ECSE 444

Lab 1 Report

February 9th, 2023

Yue Lang Li

Simon Li

**Introduction**

The purpose of this report is to provide the results of experiments conducted on Kalman filter, an important algorithm in the field of signal processing and control. The experiments aim to illustrate the properties of the Kalman filter, such as its convergence towards the input stream and the statistical properties of the deviation from the input. Additionally, the report will compare the performance of different implementation methods of the Kalman filter, including assembly, plain C, and CMSIS-DSP.

Kalman filter is a recursive algorithm that uses Bayesian estimation to estimate the state of a system based on its past and present measurements, which has shown to be extremely useful in a number of real-world applications such as autonomous driving. The algorithm provides an optimal solution to the estimation problem, in the sense that the mean and covariance of the estimate are optimal, given the measurements and the underlying system dynamics. Significantly, the algorithm is widely used in fields such as control, navigation, and signal processing, where it is crucial to estimate the state of a system based on noisy and limited observations.

**Part 1: Assembly & C Implementation**

First the Kalman filter was implemented in assembly. R0 contains a pointer to the input measurements array, R1 a pointer to the output array, R2 a pointer to the Kalman state and R3 contains the length. A loop was implemented to repeatedly calculate the output values for each iteration. During each loop, the values in the Kalman state (i.e. p, k, x) are updated, and the values of R0 and R1 are incremented by 4 to point to the next value in the array. At each iteration, division by zero and overflow are checked by moving the values of the floating point status register (FPSCR) to the application program status register (APSR) to account for edge cases. In cases of error, we branch to the error handler where an error code of -1 is returned. At the end of each iteration, the APSR flag bits are cleared.

Secondly, the same operations were implemented in plain C. The function has the same signature as the assembly one, and during each iteration, we update the prediction error covariance, calculate the Kalman gain, update the estimate of the state, and finally write and update the estimate of the state (x) to the output array. To account for edge cases, we check if the output value is NaN or INF and return -1 if either is true.

Finally, we re-implemented the algorithm once again but using the DSP library this time. The arm math library contains in-built functions that are optimized for vector and floating-point operations. In our implementation of the Kalman filter, we used the functions arm\_add\_f32, arm\_sub\_f32 and arm\_mult\_f32. Interestingly, no division function were found in the library. Therefore we used the normal division operation in C to calculate and update the prediction error covariance. This could have a significant effect on the performance of the DSP implementation. In the next part, the performance of the 3 implementations will be analyzed and compared.

**Performance of Implementation**

First we used the Instrumentation Trace Microcell (ITM), which is designed to add timestamps to trace events. If we create a timestamp before and after the function call, we can approximate the execution time of the Kalman filter function. We believe that ITM\_Port32(n) is a location in memory, and that setting it to a value will generate a trace packet with that value as the data. This also generates a timestamp in terms of elapsed cycle count and wall-clock time. We executed 101 calls to the function using the provided input measurement array with 101 values and recorded a time of [] seconds per call.

Then, the Kalman filter was run and a trace was recorded to illustrate its properties, specifically the convergence towards the input stream and the statistical properties of the deviation from the input. The recorded trace showed a decrease in deviation from the input over time, demonstrating the effectiveness of the Kalman filter in reducing error.

Subsequently, the difference between the input stream and the output from the Kalman filter was explicitly calculated. The results showed a standard deviation of 10.069, and an average of -0.025. The correlation and convolution between the two streams were also calculated and further confirmed the accuracy of the Kalman filter in tracking the input stream.

The core Kalman filter code was rewritten in plain C. It is worth mentioning that this implementation is [insert value]% slower than the assembly implementation, with an execution time of [x] compared to [y] for the assembly implementation.

Using CMSIS-DSP, the same functions performed in 2 were calculated.

Using code profiling, the running times for all the procedures with our code and CMSIS-DSP were reported. The results were [insert value]% faster than the previous implementation, with running time measured at [insert numerical value].

**Conclusion**

After analyzing the profiling data, [insert numerical value] was observed for the running time using assembly, [insert numerical value] using plain C and [insert numerical value] using CMSIS-DSP. Based on these results, it can be concluded that CMSIS-DSP is the most efficient option for implementing the Kalman filter code. However, it is important to consider the scalar version of CMSIS-DSP when necessary.

The code was run on the actual processor and the debugger was used to inspect and modify the program variables. It was found that the following variables can be watched and modified: [insert variables]. These variables can be modified without stopping the processor by [insert method]. Overall, the use of the debugger allowed for more efficient debugging and development of the Kalman filter code.

In conclusion, the experiments performed aimed at exploring the properties and performance of Kalman filter. The results showed the convergence of the filter towards the input stream and the statistical properties of the deviation to the input. The difference to the input stream, standard deviation, average, correlation and convolution between the two streams were calculated both through a custom program and CMSIS-DSP. The code profiling results indicated a faster performance when using CMSIS-DSP. The core Kalman filter code was rewritten in plain C and in C using CMSIS-DSP and the profiling data compared the three versions of the filter core. The lessons learned from the use of assembler, C and CMSIS-DSP were summarized by selecting the suitable profiling data. Finally, the code was run on an actual processor and the debugger was used to inspect and modify the program variables without stopping the processor.

The results obtained from the experiments provide valuable insights into the properties and performance of Kalman filter. The faster performance when using CMSIS-DSP highlights the significance of libraries and tools in improving the efficiency of code. The ability to modify program variables while the code is running without stopping the processor is also a significant finding.

In the future, these results could be used as a reference for further research and development in the field of Kalman filter and its applications in fields such as autonomous driving. The lessons learned on the use of assembler, C and CMSIS-DSP can also be applied to other projects, leading to more efficient and optimized code. Overall, we believe that the research has significant implications for the application of Kalman filter in various fields.

**References**:

* None