ECE 420 Final Project Report
Real-Time Kalman Filtering for Human State-Space Estimation

Steven Dimov, Rohan Gulur
{sdimov2, rgulur2} @ illinois.edu

I. Introduction

State Space Estimation is a crucial aspect of controls and signal processing engineering. State Space Estimation is essential in the real world because it allows us to know the position, and/or "state" of things. State space estimation is utilized for cars, trains, airplanes - but it doesn't end here. State spaces don't only reference objects in motion, but really refer to anything with a "state", be it changing or unchanging over time. This state could be referring to Energy consumption and capacity utilization in the power grid - biotransducer values in biosensors, and many other things. In a perfect world, these sensors give us unscathed, accurate information of values we are attempting to track. However, noise is ever so present in every sort of system of the modern world, and stem from a multitude of sources. To hamper the effects of noise, different sorts of algorithms/methods are used in practice.

For the problem that we aim to solve, we are attempting to track the estimated state of a runner. These states for the runner include position, velocity, and acceleration. We know that these states accumulate noise, as measurements are inherently noisy. We propose to utilize a Kalman Filter to estimate the state space of a runner more accurately. A Kalman Filter is a causal, recursive, algorithm that works on two main data points - a measurement and a prediction, recognizes the inherent noise in both of these data points, and uses statistics to improve the predicted estimate at each next timestep.

For our problem specifically, our prediction is kinematically based, simply off of motion equations - while our measurement is recorded from IMU acceleration data. Our algorithm

will also update the stochastic elements (prediction/measurement covariance matrices) of our system as timesteps progress, reinforcing and enhancing our estimate as both the timesteps and measurements sequentially increase.

II. Literature/Research Review

Inspiration for my use of the Kalman Filtering algorithm was inspired by a few papers, specifically those regarding large-scale state estimation of objects in motion. The first paper I referenced was published by two aeronautical engineers. A link to the paper [1], is found below, but I will give a brief explanation of what the document highlights. The paper begins mentioning the historical usage of the Kalman Filter, where its first uses were found in the aerospace industry. It mentions that the need for more advanced navigation systems on the Apollo and Lockheed C-5A airplanes began the development of the first real-time Kalman Filtering algorithms. The paper then goes on to highlight its usage in NASA's Apollo mission in depth, and highlights certain inconsistencies with the development of the filter at the time, including round-off issues as well as "small" covariance matrices that were ignored (in a sense) by the Kalman Filter.

Another paper which I found interesting was a paper published regarding the Kalman Filter's overlap/intersection with Machine Learning architecture. The paper [2], KalmanNet highlights the individual limitations of Kalman Filters and Deep Neural Networks respectively, and aims to merge these limitations into a modified joint algorithm that aims to fuse both. Oftentimes, creation of the Q and R matrices in Kalman Filters are difficult, yet this paper highlights a Deep Learning architecture that trains a loss function based on given parameters to create the optimal covariance matrices Q and R at each timestep.

These two papers served as the inspiration and background for our report because each paper highlights its ambitious goals/motivations ***as well*** as its respective challenges. For our project, we estimated the state of an object (a person :)) in motion, just as the NASA engineers estimated but for aeronautical vehicles. Since we are working with an embedded programming interface, we also had to face similar challenges with floating point / round off issues as well as potential erroneous matrices that could lead to gain coefficients of zero and "ignored" filter elements.

The need for accurate Q and R matrices is also important, and the second paper highlights the need to accurately design these matrices as the timesteps of operation progress. Since the tie-in between DSP and Deep Learning is ever so present, the tie-in with Machine Learning makes optimization of control algorithms such as the Kalman Filter interesting to pursue.

III. Technical Description

Our final project's architecture uses the accelerometer from an NVIDIA Shield to receive measurements and uses a kinematic model to determine the predicted state of the Kalman Filter.

