ECSE 444 Lab 1 Report

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

Group 11: Byron Chen, 260892558, Kevin Li, 260924898

***Abstract*—In this lab, the performance of the Kalman Filter is found to be increasing as more inputs are entered. This aspect is obtained by studying the average, standard deviation, correlation and convolution of the differences between the input and filtered data. Apart from this, the Kalman Filter is implemented in both Assembly and C code, and their performances are compared by precision and time used, and Assembly is found to be better than C code. Furthermore, to compare the performances between C and CMSIS-DSP, the data processor is implemented in both C and CMSIS-DSP. It is found that C takes more time while processing the data, and it is less precise than CMSIS-DSP. Apart from this, how the debugger works on STM32 is also investigated and a way of changing the variables when debugging is also found out.**

***Index Terms*—Kalman Filter, Floating-Point Assembly, C, CMSIS-DSP, STM32**

I. Introduction

KALMAN Filter is a viral filter used for estimating the internal state of a linear dynamic system by inputting the measurements. In this lab, we are aiming to find the properties by processing the differences between the input and filtered data by using STM32. The differences obtained will be used to calculate the average, standard deviation, correlation, and convolution for further studying. Apart from this, the performances of Assembly and C are compared by analyzing the precision and time taken for running the Kalman Filter implemented in both languages. Moreover, the performances of C and CMSIS are also compared by analyzing the precision and time taken for running the data processor for the differences implemented in both ways. At last, we investigated how the debugger works on STM32 by finding out how to check the variables and modify them while debugging.

II. Materials and Methods

In this lab, to compare the performance in different programming languages, the Kalman Filter is implemented using both Assembly code and C code on STM32.The main function, namely update, is shown below in C code:

```
p = p + q;
k = p / (p + r);
x = x + k * (measurement - x);
p = (1 - k) * p;

return x;
```

In the code shown above, q stands for process noise covariance; r stands for measurement noise covariance; x stands for estimated value; p stands for estimation error covariance; k stands for adaptive the Kalman filter gain; measurement is the input to be entered, and all the variables are floating-point numbers [1]. This function supposes to return 0 if the filter works out correct answers and returns -1 if any exceptions happen. To carry the variables easily throughout the calculations, $(q,\ r,\ x,\ p,\ k)$ is stored in a C structure, namely $kalman\_state$.

In order to implement the Kalman Filter in Assembly code, namely $kalman\_asm$, structure $kalman\_state$ are changed into an input array stored in R0 with an extra element to store the return value (as mentioned above) from $kalman\_asm$. Apart from this, measurement is stored in S0 as the second input to $kalman\_asm$. In $kalman\_asm$, to fetch the inputs from the array, VLDR is used as shown below:

```
VLDR S1,[R0]
VLDR S2,[R0,#4]
VLDR S3,[R0,#8]
VLDR S4,[R0,#12]
VLDR S5,[R0,#16]
```

However, two exceptions, overflow and divided by zero, might happen during the calculations and cause wrong outputs. In order to avoid this from happening, both C code and Assembly code include the check for the exceptions.

In C code, the maximum number that can be represented, FLT_MAX, is used to test if any overflow happens during the calculations. Any number that is bigger than FLT_MAX implies an overflow. Furthermore, we found only one division during the calculations, which is $k = p/(p + r)$. As a result, $p + r$ is compared with zero before calculating this line, and if any exception happens, the function $update$ will return -1.

In $kalman\_asm$, the register FPSCR is used to check if any overflow happens. At the beginning of the code, the overflow bit is reset to zero by the codes shown below:

```
VMRS R1,FPSCR
AND  R1,R1,0xFFFFFFFB
VMSR FPSCR,R1
```

Since the overflow bit is the third bit in FPSCR, a mask 0xFFFFFFFB is used to reset FPSCR. During the calculations, if the overflow bit in FPSCR is changed to 1, the function will return -1. And the codes that check the overflow is shown below:

```
VMRS R1,FPSCR
AND  R1,R1,#4
CMP  R1,#4
```

To check the exception caused by divided by zero, $p + r$ is compared with zero before doing the division and it is done by the following codes:

```
VCMP.F32  S5,#0.0
VMRS  APSR_nzcv,FPSCR
BEQ   exception
```

Here, S5 stores the value of $p + r$ and the second line is done in order to load the FPSCR flags into APSR, which will ensure BEQ works as expected. Apart from this, $exception$ is the label for returning the -1. As mentioned before, the return value is stored in the same array as the inputs, which is implemented by storing the return value into the address [R0, #20].

