed in two temperature reservoirs with controllable temperatures. Assume that the bar is divided into smaller imaginary segments with different temperatures. This makes it possible to create a mathematical model of the system where each temperature is represented by a state variable. The model turns out to be simple, linear, stable and does not need a regulator. It is also scalable in the way that it is trivial to expand the model with more state variables by simply dividing the bar into more segments.

![Temperature Bar Diagram](image)

Figure 4.1: The temperature bar

**Derivation**

Assume that the well-known 1D heat equation holds,

\[
\frac{\partial T(x, t)}{\partial t} = \alpha \frac{\partial^2 T(x, t)}{\partial x^2}
\]

\(T(x, t)\) denotes the temperature at time \(t\) and position \(x\). \(\alpha\) is the thermal diffusivity.

Then by approximating the left-hand side with a forward difference and the right-hand side with a central difference the equation can be discretized as,

\[
\frac{Ti(k+1) - Ti(k)}{\Delta t} = \alpha \frac{Ti+1(k) - 2Ti(k) + Ti-1(k)}{(\Delta x)^2}
\]

(4.1)

\(Ti(k)\) now represents the temperature at discrete time \(k\) and discrete position \(i\). \(\Delta x\) is the distance between the center of two bar segments. \(\Delta t\) is the sample time of the system with the requirement,

\[\Delta t \leq \frac{(\Delta x)^2}{2\alpha}\]

for the approximations to be stable (Heath, 2002).

Equation (4.1) is solved for \(Ti(k)\),

\[Ti(k+1) = c1 (Ti-1(k) + Ti+1(k)) + c2 Ti(k)\]

where \(c1 = \frac{\alpha \Delta t}{(\Delta x)^2}\) and \(c2 = 1 - 2c1\).

Now let \(Ti(k)\) be represented by the state variables \(xi(k)\). Also let \(T0(k)\) and \(TN+1(k)\) be the system control inputs \(u_{rl}(k)\) and \(u_{rr}(k)\) respectively. This will form the discrete state-space model,

\[
\begin{bmatrix}
  x1(k+1) \\
  x2(k+1) \\
  x3(k+1) \\
  \vdots \\
  xN(k+1)
\end{bmatrix} = \begin{bmatrix}
  c2 & c1 \\
  c1 & c2 & c1 \\
  \vdots & \ddots & \ddots & \ddots \\
  c1 & c2 & \cdots & \cdots & c1
\end{bmatrix} \begin{bmatrix}
  x1(k) \\
  x2(k) \\
  x3(k) \\
  \vdots \\
  xN(k)
\end{bmatrix} + \begin{bmatrix}
  c1 & 0 \\
  0 & c1
\end{bmatrix} \begin{bmatrix}
  u_{rl}(k) \\
  u_{rr}(k)
\end{bmatrix}
\]

\[
\begin{bmatrix}
  z1(k) \\
  z2(k) \\
  z3(k) \\
  \vdots \\
  zN(k)
\end{bmatrix} = \begin{bmatrix}
  1 & 1 & \cdots & \cdots & 1 \\
  \vdots & \ddots & \ddots & \ddots & \vdots \\
  1
\end{bmatrix} \begin{bmatrix}
  x1(k) \\
  x2(k) \\
  x3(k) \\
  \vdots \\
  xN(k)
\end{bmatrix}
\]
5 Simulations

Before the algorithms are implemented on a real multi-core computer they will be implemented in MATLAB. This makes it easy to simulate and evaluate the different methods on the chosen model before some of them make it to the next step.

The purpose of this simulation step is not to measure the speed of the algorithms, but rather to observe how well the algorithms give correct state estimates. No attempts have been made to use any of the multi-core functionality in MATLAB.

5.1 Setup

The principle behind the simulation code is as follows. First an instance of the discrete model is created, taking as arguments the physical length, the number of segments (states), a list of observed states and the time sample period. This will output the state-space matrices ($F$, $Γ$, $G$ and $H$).

The next step is to choose the start values for the system ($x_0$ and $P_0$). Now when everything related to the model is set up it is time to initialize the Kalman filters. Each filter is initialized with start guesses for $x_0$ and $P_0$. To mimic a real situation, the guess for the state estimate error covariance matrix $P_0$ is taken to be the true value times a number. The start-up guess for $x_0$ is then assumed to be the true value $x_0$ plus a normal distributed vector with a covariance in comport with $P_{0,guess}$

For the two filters based on partitioned observation equations (Full-order distributed Filter and Gain-Fusion Filter) a list specifying how the partitioning should be carried out is also provided.

The actual simulation consists of a big loop iterating through each time sample, $k = 0, 1, \cdots, N$. For each time sample a measurement is taken on the true model. Again, to mimic a real situation, normal distributed noise is added to the measurement vector. The noisy measurement vector $z(k)$ together with the control signal $u(k)$, the model ($F(k)$, $Γ(k)$, $G(k)$ and $H(k)$) and the error covariances $Q(k)$ and $R(k)$ are then fed into each Kalman filter. With this information the filters have all what it takes to estimate the states. At the end of each iteration the system takes one step in time by simply computing expression (2.1) for $x(k + 1)$.

A typical scheme for how a standard Kalman algorithm could be implemented on a digital computer is depicted in Figure 5.1. The figure also shows where a controller/regulator is often placed when the state estimator is part of a control loop. In this thesis work no controller is necessary because the system is stable and to avoid making things more complicated, so therefore it has been left out.

![Diagram of Scheme for implementing a Kalman algorithm](image)

Figure 5.1: Scheme for implementing a Kalman algorithm

5.2 Experiments and Results

In the first set of plots (Figure 5.2 - 5.6), the scenario is simulated where the initial temperature of the temperature bar (Section 4.1) is,

$$ x(0) = \begin{bmatrix} 250 \\ 300 \\ 350 \end{bmatrix} $$

and where a constant control signal is present,

$$ u(k) = \begin{bmatrix} 300 \\ 300 \end{bmatrix}^T, \quad k = 0, 1, \cdots $$
It is also assumed that only the first two state variables are measured, i.e.

\[ H = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \end{bmatrix} \]

System and measurement noise are added according to the same \( Q \) and \( R \) matrices that are also fed to each individual algorithm.