Below we will discuss the relevant mathematical steps that a Kalman Filter has that allow it to make its estimates and relate it to our problem statement of what we are trying to accomplish. We will discuss each (main) equation and clearly specify the two main stages of the Kalman Filter - the prediction and the update stage, and then compare it to what we expect (our ground truth)

State Prediction:

$$\hat{x}_{k|k-1} = F_k \hat{x}_{k-1|k-1} + B_k u_k \tag{1}$$

Error Covariance Prediction:

$$P_{k|k-1} = F_k P_{k-1|k-1} F_k^T + Q_k \tag{2}$$

Measurement Residual:

$$y_k = z_k - H_k \hat{x}_{k|k-1} \tag{3}$$

Residual Covariance:

$$S_k = H_k P_{k|k-1} H_k^T + R_k \tag{4}$$

Kalman Gain:

$$K_k = P_{k|k-1} H_k^T S_k^{-1} \tag{5}$$

State Update:

$$\hat{x}_{k|k} = \hat{x}_{k|k-1} + K_k y_k \tag{6}$$

Error Covariance Update:

$$P_{k|k} = (I - K_k H_k) P_{k|k-1} \tag{7}$$

Above: List of main equations / mathematical constructs of Kalman Filter – each to be explained in relation to our desired outcome below:

The first equation, *(1) - State Prediction*, of the Kalman Filter aims to predict an estimate of the next state of the system based on the current state of the system. There is a $\mathbf{B}(k)$ matrix multiplied by a $\mathbf{u}(k)$ matrix, but in our current system, we are not using it as it symbolizes a

control input. Our system dynamics assume a constant acceleration and a kinematically derived next state as shown below.

$$\mathbf{x} = \begin{bmatrix} d_x \\ v_x \\ a_x \\ d_y \\ v_y \\ a_y \end{bmatrix}$$

(our state space we are modeling)

$$\mathbf{F} = \begin{bmatrix} 1 & \Delta t & \frac{1}{2}\Delta t^2 & 0 & 0 & 0 \\ 0 & 1 & \Delta t & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & \Delta t & \frac{1}{2}\Delta t^2 \\ 0 & 0 & 0 & 0 & 1 & \Delta t \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

(the state transition matrix)

When multiplying the $\mathbf{F}$ matrix by the current state space, we predict the next state's distance, velocity, and acceleration. In our case, this kinematic model will be for a walker or runner.

The second equation, *(2) - Error Covariance Prediction*, represents our prediction for the covariance of the states of the system. Its update is determined by how we expect our states to transition, as well as the $\mathbf{Q}$ matrix—a linearly additive matrix that represents the "process" noise at each timestep. When dealing with more than one variable, multiple samples over time can provide a statistical model of each variable as a normal distribution probability density function. Naturally, there is some difference in the mean and variance between the two variables, but this is what the Kalman filter takes advantage of to make a greater estimate. In our system, this represents the covariance of the state variables (position, velocity, acceleration) that we are tracking.

The third equation, *(3) - Measurement Residual*, represents the actual error between our measurements and our prediction. As mentioned, our measurement is our IMU's real-time accelerometer output.

It is important to notice, however, that the size of our state space and our measurement are not equal in our filter. We are estimating 6 variables, but we only measure 2 (acceleration in the x direction and acceleration in the y direction). To ensure we can take the difference between these two variables, we create an $\mathbf{H}$ matrix that is multiplied with our prediction from *(1)* so that we can directly subtract two matrices of equal size. This allows us to consider the difference between the measurement we receive (two-dimensional acceleration) and two of the six state variables we are tracking (acceleration in x and y directions).

$$\mathbf{z} = \begin{bmatrix} a_x^{\text{measured}} \\ a_y^{\text{measured}} \end{bmatrix}$$

Above: *(our actual measurement column vector of length 2)*

$$\mathbf{H} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Above: (our measurement matrix to format residual)

The fourth equation, *(4)* - *Residual Covariance*, represents the covariance of the residual at the current timestep. It has a linearly added $\mathbf{R}$ matrix, which represents the additive noise (another covariance matrix) of the measurement itself at the said timestep. Since our measurement comes from the accelerometer, this noise corresponds to the accelerometer data in the x and y directions. Hence, our $\mathbf{R}$ matrix is a 2×2 matrix in this case.

The fifth equation, *(5)* - *Kalman Gain*, is arguably the most important step of the Kalman Filtering algorithm. It determines the filter coefficient, $\mathbf{K}$, at each timestep. This is calculated as a function of previously determined matrices and values during the prediction and update stages.

What follows this stage is *(6)* - *State Update*. This state update produces a more accurate estimate for the state given the computed Kalman Gain and the Measurement Residual. For our system, this represents the final updated estimate of the state at the given timestep, the result of the algorithm fusing the noisy estimates of the prediction and measurement "together".

Last but not least is *(7)* - *Error Covariance Update*. This step provides an updated estimate of the state covariance for all six states we track (position in x and y, velocity in x and y, and acceleration in x and y). This calculation ***also*** incorporates the updated Kalman Gain.