To implement the Kalman Filter completely, functions $KalmanFilter\_C$ and $almanFilter\_asm$ are implemented to loop the Kalman Filter's main function in both C and Assembly, respectively. The functions take an array of measurements, a structure $kalman\_state$, and the length of the array as inputs and output an array of x's after each update. The return values of the functions represent if the Kalman Filter is working correctly without any exception.

In order to find out the properties of the Kalman Filter, the filtered data is processed with the following steps in both C and C with CMSIS-DSP [1]:

1) Subtraction of original and data obtained by the Kalman Filter tracking.
2) Calculation of the standard deviation and the average of the difference obtained in 1).
3) Calculation of the correlation between the original and tracked vectors.
4) Calculation of the convolution between the original and tracked vectors.

Subtraction is done by calculating the difference between the input array and output array, respectively representing the measurement and the value "x" stored in the $kalman\_state$ after each update. The implementation in C is to create an array with the same input and output array size, calculate the difference of each element of two given arrays, and store the absolute value of the calculated difference to the corresponding position of the created array. In CMSIS-DSP, the function arm_sub_f32 [2] is used. This method calculates the difference between two vectors. After calculating the differences in CMSIS, the absolute values of the calculated elements are stored.

Average is calculated with the formula 1.

$$\frac{\sum_{i=0}^{N-1} a[i]}{N} \tag{1}$$

Where N is the length of the array to be calculated. In C, the implementation is done by getting the sum of the given array using a for loop and divide the sum by the length. In CMSIS-DSP, the function arm_mean_f32 [2] is used.

Standard deviation is calculated with the formula 2.

$$\sqrt{\frac{\sum_{i=0}^{N-1}(a[i] - mean)^2}{N}} \tag{2}$$

Where N is the length of the array to be calculated. In C, the implementation is done by using a for loop. In CMSIS, the function arm_std_f32 [2] is used.

Correlation is calculated with the formula 3 [3].

$$correlation[n] = \sum_{i=0}^{length} (a[i] * b[i - n]) \tag{3}$$

Where the correlation array will be an array with the size of $max(lengthA, lengthB) * 2 - 1$. Figure 1 (Appendix D) illustrates how correlation is calculated. Implementation of correlation in C requires a nested loop. The outer loop with index i is used to iterating through the correlation array. The inner loop with index j is used to iterate the first input array from 0 to i. The index for the second array is initialized with $length - 1 - j$ and will increment after every inner loop. The element of the first array will be multiplied by the element in the second array according to their respective index. If any index of the arrays exceeds the boundary (their length), the result of the multiplication will be 0. During the iteration of the inner loop, the results of the multiplication will be added together. After iterating the inner loop, the sum will be put in the corresponding position of the correlation array and iterates the next outer loop. In CMSIS, the function $arm\_correlate\_f32$ [2] is used.

Convolution is calculated with the formula 4 [3].

$$convolution[n] = \sum_{i=0}^{length} (a[i] * b[n - i]) \tag{4}$$

Where the correlation array will be an array with the size of $max(lengthA, lengthB) * 2 - 1$. Figure 2 (Appendix D) illustrates how correlation is calculated. Implementation of convolution in C is almost the same as that for correlation. It is easy to see that the only difference between the calculations of correlation and convolution is that the index for the second array is changed from $i - n$ to $n - i$. As a result, the inner loop with index j is changed to iterate the first input array from i to 0, and the initial value of the index for the second array is changed to 0. All other implementations in C remain the same. In CMSIS, the function arm_conv_f32 [2] is used.

To compare the performance between the Kalman Filter implemented by C and the one implemented by Assembly, ITM_Port32 is used, along with breakpoints and SWV Trace Log, to get the time used to run them. Furthermore, the differences between the answers from different functions are also calculated using the function $get\_difference$.

