Kalman Filters

Group 12 – ECSE 444 Final Report

Mahad Khan
Department of Electrical and Computer Engineering
McGill University
Montreal, Canada
mahad.khan@mail.mcgill.ca

***Abstract*—In this project, we will be implementing a data-processing algorithm, namely, Kalman filter using the C programming language. We will then optimize the algorithm and perform quantitative analysis and measurements of performance.**

Keywords—Kalman Filter

I. Introduction

In statistics and control theory, Kalman filtering is an algorithm that uses a series of measurements observed over time, containing statistical noise and other inaccuracies, and produces estimates of unknown variables that tend to be more accurate than those based on a single measurement alone, by estimating a joint probability distribution over the variables for each timeframe [1]. Kalman filters have numerous applications in technology. A common application is for guidance, navigation, and control of vehicles, particularly aircraft, spacecraft and dynamically positions ships. For the purpose of this project, we will be implementing a baseline discrete Kalman Filter in C. We will then optimize our algorithm and perform quantitative analysis to compare the two approaches.

II. The Discrete Kalman Filter

The Kalman filter estimates a process by using a form of feedback control: the filter estimates the process state at some time and then obtains feedback in the form of (noisy) measurements. As such, the equations of the Kalman filter fall into two groups: time update equations and measurement update equations [2]. The time update equations are responsible for projecting forward (in time) the current state and error covariance estimates to obtain the priori estimates for the next time step. The measurement update equations are responsible for the feedback i.e. for incorporating a new measurement into the priori estimate to obtain an improved posteriori estimate [2]. The discrete Kalman filter algorithm can run in real time, using only the present input measurements and the previously calculated state and its uncertainty matrix; no additional past information is required [1].

In the time update equations (1) and (2), (1) projects the state ahead, and (2) projects the error covariance ahead.

$$\hat{x}_k^- = A\hat{x}_{k-1} + B\mu_{k-1} \quad (1)$$
$$P_k^- = AP_{k-1}A^T + Q \quad (2)$$

Equations (3), (4) and (5) represent the measurement update equations the filter implements. Equation (3) computes the Kalman gain, K_k, (4) updates the estimate with measurement z_k and (5) updates the error covariance, P_k.

$$K_k = P_k^- H^T (HP_k^- H^T + R)^{-1} \quad (3)$$
$$\hat{x}_k = \hat{x}_k^- + K_k(z_k - H\hat{x}_k^-) \quad (4)$$
$$P_k = (I - K_k H)P_k^- \quad (5)$$

In the above equations, A is an $n \times n$ matrix that relates the state at the previous time step $k-1$ to the state at the current step k, in the absence of either a driving function or process noise. The $n \times l$ matrix B relates the optional control input μ to the state x. The $m \times n$ matrix H in the measurement equation relates the state to the measurement z_k. Parameters R and Q are the process noise covariance and the measurement noise covariance respectively.

III. Proposed Solution

In this project, we will be implementing a linear Kalman filter that processes one-dimensional data. We attempt to estimate a scalar random constant, let it be voltage for the purpose of our experiment. We assume that we have to ability to take measurements of the constant, but that the measurements are corrupted by some noise. In our system, the state does not change from step to step so $A = 1$. There is no control input so $\mu = 0$. Our noisy measurement is of the state directly so $H = 1$. Presuming a very small process variance, we let $Q = 1e-5$. We assume that we know that the true value of the random constant has a standard normal probability distribution, so we will "seed" our filter with the guess that the constant is 0. This implies that before starting we let $\hat{x}_{k-1} = 0$. Similarly, we need to choose an initial value for P_{k-1}, call it P_0. Because we do not know that our initial state estimate $\hat{x}_0 = 0$ was correct, choosing almost any $P_0 \neq 0$ would allow the filter to eventually converge. In our case, we will start our filter with $P_0 = 1$ [2].

To simulate the Kalman filter, we randomly chose a scalar constant $x = -0.37727$. This represents the truth value. We then simulate the system by randomly generating 50 distinct measurements z_k that had error normally distributed around x with a standard deviation of 0.1. In our simulation, we fixed the measurement variance at $R = (0.1)^2 = 0.01$. Figure 1 depicts the result of our simulation.

Experimentation showed that when the filter was told that measurement variance was 100 times greater (i.e. $R = 1$), it was "slower" to believe the measurements. When it was told that the measurement variance was 100 times

Proceedings for McGill University's ECSE 444 Final Project.

smaller (i.e. $R = 0.0001$), it was very "quick" to believe the noisy measurements [2].