Intuitively, when running the simulation, the temperatures in the bar should eventually reach the same temperatures as those in the two reservoirs, which in this case are both 300 K.

All plots except for the Full-order Distributed algorithm (Figure 5.6) behave reasonably well. The algorithm fails to estimate the state variable that is not measured (the temperature with the initial temperature 350 K).

![Standard Kalman vs True states](image1)

**Figure 5.2: Standard Kalman (for reference)**

![Fast Givens Kalman vs True states](image2)

**Figure 5.3: Fast-Givens Kalman**

![Fusion Gain Kalman vs True states](image3)

**Figure 5.4: Fusion-Gain Kalman**
The errors in the Full-order Distributed algorithm simulation become even more clear when studying a step response. The step is taken by setting both temperature reservoirs to a constant temperature, $u(k) = \begin{bmatrix} 300 & 300 \\ 600 & 600 \end{bmatrix}^T$, $k < 50$

$$u(k) = \begin{cases} \begin{bmatrix} 300 & 300 \\ 600 & 600 \end{bmatrix}^T, & k < 50 \\ \end{cases}$$

$k \geq 50$
Summary  Time has not allowed further investigations of the problem with the Full-order Distributed algorithm. Because this algorithm is more or less a special case of the Reduced-order method, that
algorithm will not be covered in the rest of the report either.

6 Algorithm Analysis

At this point only three candidates remain, namely “Fusion-Gain”, “Fast-Givens Rotation” and “Decoupled Kalman”. “Decoupled Kalman” differs from the other two in the way that it is designed to be used on exactly two cores. For this reason this algorithm is of little interest from a scalability perspective. Therefore it has been decided to abandon this algorithm for now. There are in other words only two algorithms remaining, “Fusion Gain” and “Fast Givens”.

For each one of them it remains to estimate the required number of floating-point operations per filter iteration in order to get the speedup potential. The speedup is defined as the time it would take to execute the algorithm with only one available core divided by the time it would take to run the algorithm if the parallel parts would run completely asynchronous.

To simplify and reduce the potential for human errors when computing the number of floating-point operations, a Matlab script has been written which takes an mathematical expression as input and outputs the expected number of floating-point operations.

To get a measure of the state estimate accuracy for each algorithm a Root-mean-square approximation will be computed according to,

$$\text{rms} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (x_{\text{true}} - x_{\text{est}})^2}$$

6.1 Fusion-Gain

Parallelization Strategy The parallelization strategy for the Fusion-Gain algorithm is to execute the $M$ local filter expressions on $M$ separate cores.

Speed Table 6.1 lists the expressions involved in the computation of the Fusion-Gain filter. The corresponding number of floating-points required is written in either of the columns “Sequential” and “Parallel” depending on whether that expression is executed at the local filter or not.

In the listing, it is assumed that the computation burden is divided equally on the $M$ cores ($M = 1, 2, \ldots, n_x$), i.e.,

$$n_{z_j} = \frac{n_x}{M}$$

Figure 6.2 indicates what speedup could be expected from the algorithm. The speedup gained by using a dual core processor ($M = 2$) instead of a single one ($M = 1$) is quite significant. This can be explained by the shrinkage of the $n_x^3$ term in the parallel section. Then, as the number of cores increases the same $n_x^3$ term will become almost negligible and the speedup will eventually become constant. As $M$ increases even further, to reach the same order of magnitude as $n_x$ itself, an indication of a speeddown can be noticed. This is due to the increasing amount of work required at row 7 and 8 in Table 6.1. The speedup will decay as $\propto \frac{1}{M}$ for large values of $M$.

This analysis shows that the algorithm scales poorly as the number of cores increases, but could be useful on todays multi-core computers where dual-core and quad-core processors are still commonly used.

To improve the performance of the algorithm an alternative could be to parallelize the remaining two $O(n_x^3)$ terms (row 2 and 10) in Table 6.1 using multicore libraries for standard block multiplication available in e.g. Intel MKL (see Section 7.1).
<table>
<thead>
<tr>
<th>#</th>
<th>Expression</th>
<th>Floating-point operations per filter iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td>1</td>
<td>( x_j(k</td>
<td>k-1) = x(k</td>
</tr>
<tr>
<td>2</td>
<td>( P_j(k</td>
<td>k-1) = P(k</td>
</tr>
<tr>
<td>3</td>
<td>( K_j = \frac{1}{\gamma_j} P^{-1} H_j^T \left[ H_j P^{-1} H_j^T + \frac{1}{\gamma_j} R_j \right]^{-1} )</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>( \dot{x}_j(k</td>
<td>k) = x(k</td>
</tr>
<tr>
<td>5</td>
<td>( K_j H_j )</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>( K_j R_j K_j^T )</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>( \dot{x}(k) = (1 - N) \dot{x}(k</td>
<td>k-1) + \sum_{j=1}^{M} \dot{x}_j(k</td>
</tr>
<tr>
<td>8</td>
<td>( P(k</td>
<td>k) = \left[ I - \sum_{j=1}^{M} K_j H_j \right] P(k</td>
</tr>
<tr>
<td>9</td>
<td>( \dot{x}(k+1</td>
<td>k) = F \dot{x}(k</td>
</tr>
<tr>
<td>10</td>
<td>( P(k+1</td>
<td>k) = FP(k</td>
</tr>
</tbody>
</table>

| Total Sum: | \( 8n_z^2 + (2n_w + 2M - 2)n_x^2 - \) \( + (2n_w - n_x + 2n_u + M)n_x + 1 \) | \( 8n_x n_z + 6n_x n_z^2 + 3n_z^2 + n_z^2 + n_x^2 \) |

| Total Sum (when \( n_z = n_w = n_x \) and \( n_z = \frac{n_x}{M} \)): | \( 12n_x^2 + (2M - 3)n_x^2 + (2n_u + M)n_x + 1 \) | \( (\frac{n_x}{M} + \frac{n_x}{M^2} + \frac{3}{M})n_x^2 + (\frac{1}{M} - 1)n_x^2 \) |
Figure 6.1: Predicted number of floating-point operations per filter iteration, $n_x = 1000$, $n_u = 2$, $n_z = 1000$, $n_w = 1000$

Figure 6.2: Predicted speedup, $n_x = 1000$, $n_u = 2$, $n_z = 1000$, $n_w = 1000$

**Accuracy** The performance achieved with the algorithm comes at the cost of slightly worse accuracy. As mentioned earlier in the report, this Fusion Gain algorithm is not “optimal”, in contrast to the original Kalman filter, so it is expected for this algorithm to give greater estimation errors. One would also expect the divergence to increase when the number of subsystems are chosen large. Figure 6.3 confirms this theory. The standard Kalman filter is a better estimator in this regard.