Apart from this, the performances of the data processor implemented in C and the one in C with CMSIS-DSP are also compared following the same steps as described above.

III. Results

The following data are obtained by inputting the $TEST\_ARRAY$ (Appendix A and Figure 3) with length of 101. Furthermore, (q, r, x, p, k) is set to be $(0.1, 0.1, 5, 0.1, 0)$. All the implementations and the tests are covered in the attached zip file namely "codes".

First of all, to figure out the properties of the Kalman Filter, filtered data (shown in Appendix B and Figure 4) is processed by the steps given in section 2. Since the results of the subtraction, correlation, convolution is in vectors, and

Process	Result
Average	0.130961165
Standard Deviation	0.194331005

TABLE I
THE AVERAGE AND STANDARD DEVIATION OF THE DIFFERENCES BETWEEN ORIGINAL DATA AND FILTERED DATA CALCULATED BY CMSIS.

they are shown in the Appendix C and D. The results of the average and standard deviation are shown in Table 1.

After tracing the Kalman Filter, the performances of the Kalman Filter implemented by C and the one implemented by Assembly are compared. The differences between the outputs are 0, which implies that they all get the same values by processing the Kalman Filter. The time used by both implementations is shown in Table 2.

Language	Time Used
C	0.2956 ms
Assembly	0.1855 ms

TABLE II
TIME USED BY RUNNING THE KALMAN FILTER IN BOTH C AND ASSEMBLY.

To compare the performances of the data processor implemented in C and the one in C with CMSIS-DSP, their differences in data and the time used are calculated. In order to make the data more readable, the mean of the differences between the vectors for the same step get from C and CMSIS are shown in Table 3. Furthermore, the time used by the different steps with different implementations is shown in Table 4.

Process	Average Difference
Subtraction	0.000000000000
Average	0.000000000000
Standard Deviation	0.000964432955
Correlation	0.000000000000
Convolution	0.000674878771

TABLE III
AVERAGE DIFFERENCE BETWEEN THE RESULTS FOR DIFFERENT PROCESSES FROM C AND CMSIS.

Process	Language	Time Used
Subtraction	C	0.038359 ms
Subtraction	CMSIS	0.008058 ms
Average	C	0.024517ms
Average	CMSIS	0.004267ms
Standard Deviation	C	0.128175ms
Standard Deviation	CMSIS	0.007566ms
Correlation	C	9.019975ms
Correlation	CMSIS	0.586359ms
Convolution	C	6.366958ms
Convolution	CMSIS	0.585558ms

TABLE IV
TIME USED BY DIFFERENT PROCESSES IN C AND CMSIS.

At last, to check if the Kalman Filter implementation could deal with exception correctly, the structure $kalman\_state$ is assigned to be all 0 and all FLT_MAX, which test divided by zero and overflow separately. The Kalman Filter functions return -1 after running the data, showing that the implementation can correctly deal with the exceptions.

IV. DISCUSSION

In Table 1, we can see that the Kalman Filter performs well when predicting the next signal. Both the accuracy and the precision is held to be very high. By observing the differences between the inputs and the filtered data in Figure 5 (Appendix D), it is easy to see that the Kalman Filter begin to work properly right after entering few signals. This shows that the Kalman Filter is an efficient filter. Furthermore, the results from correlation and convolution (in Figure 6 and 7) also show that the inputs and the outputs tend to be more similar.

In Table 2, we can see that the running time for the Kalman Filter in Assembly is nearly twice faster than that in C, whereas they are performing with the same precision.

In Table 3, C has some inaccurate results when calculating standard deviation and convolution. This is caused by the rounding errors during the calculations. In subtraction, the numbers to be subtracted are not very close to each other. When calculating the average, all numbers are added and divided by an integer which will not cause any rounding error. However, when calculating the standard deviation, all the numbers are subtracted by the average calculated before, which can cause cancellation errors. Furthermore, denormalization might happen during the calculation since the subtracted numbers are powered by 2 and added together. This might also cause rounding errors. For correlation and convolution, the calculations in them are almost the same, but the arrangement between elements is different, which will be the reason to cause rounding errors by denormalization.