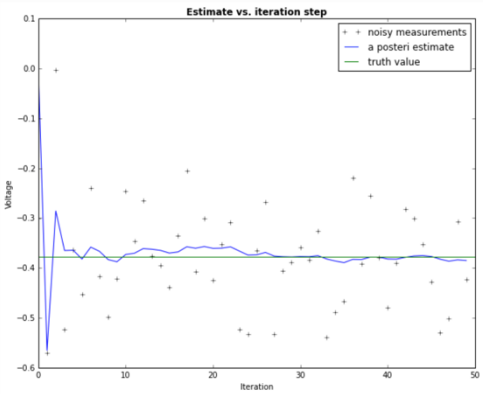

Figure 1. Result of simulation with R = 0.01

IV. Software Components of our System

While implementing the baseline Kalman filter algorithm, we provided the filter with data of type **long double**. This data type was chosen because real world systems require more precision of data. The update equations were run in a **for loop**. For our simulations, the loop was iterated 50 times. We randomly generated 50 different measurement z_k using C's **stdlib.h** function **rand()**. These measurements were fed to the filter in real-time. To measure the execution time of our performance critical code, we used the **time.h** standard C library. Furthermore, to display the results of our simulation, we used the **printf()** function. The baseline Kalman filter for simulation can be found in the file **kalman_filter.c** in our project directory. For the purpose of algorithm evaluation, we performed some addition arithmetic operations to find the reduction in the error of the system due to the Kalman filter. The optimized simulation program is stored in **kalman_filter_optimized.c**. Test data for performance analysis was produced by a Python script that generated 1000 normally distributed measurements around x with a standard deviation of 0.1. The script stores these values in **values.txt**. These values are accessed via a modified Array data structure by our performance analysis programs: **kalman_filter_test.c**, **kalman_filter_optimized_test.c**

V. Optimizations

In order to optimize our algorithm and reduce its execution time, we attempted at applying various optimization strategies at the assembly level.

I. ***Re-Ordering Instructions:*** Re-ordering instructions allow us to improve execution latency by eliminating memory accesses or delays due to data dependencies. In C, we represented (1) as *xhatminus = xhat_last*, (2) as *Pminus = P_last + Q* and (3) as *K=Pminus * (1.0/(Pminus + R))*. We noted that if we perform the sum $P + Q$ outside the loop, we can update (3) to *K = Pminus * (1.0/(P_last + sum_R_Q))*. This reduces one arithmetic operation within the performance critical section of our system. We then moved (2) before (1) to reduce delays due to data dependencies.

II. ***Data Type: long double vs float:*** We then changed the data type of our variable to **float**. A **long double** data type requires more storage than a **float**, therefore it takes longer for the processor to read the data. This optimization comes at the cost of the precision of the system.

III. ***Memory Allocation: Stack vs Heap****:* Although our baseline filter that we use for the simulation of the filter receives real-time randomly generated data, the baseline filter implementation for performance analysis stores variables and input data in heap memory which comes from the **malloc** operation. In order to speed-up our system, we then stored the data in a stack. The stack is faster because the access pattern makes it trivial to allocate and deallocate memory from it, while the heap has much more complex bookkeeping involved in allocation/deallocation. Furthermore, heap allocation typically has to be multi-threading safe and this causes synchronization overhead.

IV. ***Loop Unrolling:*** Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. It allows us to remove or reduce iterations. The fewer branch jumps at assembly level increases the program's speed by eliminating loop control and test instructions. In order to further optimize our system, we unrolled the loop 5 times for our simulation program (50 iterations in total) and 10 times for our performance analysis program (1000 iterations in total). We attempted at further unrolling the loop but realized that the program binary size increases which reduces performance.

V. ***In-Lining Functions****:* Inline functions may improve performance as data does not need to be pushed and popped on/off the stack. However, we observed that they did not optimize our test program, and hence they were excluded from the code and results. They only negligibly optimized our simulation program.

VI. Results

With our test data, the total error with noisy measurements was **81.74** and with Kalman filtering was **8.51**. This results in a **reduction in error by 90%.**

We were able to optimize the performance of our system by **73.02%**. Table 1 below summarizes the performance analysis results. Note: gcc compile flag was set to -O0.

Optimization	Exec. Time (s)	Speed-up (s)
None (baseline)	6.3e-5	-
I	4.0e-5	2.3e-5
II	3.0e-5	1.0e-5
III	1.9e-5	1.1e-5
IV	1.7e-5	0.2e-5

Table 1. Performance optimization results

References

[1] "Kalman filter," *Wikipedia*, 21-Mar-2020. [Online]. Available: https://en.wikipedia.org/wiki/Kalman_filter. [Accessed: 11-Apr-2020].

[2] Greg Welch and Bishop Gary, *An Introduction to the Kalman Filter.* USA: University of North Carolina at Chapel Hill, 1995.