### 6.2 Fast-Givens Rotation

To make it easier to follow, Algorithm 11 and 12 are duplicates of Algorithm 7 and 9.

**Parallelization Strategy** The rotation part of the algorithm starts at the last element on the first row in the $A$ matrix. At this point this element is zeroed out using the rotation matrix $\Psi_{i,j}$. After this, the same transformation has to be applied to the elements below the current element. The plan is to perform this operation on each element in parallel. With this done, the next step is to zero out the element to the left of the current element and then repeat the process to the elements below that one. This procedure continues until reaching the diagonal element of the current row. Then the algorithm jumps to the next row and the process described above repeats.

$A$ is a $m \times n$ matrix, where $m = n_x + n_z$ and $n = 2n_x + n_z$. This means that when the outer loop is at index $i$, the parallel section will be partitioned in $M$ slices where each of the $M$ cores is responsible for transforming $\frac{m-i}{M}$ rows. Clearly, as the outer loop traverses the rows of the matrix ($i$ increases) the total parallelizable work becomes smaller and smaller.

**Speed** The rotation will be implemented in a slightly different way than presented in the original paper (Itzkowitz/Baheti, 1989) where a $2n_z + n_x$ dimension unit matrix is set up each time in the 2nd
Figure 6.3: Root-mean-square errors. Standard Kalman in dashed lines.
Algorithm 11 The Kalman Filter using triangularization

Input: $\hat{x}_o$, $L_p, 0$, $F(k)$, $H(k)$, $R(k)$, $Q(k)$, $G(k)$, $z(k)$ for $k \geq 0$
Output: $\hat{x}(k)$ for $k > 0$

Steps: For $k = 0, 1, 2, \cdots$, do the following:

1. Set up matrices $A$ and $D$ such that:
   
   $$ A = \begin{bmatrix} I & H(k)L_p(k) & 0 \\ 0 & F(k)L_p(k) & G(k) \end{bmatrix} \quad \text{and} \quad D = \begin{bmatrix} R(k) & 0 & 0 \\ 0 & D_p(k) & 0 \\ 0 & 0 & Q(k) \end{bmatrix} $$

2. Triangularize $A$ to obtain:
   
   $$ A' = \begin{bmatrix} L_e(k) & 0 & 0 \\ K(k)L_e(k) & L_p(k+1) & 0 \\ D_e(k) & 0 & 0 \\ 0 & D_p(k+1) & 0 \\ 0 & 0 & D_a \end{bmatrix} $$

   such that $S = A D A^T = A' D' A'^T$.

3. Compute $K(k)[z(k) - H(k)\hat{x}(k)]$ as follows:
   
   (a) Compute $c(k) = z(k) - H(k)\hat{x}(k)$
   (b) Solve $L_e(k)s(k) = c(k)$ for $s(k)$
   (c) Compute $K(k)[z(k) - H(k)\hat{x}(k)] = [K(k)L_e(k)]s(k)$

4. Compute and output:
   
   $$ \hat{x}(k+1) = F(k)\hat{x}(k) + \Gamma(k)u(k) + K(k)[z(k) - H(k)\hat{x}(k)] $$

Algorithm 12 Triangularization using Fast Givens Rotation

For each row $A_i$ in $A$ ($i = 1, \cdots, M$)

For each element $j$ in $A_i$ ($j = N, \cdots, i+1$)

   Calculate $D'_i$

   Zero out row $A_i$ using the rotation matrix $\Psi_{i,j}$ to calculate $A'_i$

TO BE PARALLELIZED {

   For each $A_q$ below $A_i$ ($q = i+1, \cdots, M$)

   Calculate $A'_q$ using $\Psi_{i,j}$

} EndFor

EndFor
Table 6.2: Number of runs at each loop level

<table>
<thead>
<tr>
<th>Expression</th>
<th>#Iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1st level for-loop</td>
<td>(n_x + n_z)</td>
</tr>
<tr>
<td>2nd level for-loop</td>
<td>(\frac{2}{3}n_x^2 + (2n_z - \frac{2}{3})n_x + \frac{1}{2}n_z^2 - \frac{2}{3}n_z)</td>
</tr>
<tr>
<td>3rd level for-loop</td>
<td>(\frac{5}{6}n_x^2 + (2n_z - 3)n_z^2 + \left(\frac{1}{2}n_x^2 - \frac{9}{7}n_z + \frac{13}{6}\right)n_x + \frac{1}{3}n_x^3 - \frac{3}{2}n_x^2 + \frac{13}{18}n_x)</td>
</tr>
</tbody>
</table>