These are the steps the Kalman Filter follows to estimate the state and state covariance values at all timesteps greater than the zeroth timestep.

The delta in time between sample is not exactly perfectly constant from our tablet's accelerometer - to compensate, we adaptively update this time difference comparing the delta in time from the previous sample to the current sample and synchronously update our state transition (F) matrix's delta time parameter.

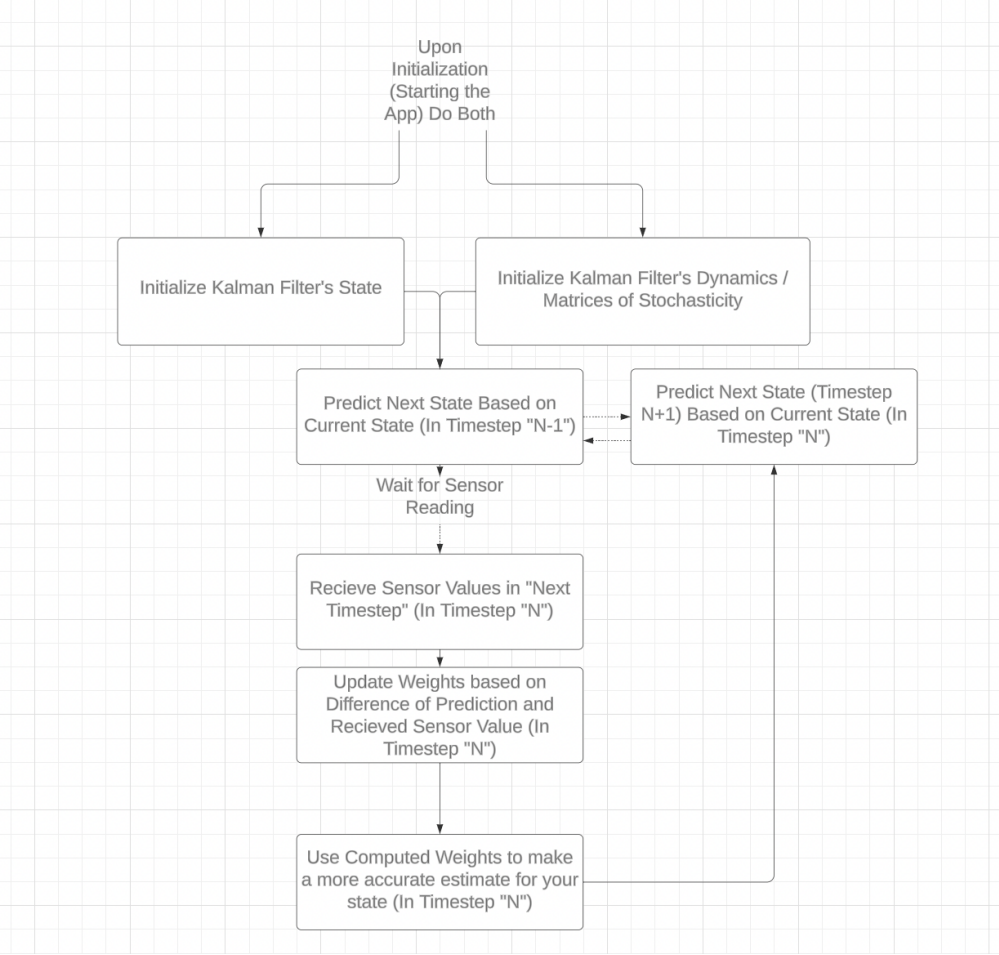

Fig. 1. Basic High-Level Block Diagram of the workings of a Kalman Filter. Note that my notion for predicting the state at timestep "N" from "N-1" and predicting the state at timestep "N+1" from "N" is interchangeable (hence the arrows pointing at eachother). I simply included the latter for the sake of more clearly explaining a full cycle from one nonzero timestep to the next.

There are multiple different "ground truths" that we used to test our estimations against - one being a separate device's acceleration, another being a NVIDIA Shield GPS' calculated acceleration, and the last being the MSE comparison against a ground truth of "zero" acceleration. These methodologies are further discussed in the Testing section.

IV. End Result and Screenshots of Application

We accomplished our Minimum Viable Product (MVP) for this project by creating a project capable of plotting any of the three states we aimed to track (acceleration, velocity, and position) in real time.

The user interface of our device is rather simple - since our device's goal is to improve state estimation, our user interface consists of a graph that plots the states we choose to track (acceleration in the X and Y direction, in our case) in real time.