V. SUMMARY

In conclusion, the Kalman Filter is an efficient filter used for estimating the internal state of a linear dynamic system by inputting the measurements. Furthermore, the precision of calculations in Assembly and C is the same, whereas Assembly can perform them faster than C. Apart from this, both the precision and time were taken of calculations in CMSIS are found to be better than that in C. As a result, SMSIS should be mainly used when developing on STM32, and Assembly could be chosen for implementing the functions when CMSIS cannot do the job.

When using the debugger on STM32, there are two ways to monitor the values of the variables. The first method is putting the mouse on the variable for inspecting. The second way is to check the variables table. It is necessary to remember that this only works when the debugger already processes the variables. Furthermore, the variables can also be modified using the variables table and registers table without stopping the processor.

VI. WORK DISTRIBUTION

Both Byron and Kevin participated all parts through the lab. Byron is mainly responsible for implementing the Kalman Filter and analyzing the performance of all the parts; while Kevin is mainly responsible for implementing the filtered data processes and investigating the property of the Kalman Fliter.

APPENDIX A
TEST ARRAY

10.4915760032, 10.1349974709, 9.53992591829, 9.60311878706, 10.4858891793, 10.1104642352, 9.51066931906, 9.75755656493, 9.82154078273, 10.2906541933, 10.4861328671, 9.57321181356, 9.70882714139, 10.4359069357, 9.70644021369, 10.2709894039, 10.0823149505, 10.2954563443, 9.57130449017, 9.66832136479, 10.4521677502, 10.4287240667, 10.1833650752, 10.0066049721, 10.3279461634, 10.4767210803, 10.3790964606, 10.1937408814, 10.0318963522, 10.4939180917, 10.2381858895, 9.59703103024, 9.62757986516, 10.1816981174, 9.65703773168, 10.3905666599, 10.0941977598, 9.93515274393, 9.71017053437, 10.0303874259, 10.0173504397, 9.69022731474, 9.73902896102, 9.52524419732, 10.3270730526, 9.54695650657, 10.3573960542, 9.88773266876, 10.1685038683, 10.1683694089, 9.88406620159, 10.3290065898, 10.2547227265, 10.4733422906, 10.0133952458, 10.4205693583, 9.71335255372, 9.89061396699, 10.1652744131, 10.2580948608, 10.3465431058, 9.98446410493, 9.79376005657, 10.202518901, 9.83867150985, 9.89532986869, 10.2885062658, 9.97748768804, 10.0403923759, 10.1538911808, 9.78303667556, 9.72420149909, 9.59117495073, 10.1716116012, 10.2015818969, 9.90650056596, 10.3251329834, 10.4550120431, 10.4925749165, 10.1548177178, 9.60547133785, 10.4644672766, 10.2326496615, 10.2279703226, 10.3535284606, 10.2437410625, 10.3851531317, 9.90784804928, 9.98208344925, 9.52778805729, 9.69323876912, 9.92987312087, 9.73938925207, 9.60543743477, 9.79600805462, 10.4950988486, 10.2814361401, 9.7985283333, 9.6287888922, 10.4491538991, 9.5799256668

APPENDIX B
FILTERED DATA

8.6610508, 9.58226776, 9.55605602, 9.58514977, 10.1418571, 10.1224546, 9.74435043, 9.75251293, 9.7951746, 10.1013975, 10.3391771, 9.86578465, 9.76877975, 10.1810875, 9.88773918, 10.1246004, 10.0984659, 10.220212, 9.81916523, 9.7259388, 10.1747732, 10.3317232, 10.2400331, 10.095767, 10.2392616, 10.3860197, 10.3817406, 10.2655506, 10.1211443, 10.351531, 10.2814798, 9.8584671, 9.71577072, 10.0037298, 9.78946209, 10.160965, 10.1197004, 10.0056438, 9.82303143, 9.95118523, 9.99207783, 9.80552387, 9.76442814, 9.61660385, 10.0556984, 9.74127865, 10.1220598, 9.9772377, 10.0954466, 10.1405153, 9.98202133, 10.1964703, 10.2324724, 10.3813381, 10.1539373, 10.3187246, 9.94458389, 9.91122818, 10.0682373, 10.1855755, 10.285059, 10.0992813, 9.91045856, 10.0909615, 9.93503761, 9.91049671, 10.1441193, 10.0411348, 10.0406761, 10.1106472, 9.90817261, 9.79447174, 9.66882706, 9.97956467, 10.1167793, 9.98682022, 10.1959095, 10.3560438, 10.4404249, 10.2639103, 9.85697269, 10.2324247, 10.232564, 10.2297249, 10.3062391, 10.2676134, 10.3402567, 10.0730133, 10.0168152, 9.71457958, 9.70139027, 9.84260082, 9.77881241, 9.67166042, 9.74851131, 10.2099276, 10.2541218, 9.97254944, 9.76009369, 10.185956, 9.811409