Table 6.3: Floating-point operations at each loop iteration

<table>
<thead>
<tr>
<th>2nd Loop Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
</tr>
<tr>
<td>(d'<em>i = A</em>{ii}D_i + A_{ij}D_j)</td>
</tr>
<tr>
<td>(d'<em>j = 2A</em>{ij}D_j)</td>
</tr>
<tr>
<td>(\Psi_{ij} = -A_{ij})</td>
</tr>
<tr>
<td>(\Psi_{jj} = A_{jj})</td>
</tr>
<tr>
<td>(\Psi_{ii} = \frac{A_{ii}D_j}{d'_j})</td>
</tr>
<tr>
<td>(\Psi_{ji} = \frac{A_{ij}D_i}{d'_i})</td>
</tr>
<tr>
<td><strong>Total Sum:</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>3rd Loop Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expression</td>
</tr>
<tr>
<td>(A_{ki} = \Psi_{ii}A_{ki} + \Psi_{ji}A_{kj})</td>
</tr>
<tr>
<td>(A_{kj} = \Psi_{ij}A_{ki} + \Psi_{jj}A_{kj})</td>
</tr>
<tr>
<td><strong>Total Sum:</strong></td>
</tr>
</tbody>
</table>

level “for loop”. Later the same matrix, \(\Psi\), is also used to rotate the current top row. Both these operations are approximately \(O(n_x^2)\), and in the 2nd level “for loop” (Table 6.2) they would become \(O(n_x^4)\). Using the fact that a multiplication between the rotation matrix, \(\Psi\), and a row vector only makes changes to two elements in the vector this operation could be reduced to a constant number of operations as seen in Table 6.3.

Apart from this most of the flops calculations are straightforward. However, the operations inside the “for loops” need special attention. Firstly, the number of times each loop is executed needs to be calculated (Table 6.2). Secondly, the number of flops performed at each loop iteration is determined (Table 6.3). Assuming that the operations performed at the 3rd loop level can be parallelized the final result can be seen at row 3 in Table 6.4.

**Accuracy** The Fast Givens algorithm is essentially a parallel implementation of a one-step-ahead Kalman predictor. Therefore it does not have the same chance of estimating the state vector as accurately as the filtering Kalman. Table 6.5 reveals this fact.

![Figure 6.4: Predicted number of floating-point operations per filter iteration, \(n_x = 1000, n_u = 2, n_z = 1000, n_w = 1000\)](image)
<table>
<thead>
<tr>
<th>#</th>
<th>Expression</th>
<th>Floating-point operations per filter iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Sequential</td>
</tr>
<tr>
<td></td>
<td>$HL_p$</td>
<td>$2n_x^2n_z - n_xn_z$</td>
</tr>
<tr>
<td></td>
<td>$FL_p$</td>
<td>$2n_x^2n_z - n_x^2$</td>
</tr>
<tr>
<td>1</td>
<td>$A = \begin{bmatrix} I_{n_z \times n_z} &amp; [HL_p] \ 0_{n_z \times n_z} &amp; [FL_p] \end{bmatrix} G$</td>
<td>$2n_xn_z + n_x^2 + n_wn_x$</td>
</tr>
<tr>
<td>Note:</td>
<td></td>
<td>Assuming that $A$ is overwritten</td>
</tr>
<tr>
<td>2</td>
<td>$D = \begin{bmatrix} R \ D_p \end{bmatrix}$</td>
<td>$2n_x^2 + n_z^2$</td>
</tr>
<tr>
<td>3</td>
<td>$[A',D'] = \text{Rot}(A,D)$</td>
<td>$\frac{27}{2}n_z^2 + (26n_x - \frac{39}{2})n_x + \frac{13}{2}n_x^2 - \frac{39}{2}n_z$</td>
</tr>
<tr>
<td>4</td>
<td>$c = z - Hx$</td>
<td>$2n_xn_z$</td>
</tr>
<tr>
<td>5</td>
<td>Solve $L_eo = c$</td>
<td>$n_z^2$</td>
</tr>
<tr>
<td>6</td>
<td>$K_{z,z}Hx = [KL_e] \cdot o$</td>
<td>$2n_xn_z - n_x$</td>
</tr>
<tr>
<td>7</td>
<td>$x^\prime = Fx + Tu$</td>
<td>$2n_x^2 + 2n_xn_z - n_x$</td>
</tr>
<tr>
<td>8</td>
<td>$x = x^\prime + K_{z,z}Hx$</td>
<td>$n_x$</td>
</tr>
<tr>
<td>9</td>
<td>$L_{p+} = A'(n_z +1 : n_z + n_x, n_z +1 : n_z + n_x)$</td>
<td>$n_x^2$</td>
</tr>
<tr>
<td>10</td>
<td>$D_{p+} = D'(n_z +1 : n_z + n_x, n_z +1 : n_z + n_x)$</td>
<td>$n_x^2$</td>
</tr>
<tr>
<td></td>
<td><strong>Total Sum:</strong></td>
<td>$2n_x^2 + (2n_x + \frac{39}{2})n_x^2 + (31n_x + 2n_u + n_w - \frac{41}{2})n_x + n_x^3 + \frac{11}{2}n_x^2 - \frac{39}{2}n_x$</td>
</tr>
</tbody>
</table>
Figure 6.5: Predicted speedup, $n_x = 1000$, $n_u = 2$, $n_z = 1000$, $n_w = 1000$

Table 6.5: Root-mean-square errors for Fast Givens

For $n_z = 0.2n_x$:

<table>
<thead>
<tr>
<th>$n_x$</th>
<th>Fast Givens</th>
<th>Std Kalman</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.31</td>
<td>0.61</td>
</tr>
<tr>
<td>500</td>
<td>1.22</td>
<td>0.71</td>
</tr>
<tr>
<td>1000</td>
<td>1.23</td>
<td>0.73</td>
</tr>
</tbody>
</table>

For $n_z = 0.5n_x$:

<table>
<thead>
<tr>
<th>$n_x$</th>
<th>Fast Givens</th>
<th>Std Kalman</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.41</td>
<td>0.36</td>
</tr>
<tr>
<td>500</td>
<td>1.20</td>
<td>0.44</td>
</tr>
<tr>
<td>1000</td>
<td>1.19</td>
<td>0.45</td>
</tr>
</tbody>
</table>

For $n_z = 1.0n_x$:

<table>
<thead>
<tr>
<th>$n_x$</th>
<th>Fast Givens</th>
<th>Std Kalman</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>1.32</td>
<td>0.25</td>
</tr>
<tr>
<td>500</td>
<td>1.25</td>
<td>0.26</td>
</tr>
<tr>
<td>1000</td>
<td>1.23</td>
<td>0.27</td>
</tr>
</tbody>
</table>
6.3 Summary

Now when the analysis of the multicore performance is finished it would be interesting to do a comparison between the methods. This can be done by setting the system order to some realistic high value (here \( n_x = 1000 \)) and plot the flops count and speedup for different number of system outputs (here \( n_z = [0.2 \ 0.5 \ 1.0]n_x \)). The results are interesting and can be seen in Figure 6.6 and 6.7.

