Lab 1: Kalman Filter using Floating-Point Assembly Language Programming and its Evaluation with C and CMSIS-DSP

Kaicheng Wu
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
kaicheng.wu@mail.mcgill.ca
260892789

Tian Feng
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
tian.feng@mail.mcgill.ca
260913386

***Abstract*—In this lab, we implemented the Kalman filter using Arm Assembly, standard C, and the CMSIS-DSP library on a STM32 board and evaluated the performance of each one of them. The experiment involved coding the Kalman filter algorithm in Arm Assembly, standard C, and utilizing the CMSIS-DSP library. To evaluate their performances, the code was profiled, and execution times were measured.**

***Index Terms*—Kalman Filter, Arm Assembly, CMSIS-DSP, STM32**

I. Introduction

The Kalman filter is an adaptive estimator for modeling the state of a system. It minimizes errors, especially in linear systems with Gaussian noises. Its adaptive nature ensures that the filter parameters evolve with each running cycle, thereby enhancing its accuracy and effectiveness by converging towards the input stream.

For this experiment, we will be developing a single variable Kalman filter with Arm Assembly, standard C, and the CMSIS-DSP library, respectively. We will then calculate the following properties for them: the difference between the original and data obtained by Kalman filter tracking, the standard deviation and the average of the difference, the correlation between the original and tracked vectors, and the convolution between the two vectors. We will also compare their performances and utilities via code profiling.

II. Implementation

A. Standard C

The implementation of the Kalman filter in standard C is a relatively straightforward task.

The initial step involves defining a structure called Kalman, which encapsulates essential parameters: namely, q (representing the process noise variance), p (representing the estimation error covariance), r (representing the measurement noise variance), k (representing the Kalman gain), and x (representing the input value). This structured approach enables efficient handling of the Kalman filter's core elements.

With each iteration, these parameters undergo updates to refine the filter's performance. Specifically, p evolves by incorporating the process noise variance ($p + q$), while k dynamically adjusts based on the covariance and measurement noise ($\frac{p}{p+r}$). Furthermore, the input value x is refined through a weighted combination of the current value and the disparity between the measurement and the current estimate ($x + k \cdot (measurement - x)$). Finally, the estimation error covariance p is recalibrated to reflect the refined estimates ($(1 - k) \cdot p$). This iterative process ensures that the Kalman filter effectively balances noise and measurement uncertainties to yield accurate and reliable estimates.

B. ARM Assembly

The assembly code for the Kalman filter follows the same principle as the standard C code.

First, we load the values of q, p, r, k, and x from memory into the respective single-precision floating-point registers. Then, we compute the parameters the same way we did with the standard C code. We first add the process noise variance q to the estimation error covariance p to obtain a new value for p. Then, we compute a temporary sum of p and the measurement noise variance r. Using this sum, we calculate the Kalman gain k by dividing p by the temporary sum. Next, we compute the difference between the measurement and the current estimate ($m - x$) and multiply it by the Kalman gain k, storing the result in a temporary register. This result is then added to the current value of x, updating it. Additionally, we recalculate a portion of p by multiplying p with k and subtracting it from the current p value. Finally, the updated parameter values are stored back into memory, and the program returns.

C. CMSIS-DSP

The CMSIS-DSP implementation is also very similar to standard C.

We kept using the same Kalman structure created for standard C. The only difference is that this time instead of the

standard C arithmetic operations, we use the following functions: arm_add_f32 for addition, arm_sub_f32 for subtraction, and arm_mult_f32 for multiplication. Since there is no division for float32_t, we simply used the conventional / division for C.

D. Result Analysis Functions

We implemented a function labeled KalmanFilter, which calls the updateKalmanFilter functions implemented repetitively over an array to filter them.

We also implemented a series of functions in C to process the filtered data. Namely, *calculateDifference* which computes the difference between each element in the input array and its counterpart in the filtered array, *calculateAverage* which computes the average of the difference obtained previously, *CalculateStd* which computes the standard deviation of an array, *calculateConvolution* which computes the convolution of the input and the filtered array and finally *calculateCorrelation* which computes the correlation between the input and the filtered array.

All of the aforementioned functions have built-in counterparts in CMSIS-DSP, which we called to cross-validate our C implementations.

E. Error Checking

The program can encounter numerous errors, namely division by zero and float overflow. These errors must be dealt with accordingly. We check for these errors by implementing overflow-checking functions in the file that check the correctness of the result after any risky action. In the assembly code, fcspr registers are redirected to aspr registers using VMRS instruction to enable conditioning for vector operations.

III. Results

By tracing the execution of the algorithms, we could make the following observations.

A. Convergence property of the Kalman filter

By running the Kalman filter on the input vector, we could observe the convergence towards the input. Towards the end of the output array, all entries have reached a point where they became virtually identical. The standard deviation dropped from 0.3.7 for the input, to 0.256 for the output

For the longer array given for testing, see the results in the *results.csv* file.

B. Other analysis of the filtered data

Furthermore, we analyzed the output of the Kalman filter by computing its difference with the input, the average and the standard deviation of the difference, the correlation of the output with the input, and the convolution between the two.

For the same input array used previously, we got an average difference of 0.025(0.025 for CMSIS), a standard deviation of 0.257(0.258 for CMSIS), and a correlation of 0.675(0.669 for CMSIS). The outputs we got from standard C and the implementation using CMSIS-DSP are almost identical. A difference of less than 2

C. Python verification

We used collab to check the correctness of the statistical data. The data are all correct referring to the .ipynb file in the root folder.

D. Execution time of different implementations

We also profiled our code to compare the execution time(in clock cycles of the three implementations of the Kalman filter. We received timing packets from the SWV trace log redirected from ITM.

TABLE I
Execution time of Kalman Filter@80Mhz w/BL45IIOT01A

	ARM Assembly	Plain C	CMSIS-DSP
Execution time (clock cycle)	10064	45967	98655

Interestingly, the implementation in plain C performs significantly worse than ARM Assembly but outperforms CMSIS-DSP by 100

TABLE II
Execution time of other functions @80Mhz w/BL45IIOT01A

	Plain C	CMSIS-DSP
Difference (clock cycle)	3758	874
Average (clock cycle)	2467	543
Standard Deviation (clock cycle)	108860	946
Correlation (clock cycle)	226633	8781
Convolution (clock cycle)	841087	68695

For the analysis functions on the filtered data, as expected, the CMSIS-DSP library outperforms plain c by quite a large margin, especially in the case of convolution with larger arrays, where the nested loops create a huge number of operations.

IV. Conclusion

To conclude, we implemented the Kalman filter using Arm Assembly, standard C, and the CMSIS-DSP library on an STM32 board and evaluated the performance of each one of them. We learned about the Kalman filter's basic properties while also experimenting with the functionalities of CMSIS-DSP. The high performance and utility of the CMSIS-DSP gave us a new appreciation for the library. Kaicheng was responsible for implementing the C code and the report whereas Tian was responsible for implementing and testing the assembly and CMSIS code.