APPENDIX C
RESULTS FROM PROCESSING THE FILTERED DATA IN CMSIS

SUBTRACTION:

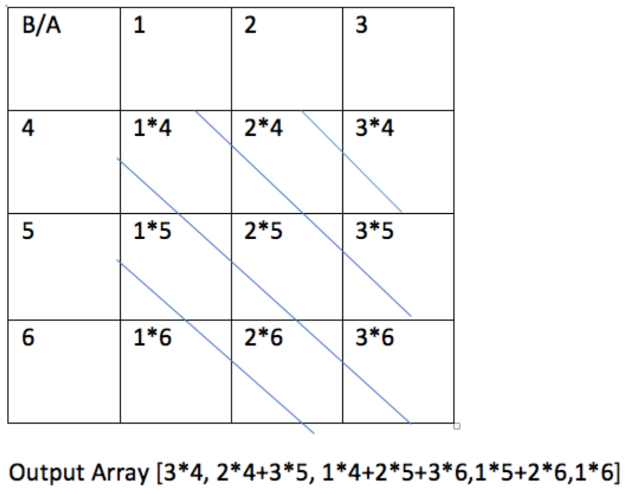
1.8305254, 0.552729607, 0.0161304474, 0.0179691315, 0.344032288, 0.0119905472, 0.233680725, 0.00504398346, 0.0263662338, 0.189256668, 0.14695549, 0.292572975, 0.0599527359, 0.25481987, 0.18129921, 0.146389008, 0.0161514282, 0.0752439499, 0.247860909, 0.0576171875, 0.277394295, 0.0970010757, 0.0566682816, 0.0891618729, 0.088684082, 0.0907011032, 0.00264453888, 0.0718097687, 0.0892477036, 0.14238739, 0.0432939529, 0.261436462, 0.0881910324, 0.177968025, 0.132424355, 0.22960186, 0.0255022049, 0.0704908371, 0.11286068, 0.079202652, 0.0252723694, 0.115296364, 0.0253992081, 0.0913600922, 0.271374702, 0.194322586, 0.235336304, 0.0895051956, 0.0730571747, 0.0278539658, 0.0979547501, 0.132535934, 0.0222501755, 0.0920038223, 0.14054203, 0.101844788, 0.231231689, 0.020614624, 0.0970373154, 0.0725193024, 0.0614843369, 0.114817619, 0.116698265, 0.111557007, 0.0963659286, 0.0151672363, 0.144387245, 0.0636472702, 0.000284194946, 0.0432443619, 0.125136375, 0.0702705383, 0.0776519775, 0.192047119, 0.0848026276, 0.0803194046, 0.129223824, 0.0989685059, 0.0521497726, 0.109092712, 0.251501083, 0.232042313, 8.58306885e-005, 0.00175476074, 0.0472888947, 0.0238723755, 0.0448961258, 0.165164948, 0.0347318649, 0.18679142, 0.00815105438, 0.087272644, 0.0394229889, 0.0662231445, 0.0474967957, 0.285171509, 0.0273141861, 0.174020767, 0.131304741, 0.263197899, 0.231483459

CORRELATION:

102.937141, 206.305328, 299.233521, 394.939667, 502.462524, 605.915161, 697.987, 792.006348, 888.593506, 990.077759, 1091.5481, 1184.30762, 1281.73352, 1387.31689, 1488.1416, 1592.64514, 1696.57141, 1801.91882, 1900.31946, 1998.88684, 2101.66089, 2208.96167, 2315.32349, 2418.62036, 2523.69458, 2628.67163, 2733.75537, 2835.24854, 2931.92822, 3034.90234, 3136.60303, 3234.06836, 3330.17847, 3432.15845, 3530.96924, 3633.32422, 3733.86353, 3834.09644, 3930.89648, 4032.47266, 4135.43896, 4234.2334, 4332.09229, 4426.73779, 4529.45605, 4627.9585, 4732.60889, 4835.95605, 4939.43018, 5043.39404, 5141.78418, 5246.61523, 5350.03418, 5453.73828, 5555.229, 5656.44092, 5754.80029, 5849.4248, 5948.90381, 6049.0835, 6153.17432, 6252.58252, 6348.75, 6450.70703, 6550.56201, 6651.14404, 6751.75635, 6852.13281, 6949.51221, 7050.3833, 7150.93457, 7251.63232, 7348.21875, 7452.81543, 7559.32178, 7661.99414, 7766.82812, 7872.20996, 7979.35889, 8083.74365, 8182.41064, 8283.38477, 8384.11035, 8488.46289, 8593.09375, 8696.4043, 8798.91406, 8900.42285, 8997.54785, 9092.0957, 9192.25488, 9293.42871, 9388.15332, 9481.62207, 9577.07324, 9684.53516, 9788.79102, 9881.81934, 9974.07422, 10076.2959, 10161.5312, 10053.6885, 9950.61426, 9857.72754, 9761.51953, 9654.25293, 9550.75098, 9458.19434, 9363.74219, 9267.22559, 9165.06055, 9064.79395, 8970.84863, 8873.10059, 8768.33691, 8667.56836, 8562.72461, 8458.65332, 8354.53809, 8255.88477, 8156.33057, 8052.58936, 7946.89502, 7840.82129, 7736.93164, 7632.2168, 7527.5752, 7422.26172, 7320.87354, 7223.97412, 7121.62842, 7020.89404, 6922.79639, 6825.66553, 6723.42334, 6624.74902, 6522.22314, 6422.06738, 6322.16943, 6225.40771, 6124.07764, 6021.09082, 5922.88672, 5824.56348, 5729.51855, 5627.07666, 5527.93213, 5423.44678, 5321.05273, 5216.88184, 5113.35889, 5014.24756, 4909.35742, 4806.17188, 4701.72412, 4600.73779, 4500.07373, 4401.73877, 4306.04297, 4206.875, 4106.17383, 4003.30713, 3903.63232, 3806.59106, 3705.20557, 3605.25366, 3504.66284, 3403.60352, 3304.32812, 3207.08862, 3106.95312, 3006.26685, 2905.50635, 2807.83228, 2703.479, 2597.56128, 2493.7395, 2388.33154, 2283.19971, 2176.26782, 2071.68506, 1973.13892, 1871.78809, 1770.73743, 1666.20911, 1562.13513, 1459.11353, 1356.42297, 1255.29138, 1158.55383, 1064.49146, 964.035156, 863.630371, 767.992798, 674.356018, 578.292908, 472.268372, 368.808899, 275.068329, 182.298065, 82.9722214