The plots clearly show that there is no gain in using the Fusion Gain algorithm at all for small number of outputs (the upper left plot in Figure 6.6). For a greater number of outputs the results implicate that neither of the parallel algorithms perform better than the standard implementation of the Kalman filter on a single-core CPU. When on a dual-core or better, Fusion Gain begins to outperform the Standard Kalman Filter.

Fast Givens is faster than the Standard Kalman Filter independent of the number of cores when only 20% of the states are measured. When more states are measured Fast Givens loses some of the advantage it had over Fusion Gain, but it is still faster.
Figure 6.7: Speedups relative to the Standard Kalman Filter
7 Implementation

7.1 Introduction

While the final benchmarks will be carried out on the super-computers at UPPMAX, to ease in the development process it is desirable to be able to run and debug the program on an ordinary desktop computer first. To achieve this, a mixture of OpenMP, BLAS, LAPACK, Intel MKL and the C programming language is used.

LAPACK and BLAS To perform the matrix operations the well-known software library for numerical linear algebra LAPACK (Anderson et al., 1999) will be used. It contains routines for performing common linear algebra tasks. However, in this project, only the provided functions for performing matrix inversion are used. The BLAS library will be used for the rest of the operations appearing in the algorithms, e.g. matrix multiplications.

The BLAS library is specially designed to efficiently perform fundamental matrix operations such as $C = \alpha AB + \beta C$, and is also used by LAPACK internally. The efficiency of BLAS depends heavily on how good the BLAS library is optimized for the architecture in question. CPU vendors (e.g. Intel and AMD) provide BLAS libraries tuned to perform well on their platforms. During the development process on the desktop computer a generic, non-optimized version of BLAS was used.

BLAS and LAPACK was originally written in the programming language Fortran. Luckily there exist C interfaces, like for instance CLAPACK. Unfortunately the function interface is a bit impractical to use, so I have therefore written a simple interface for each (C)LAPACK and BLAS call that I need. This greatly improves the readability of the code.

Intel MKL Intel Math Kernel Library (MKL) (Intel) is a commercial software package that includes LAPACK and BLAS libraries tuned to perform well especially on Intel’s own CPUs. The library also takes advantage of multiple cores when possible. Fortunately, this feature can be inactivated. This will be needed when evaluating the different algorithms later, because we do not want MKL to interfere with our proposed parallelization strategies. However, it is not entirely necessary to inactivate this built-in feature since MKL will not parallelize anything if it detects that the program execution is within a OpenMP parallel section. In other words, this means that MKL calls within the sequential parts of the algorithms will be parallelized, while calls to MKL within the parallel parts will not be parallelized. This will still work, but to keep things simple and to be able to make theoretical predictions of the performance gain MKL will be set up to run single-threaded.

7.2 The Algorithms

7.2.1 Fusion-Gain

As mentioned, BLAS and LAPACK will be used to implement this algorithm. It is important to use the non-threaded versions of these libraries, since otherwise the computations may run in parallel. Normally this would be a good thing, but in this case the purpose is to make use of the parallelization strategy inherited in the Fusion-Gain algorithm itself.

Table 7.1, 7.2 and 7.3 show a possible implementation of the Fusion-Gain algorithm (compare with Table 6.1), written in a way that can be directly translated into LAPACK and BLAS calls.

When a new variable (vector or matrix) is needed for storage, the dimension of that matrix is written in the first column. This gives a hint of the memory requirement. All variables are allocated in the initialization part of the program, before the entrance of the simulation loop. Variables that are overwritten during function calls are colored red.
Table 7.1: Implementation approach for the Global Fusion step

<table>
<thead>
<tr>
<th>Global Fusion</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each local filter ((j=1,\ldots)):</td>
</tr>
<tr>
<td>([x_j, KH_j, KRKT_j] \rightarrow )</td>
</tr>
<tr>
<td>Local_Measurement_Update(xp, P)</td>
</tr>
<tr>
<td>This loop should be parallelized!</td>
</tr>
<tr>
<td>Return pointers to KH(_j) and KRKT(_j) and send the pointers to (x^-) and (P^-)</td>
</tr>
</tbody>
</table>

\[
\hat{x}(k|k) = (1 - N)\hat{x}(k|k - 1) + \sum_{j=1}^{N} \hat{x}_j(k|k) \\
x_0 = (1-N)\hat{x}_0 + x_0 \\
\text{For each local filter} (j=2,\ldots): \\
x_0 = x_j + x_0 \\
x_0 \text{ holds the resulting } x \text{ estimation} \\
K_{H,j} = \text{inv}(H_{j}(k)) \\
P(k|k) = \left[I - \sum_{j=1}^{N} K_{j}(k)H_j(k)\right]P(k|k - 1)\left[I - \sum_{j=1}^{N} K_{j}(k)H_j(k)\right]^T + \sum_{j=1}^{N} K_{j}(k)R_{j}(k)K_{j}^T(k) \\
KH_0 = I - KH_0 \\
\text{z is overwritten} \\
\text{For each local filter} (j=2,\ldots): \\
KH_0 = KH_0 - KH_j \\
KH_0 \text{ holds the sum } KH_j \\
KRKT_0 = KRKT_0 + KRKT_0 \\
KRKT_0 \text{ holds the sum } KRKT_j \\
KRKT_0 = t1Pp*KH_0' + KRKT_0 \\
P \rightarrow KRKT_0 \\
\text{Pointing (no flops)}

Table 7.2: Implementation approach for the Local Measurement-Update step

<table>
<thead>
<tr>
<th>Local Measurement-Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>(n_x \times x_j = x)</td>
</tr>
<tr>
<td>(K_j(k) = \frac{1}{\gamma} P(k</td>
</tr>
<tr>
<td>(n_z \times n_x )</td>
</tr>
<tr>
<td>(HP_j = H_j^*P )</td>
</tr>
<tr>
<td>(n_z \times n_z )</td>
</tr>
<tr>
<td>(HPHT_{R,j} = lti.R )</td>
</tr>
<tr>
<td>(HPHT_{R,j} = HP_j^*itti.H + \frac{1}{\gamma} HPHT_{R,j} )</td>
</tr>
<tr>
<td>(n_x \times n_z )</td>
</tr>
<tr>
<td>(n_x \times n_z )</td>
</tr>
<tr>
<td>(\hat{x}_j(k</td>
</tr>
<tr>
<td>(n_z \times n_x )</td>
</tr>
<tr>
<td>(z_j = -itti.H*\hat{x}_0 + z_j )</td>
</tr>
<tr>
<td>(x_j = K_{j}z + x_j )</td>
</tr>
<tr>
<td>(K_j(k)H_j(k) )</td>
</tr>
<tr>
<td>(n_x \times n_x )</td>
</tr>
<tr>
<td>(n_x \times n_x )</td>
</tr>
<tr>
<td>(K_j(k)R_j(k)K_{j}^T(k) )</td>
</tr>
<tr>
<td>(n_x \times n_z )</td>
</tr>
<tr>
<td>(n_x \times n_x )</td>
</tr>
</tbody>
</table>
Table 7.3: Implementation approach for the Global Time-Update step

<table>
<thead>
<tr>
<th>Global Time-Update</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \hat{x}(k+1</td>
</tr>
<tr>
<td>( n_x \times n_p )</td>
</tr>
<tr>
<td>( \text{xp} = \text{lti.F*x} )</td>
</tr>
<tr>
<td>( \text{xp} = \text{lti.G*u + xp} )</td>
</tr>
<tr>
<td>( n_x \times n_w )</td>
</tr>
<tr>
<td>( n_x \times n_x )</td>
</tr>
<tr>
<td>( n_x \times n_x )</td>
</tr>
<tr>
<td>( \text{Pp = FP*lti.F'} + \text{Pp} )</td>
</tr>
</tbody>
</table>

From Table 7.1 it is clear that this is the section that should be parallelized. For this the OpenMP statement “parallel for” can be used. The following code snippet shows how this may look like,

```c
#pragma omp parallel shared (z_arr, xp, Pp, lti_locals, wr_locals) private (i)
{
    #pragma omp for
    for (i = 0; i < n_locals; i++) {
        fusionGainLocalExec (z_arr[i], xp, Pp, &lti_locals[i], &wr_locals[i]);
    }
}
```

The “shared” and “private” clauses specify the scopes of the variables. As can be seen most of the variables (pointers in this case) are shared among the threads. This is perfectly valid since the pointers do not change their values. The iteration variable, \( i \), however, does change and should therefore not be declared as “shared”. Declaring \( i \) “private” ensures that each thread has its own copy of it.

### 7.2.2 Fast Givens Rotation

The Fast Givens method (Algorithm 11 and 12) together with the modifications proposed in Section 6.2 are used as the basis for the implementation. BLAS and LAPACK will be used for the matrix operations before and after the rotation. The rotation itself (Algorithm 12) does not use ordinary matrix operations at all, and is more about looping through the matrix and performing simple scalar operations on the matrix elements.

### 7.3 The Benchmark Software

The benchmark program will behave very similar to the MATLAB program used earlier in the analysis of the algorithms. A key difference is that this program will take advantage of multiple cores through the use of OpenMP. Also, a timer has been included that measures the speed of the algorithms for different simulation setups. A simulation setup is defined by e.g. the system model, the number of cores, initial values of the state vector \( (x_0) \) and the initial value of the state error covariance matrix \( (P_0) \).

The problem with a low-level language like C is that it easier to make mistakes during coding. To reduce the possibility of errors, all output from the program is written to data log files containing the computed state estimations together with the current simulation setup. It is then possible to compare this with the data output from the original MATLAB script. This approach has been taken to verify the correctness of the C implementation.
8 Preparations for the Final Run

8.1 Setting up the Test Cases

Main Test It is now time to stage the final test of the proposed algorithms. The most interesting part is perhaps to determine the speedup factor for different benchmark setups. The plan is to do this for a small system model consisting of 20 states, a medium sized model of order 500 and a large model of order 1000. The expected outcome of this test is to see that there will be little or no gain in applying the parallel algorithms to a small model, due to the overhead involved in the OpenMP fork-join process. As the model order increases the signs of the overhead will decay, and both of the parallel algorithms will hopefully perform better than the standard sequential Kalman filter. According to the outcome of the algorithm analysis in Section 6, the speedup will increase even further as the model order grows larger.

To see how the speedup is also affected by the number of system outputs, \( n_z \) will vary as 20\%, 50\% and 100\% of the model order \( n_x \).

For the “Fusion Gain” algorithm, different number of local subsystems will be tested to see how it affects the performance.

Since the computations will be performed on quad-core CPUs, the number of threads to test the algorithms on will be 1, 2, 3 and 4 respectively.

Cache Size Limit Additional tests will be carried out to investigate when the limited cache size becomes a bottleneck. For small problem sizes the Flops (“Floating-point operations per second”) capability of the CPU is assumed to be the limiting factor. But for large systems there will not be enough space in the cache to hold the same amount of data. To see when this happens, the idea is to notice when the experimental measurements start to diverge from the theoretical predictions, or in other words, when the time measure does no longer increase as \( O(n_x^3) \).

A Word on the Fusion Gain Algorithm The Fusion Gain algorithm has an extra variable that affects the performance noticeably, namely the number of subsystems to divide the full system in. To make things simple, the “natural” choice, of having the same number of subsystems as the number of cores has been selected. This means that when the speedup plot of the Fusion Gain algorithm itself is generated, the speedup values will be relative to the time it would take to run the algorithm with the same number of subsystems on a single-core CPU. This choice prevents the speedup plot from possibly give false hopes of greater than linear speedup, which otherwise would be possible as mentioned in the analysis of the algorithm (Section 6.1).