CONVOLUTION:
90.8680725, 188.312805, 280.000122, 372.00116, 477.552063, 580.243591, 673.081848, 767.808594, 864.481628, 968.195312, 1075.73999, 1170.61475, 1265.15259, 1371.21558, 1468.55005, 1571.05615, 1672.38135, 1778.00134, 1872.56262, 1966.58154, 2072.36841, 2179.71704, 2283.76514, 2384.46411, 2490.13257, 2599.53394, 2707.01978, 2810.60596, 2911.93286, 3020.66675, 3127.27441, 3221.49609, 3314.30688, 3416.52588, 3512.29761, 3616.40552, 3718.34863, 3818.66333, 3914.07153, 4013.93652, 4114.04248, 4208.70605, 4304.10547, 4395.88867, 4498.96729, 4593.25537, 4697.24609, 4795.06445, 4897.97217, 5001.44189, 5101.17578, 5205.04199, 5309.52881, 5417.85449, 5519.53027, 5626.40137, 5724.1001, 5822.06738, 5923.00635, 6028.41211, 6133.78369, 6235.35254, 6332.54395, 6435.10986, 6533.79004, 6631.45801, 6734.65723, 6834.57227, 6936.88232, 7038.99658, 7136.78467, 7231.29199, 7324.50879, 7426.74561, 7529.5918, 7629.18213, 7733.75879, 7841.38721, 7950.32227, 8055.29395, 8149.69727, 8256.43848, 8361.33984, 8465.97168, 8571.44727, 8675.98926, 8784.28125, 8883.99316, 8984.80859, 9076.59766, 9171.31055, 9268.41699, 9363.35742, 9456.16309, 9552.01367, 9658.81934, 9763.65625, 9861.06543, 9955.83887, 10063.1582, 10156.9092, 10065.167, 9967.58008, 9877.95117, 9786.16504, 9679.5166, 9577.64355, 9484.41309, 9389.54199, 9292.69824, 9188.18652, 9080.7373, 8987.41992, 8890.9668, 8784.26758, 8687.62402, 8584.01562, 8481.8584, 8376.33984, 8282.48242, 8189.19141, 8083.7627, 7975.4668, 7871.04004, 7770.00732, 7665.06885, 7555.86816, 7448.11133, 7344.33447, 7243.09229, 7135.0415, 7029.93799, 6936.26123, 6841.30908, 6740.58594, 6645.73584, 6540.71826, 6438.14209, 6337.89941, 6242.90039, 6142.30811, 6042.51318, 5947.22705, 5852.85693, 5760.86035, 5657.41748, 5563.83398, 5459.50098, 5360.4043, 5257.30371, 5155.09961, 5056.37012, 4951.11865, 4845.60791, 4738.35059, 4636.5415, 4528.67529, 4431.90234, 4333.99365, 4232.41602, 4128.18994, 4021.87793, 3921.60034, 3823.93848, 3721.07544, 3623.16846, 3523.88208, 3419.4519, 3319.68115, 3219.28833, 3116.26855, 3018.83423, 2924.45386, 2831.88647, 2729.68823, 2625.95947, 2527.30566, 2422.73486, 2314.74268, 2205.20996, 2101.85889, 2007.11694, 1900.13562, 1795.38318, 1690.30554, 1583.79626, 1478.6687, 1371.77905, 1271.42761, 1171.54749, 1079.6416, 985.782104, 887.747192, 792.291931, 699.889221, 604.313354, 497.099091, 391.736786, 294.407593, 200.101624, 93.992569

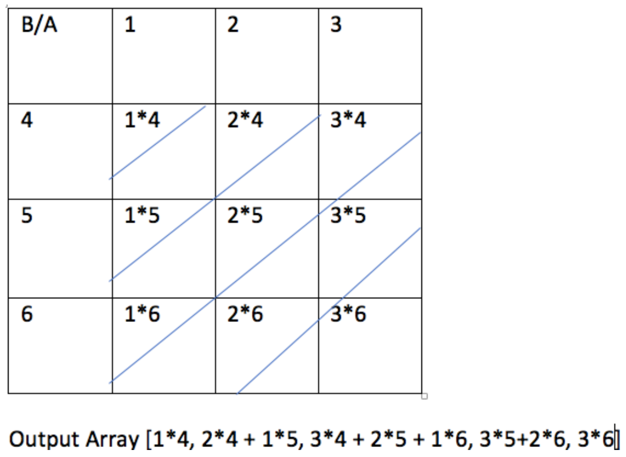
APPENDIX D
GRAPHS



B/A	1	2	3
4	1*4	2*4	3*4
5	1*5	2*5	3*5
6	1*6	2*6	3*6

Output Array [3*4, 2*4+3*5, 1*4+2*5+3*6,1*5+2*6,1*6]

Fig. 1. How Correlation is calculated.



B/A	1	2	3
4	1*4	2*4	3*4
5	1*5	2*5	3*5
6	1*6	2*6	3*6

Output Array [1*4, 2*4 + 1*5, 3*4 + 2*5 + 1*6, 3*5+2*6, 3*6]

Fig. 2. How Convolution is calculated.