When comparing to the standard sequential Kalman algorithm the speedups will be relative to precisely the same standard sequential Kalman filter at all points in the plot.

8.2 Computer Environment

To get accurate time measures on how the algorithms perform, it is desirable to use a multi-core computer system with as few other running processes as possible. The computers at Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) fulfill this criteria perfectly. UPPMAX has three different multicore systems (Kalkyl, Grad and Halvan). The benchmarks will be carried out on Kalkyl, which is a cluster of 348 computer nodes, where each node has two Intel 5520 quad-core CPUs, each on with 8 MB of shared memory among the four cores. Because this work is focused towards the multicore architecture, only one quad-core CPU on one node will be used.

To run “jobs” at UPPMAX it is necessary to use a batch system called SLURM. Jobs can either be run in the background (“batch mode”) or in the foreground (“interactive mode”). Interactive jobs are
limited to run for at most one hour and typically starts right away. To run longer jobs or run more than one job simultaneously batch jobs have to be used. These are slightly more complicated to set up, so for this work everything have been run in interactive mode using the command,

```
interactive -A p2008031 --exclusive -N 1 -n 8 -t 1:0:00
```

This starts an interactive session on one compute node (“-N1”). “--exclusive -n8” guarantees sole access to an entire compute node.

All BLAS and LAPACK functionality has been provided by the Intel MKL library.

9 Final Results

9.1 Fusion Gain

The first results from the benchmark run are shown in Figure 9.1. All time measurements are decaying which is promising. For later comparison to the theoretical predictions Figure 9.2 is more informative. It tells that this method benefits from being run on a multi-core system which is in line with the predictions.

**Comparison with Sequential Kalman** The grand test is to see how the algorithms compare to the Standard Kalman Filter. Figure 9.3 shows that there is no use in choosing the Fusion Gain algorithm when the number of measured signals is small, since it is then slower than the Standard Kalman filter. At $n_z = 0.5n_x$, Fusion Gain becomes almost equal to the Standard Kalman filter performance wise. When all states are measured one can expect a speedup of about 1.5. The algorithm generally performs worse for small systems (order 20 in these cases) because of the overhead as mentioned earlier in the report.
Figure 9.2: Fusion Gain, experimental measurement

Figure 9.3: Fusion Gain, experimental measurement
9.2 Fast Givens

The time consumed by the Fast Givens algorithm is presented in Figure 9.4 and the self-speedup in Figure 9.5.

The experimental results make clear that it is meaningless to apply this implementation of the Fast Givens algorithm on small systems. The speedups for $n_x = 20$ are not just less than 1, but also get worse when increasing the number of cores.

For the larger systems ($n_x = 500$ and $n_x = 1000$) the speedup increases with the number of cores.

Comparison with Sequential Kalman Figure 9.6 illustrates how much faster Fast Givens is in relation to the Standard Kalman filter. Unfortunately, it shows that the Fast Givens approach is not nearly as fast as a Standard Kalman Filter in neither of the situations. The only thing that could possibly favor the Fast Givens algorithm is its better numerical stability.

9.3 Experimental versus Predicted Results

Fusion Gain With the benchmark results in hand, it is possible to evaluate how well the predicted speedups compare to the real ones. Figure 9.7 shows the self speedup of the Fusion Gain algorithm for $n_z = 0.5n_x$. The dashed lines are the predicted speedups as derived in Section 6.1.

It can be concluded that the theoretical predictions align almost perfectly with the measured speedups for $n_x = 500$ and $n_x = 1000$.

The small system ($n_x = 20$) actually performs significantly better than the predicted, which could be cache size related. The speedup numbers are computed as the time it takes to run the algorithm on just one core divided by the time it takes to run the same algorithm on more than one core. Because of the structure of the Fusion Gain algorithm the amount of data handled by each core decreases as
Figure 9.5: Fast Givens, experimental measurement

Figure 9.6: Fast Givens, experimental measurement
the number of cores increases. So a significant amount of cache misses are likely to occur in the single core case, which causes the speedups to grow.

**Fast Givens** The speedup plot for the Fast Givens method does not correspond as well with the theoretical prediction as in the Fusion Gain case (Figure 9.8). The experimental speedup numbers are somewhat smaller than the predicted ones.

The $n_x = 1000$ system lies closer to the theoretical value than the smaller system $n_x = 500$, and the $n_x = 20$ system especially. The reason for this could either be a bad theoretical analysis or, more likely, extra computation costs involved in the parallelization of the actual Fast Givens rotation, since it is the only part that is parallelized in the algorithm. More precisely, the overhead from the fork-join procedure could probably be blamed for this. As mentioned earlier in the report, the parallelizable work decreases for each iteration in the loop which leads to an increase in overhead.

**Comparison to the Standard Kalman Filter** If the speedup plots (Figure 9.3 and 9.6, for $n_x = 1000$) are compared to the predicted speedups in relation to the standard Kalman filter presented in Figure 6.7, it can be seen that the speedup curves for the Fusion Gain match up fairly well. The Fast Givens method on the other hand, is much slower in reality, e.g. 2.2 in theory and less than 0.2 in practice (4 cores, $n_x = 1000$, $n_z = 0.5n_x$). This suggests that the theoretical analysis of the number of floating-point operations per filter iteration is not perfect when it comes to modeling absolute execution time.
Figure 9.8: Fast Givens, Theory vs. Practice ($n_z = 0.5n_x$)
9.4 Intel MKL with Thread Support Enabled

Up to this point, only benchmark runs that could be supported by the theoretical analysis done in Section 6 have been considered. In real life situations however, where a fast estimator is needed, one would probably do even without such an analysis as long as it performs well. Previously in the report, it was mentioned that all benchmarks would be run with thread support disabled in the Intel MKL library. In this section benchmark results are presented with thread support enabled. This means that MKL routines such as matrix multiplication and inversion make use of parallelism internally.

The first result shows the self-speedup of the standard Kalman filter algorithm (Figure 9.9). It can be noticed that the speedup increases almost linearly with the number of cores when few states are observed. Then, as more states are observed the speedup drops, e.g. to 3 when 4 cores are used (for $n_z = n_x$). Still it performs much better compared to what was seen in earlier plots.

When MKL thread support is enabled and the execution reaches an OpenMP parallel section, none of the MKL calls therein will make use of threading. In the case with the Fusion Gain method, this means that MKL calls in the local measurement-update step (Table 7.2) do not take advantage of multiple cores. All cores are still utilized however, since each core runs its own local measurement-update step. During the two other steps (Global Time-Update and Global Fusion) MKL routines are threaded since they take place outside the parallel section.

When just considering the self-speedup plot (Figure 9.10) all three curves lie somewhere between the $n_z = 0.2n_x$ and $n_z = 1.0n_x$ curves of the standard Kalman filter (Figure 9.9).
It is not possible to tell which method is the faster by just looking at the self-speedup plots. Figure 9.11 shows how much faster the Fusion Gain algorithm is compared to the standard Kalman filter (with threading enabled). This plot reveals that the standard Kalman filter is faster even for $n_z = 0.5n_x$ (as opposite to the previous conclusion made from Figure 9.3). The standard Kalman filter apparently has more to win on enabling the threaded MKL library, but the Fusion Gain method is still faster in the many-outputs case.
Figure 9.11: Speedup for the Fusion Gain algorithm relative to the Standard Kalman Filter (threaded MKL)

10 Discussion

10.1 Conclusions

The results from the benchmark indicate that in some situations it could be advantageous to use the Fusion Gain algorithm instead of the Kalman filter implemented on standard form. These situations are when a large portion of the state variables are measured. When measured on a quad-core computer the speedup turned out to be about 1.5 for the tested systems of order 500 and 1000. For the system of order 20, the speedup was around 1.2. The lower speedup could be explained by increasing overhead time in relation to the total execution time.

If 50 % or fewer of the state variables were measured, there was no point in trading the standard Kalman filter for the Fusion Gain filter. Actually, because of the slightly worse numerical accuracy with Fusion Gain, the standard Kalman filter is definitely preferable.

The Kalman filter using the Fast Givens algorithm was a let-down. It did not perform nearly as good as it was supposed to according to the predictions. In all tested cases it was much slower than the standard Kalman filter. Its state estimates also got further away from the true states than both the Fusion Gain algorithm and the standard Kalman filter. This is natural because the algorithm in the original paper was constructed as a one-step-ahead predictor, i.e. it uses measurements from time $k$ to estimate the state vector at $k + 1$. However, the method should be more numerically stable compared to both the other filters, because it is based on a square-root algorithm.
Theoretical Predictions  The number of floating-point operations per filter iteration predictions derived in Section 6 were quite accurate when it came to predict the self-speedups for both algorithms, but were not perfect.

When comparing with the standard Kalman filter, the Fast Givens method performed significantly worse than predicted. This suggests that the theoretical Flops analysis has issues when it comes to modeling absolute execution time. In this particular algorithm implementation the number of pure floating point operations is quite small compared to the rest of the machine code, e.g. looping overhead and bad instruction pipeline usage. The Fusion Gain algorithm and the standard Kalman filter algorithm are more similar to each other in that they both spend most if the computation time in matrix algebra library routines (Intel MKL in this case).

The predictions were made solely on the number of Flops which explains why it could not perfectly predict the speedup in some cases. As mentioned in Section 9 other factors also have impact on the the actual speedup, for instance limited memory bandwidth, cache sizes and overhead. If those factors were modeled as well, the predictions would probably have been even more accurate.

Although not perfect, the theoretical analysis still serves the purpose of predicting how well the algorithms will work in reality. Without any type of theoretical analysis it would be difficult to know what to expect from the implemented algorithms.

As a side-note, the expression-to-Flops generator, mentioned earlier in the report, proved to be very useful in reducing the risk of human errors, when computing the Flops for each expression in the two algorithms (Fusion Gain and Fast Givens).

Threaded MKL  At the end of the previous section, benchmarks were run to determine how the built-in threading in the Intel MKL library would affect the results. The results showed that the standard implementation of the original Kalman filter, without any modifications, benefited greatly from multiple cores. The Fusion Gain algorithm also took advantage of the threading used in the MKL calls in the sequential sections of the algorithm. In the parallel section (Local Measurement-Update) threading within the MKL routines was disabled automatically.

The conclusion drawn from these tests is that the Fusion Gain algorithm is only to prefer (performance wise) over a standard implementation of the Kalman filter when having a very large output vector.

10.2 Future Work

Unsolved issues were discovered when simulating (Section 5.2) the distributed Kalman filter described in Section 3.1.1. It is unclear where the error source lies. This lead to that method, and the reduced-order method too, being discarded from this survey. If the problem could be solved, there might be a chance for these algorithms to become useful.

It is also doubtful whether this method is any faster than the one originally described in (Willner/Chang/Dunn, 1976). The authors of (Hashemipour/Roy/Laub, 1988) claim that their method is more efficient because it does not need to communicate the global estimate back to the local filters, but in a multi-core environment the cost might not be too big to still be of use. Besides, making sure that each local filter gets hold of the global estimate should lead to improved robustness of the algorithm. So, this method might also be worth to take a further look at.

To avoid numerical instability in real life applications some sort of square-root implementation is preferable. For the Fusion Gain algorithm to be useful in such cases, it should somehow be converted into square-root form.

The Fast Givens method deserves more attention. The theoretical prediction tells that this method consumes fewer number of floating-point operations per filter iteration than both the other methods. Still, it performs very bad according to the benchmarks.

It would be interesting to see if the theoretical computation time analysis could be improved, i.e. not only depend on the pure Flops usage.
References


Faxén, Karl-Filip et al.: Multicore computing - the state of the art. 2008


52