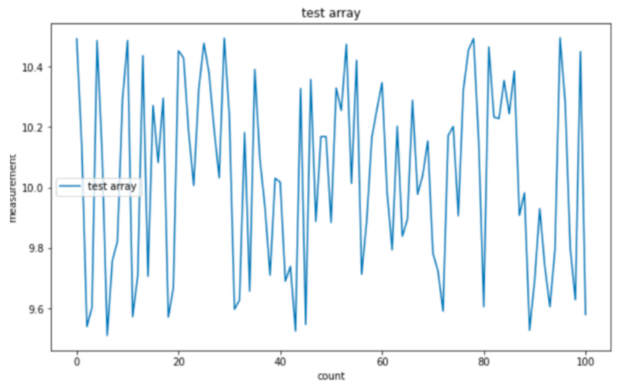


Fig. 3. Test Array

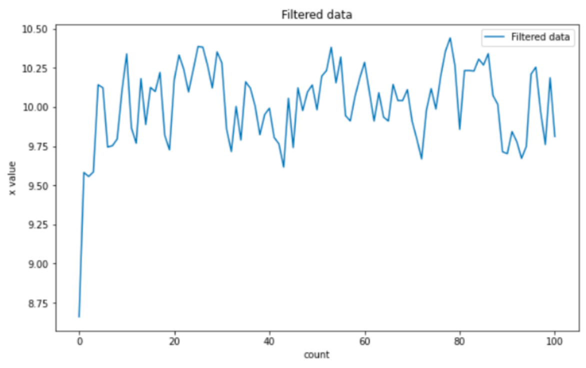


Fig. 4. Filtered Data

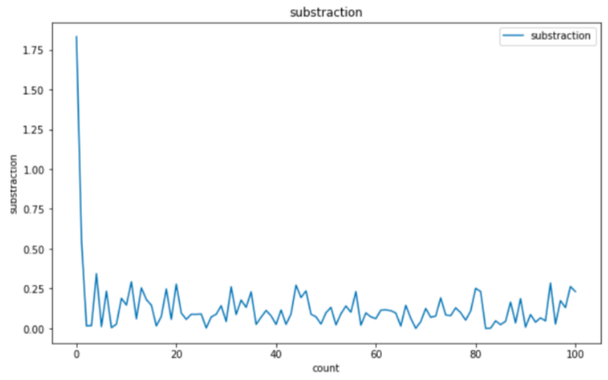


Fig. 5. Results from Subtraction

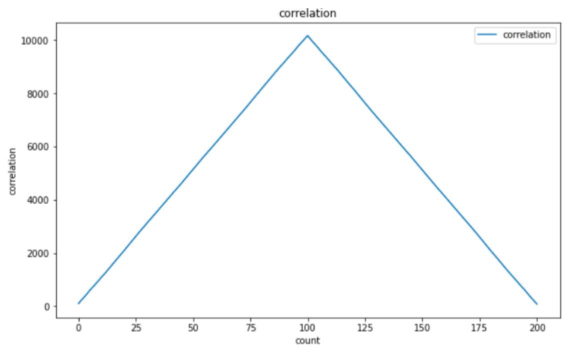


Fig. 6. Results from Correlation

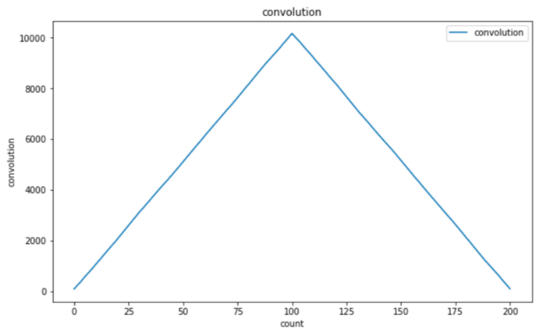


Fig. 7. Results from Convolution

References

[1] ECSE 444 Lab 1 Handout.

[2] arm_math.h File Reference, CMSIS DSP Software Library. ARM Ltd. web site: http://www.disca.upv.es/aperles/arm_cortex_m3/curset/CMSIS/Documentation/DSP/html/arm__math_8h.html, Mar 28, 2012.

[3] Convolution and Correlation. Tutorialspoint, Web site: https://www.tutorialspoint.com/signals_and_systems /convolution_and_correlation.htm