For the sake of demo and for the ease of viewing (rather than plotting all of the different variables we are tracking at once, we simply plotted two variables as shown on the below graph.

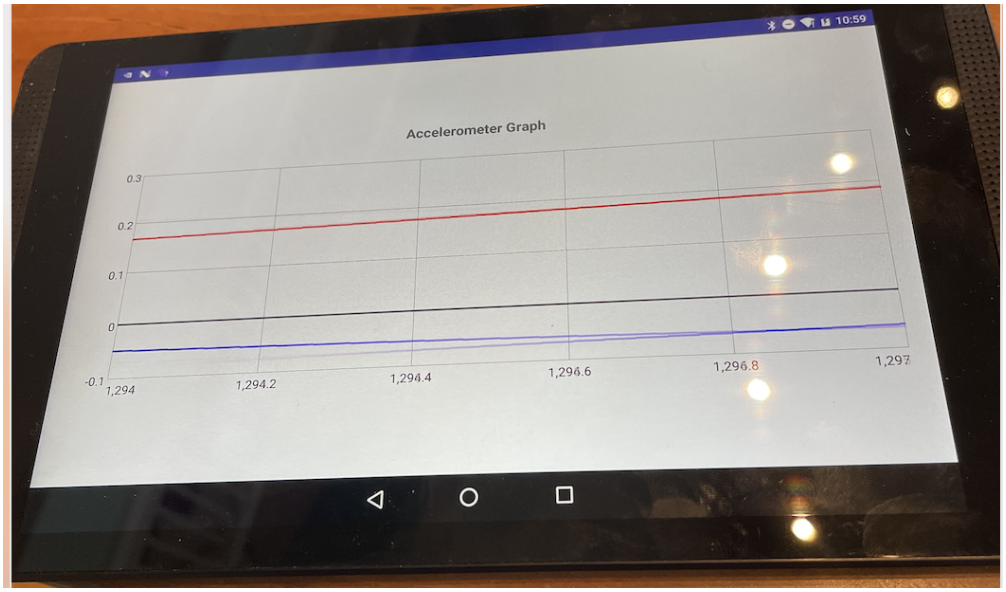

Fig. 2. Real Time Plotting of Acceleration in the X (Red Line) and Y (Blue Line) direction

Further demonstration is available in the video links attached to this project. In the videos, the red and blue graphs correspond to the filtered acceleration values, in the x and y directions respectively. The green and yellow graphs correspond to the raw acceleration in the x and y directions respectively. As visible, the filtered data appears less noisy. The noise level can be further verified further with the MSE plotting.

If other variables, such as the velocity or position, needed to be plotted, the code can simply be modified to plot these other states.

```
// Access the 5th component (index 4) and 6th component (index 5)
double fifthComponent = getState.get( row: 4,  col: 0);  // 5th component of the state vector
double sixthComponent = getState.get( row: 5,  col: 0);  // 6th component of the state vector

//get velocity components of the kalman filter
float thirdcomponent = (float) getState.get( row: 2,  col: 0);
float fourthcomponent = (float) getState.get( row: 3,  col: 0);
```

Fig. 3. Accessing the "states" of the Kalman Filter, with components 3, 4, 5, and 6 corresponding to the velocity in the X and Y direction and acceleration in the X and Y direction (respectively)

```
graphManager.updateGraph((float) fifthComponent, (float) sixthComponent);
```

Fig. 4. Plotting some of the above accessed "states"

As seen in the above screenshots, it is easy to access the states of the Kalman Filter and plot these states with rapid real-time updates using the Graph Manager. To plot velocity in X and Y, for example, i would simply have to replace "fifthComponent" and "sixthComponent" with "thirdComponent" and "fourthcomponent".

V. Final Results and Testing

Testing the accuracy of the Kalman Filter posed numerous challenges but we schemed multiple methods to test the filter, with each method having its own pros and cons.

Our very first method involved using an external device to serve as a "ground truth". This external device, for example, could be a Smartwatch/Fitness watch etc; but we used a Garmin Running Watch. The methodology behind this testing involved wearing the watch on one's wrist while also holding the tablet in such a manner where the coordinate basis for both is the same.

Ensuring that the X, Y, and Z directions of both the watch and the tablet were consistent allowed us to have confidence that the X and Y accelerations in both the NVIDIA Shield as well as the Garmin Watch relative to each other more or less shared the same coordinate references.

Ensuring the sample rate being exactly equal for both is less important due to our knowledge that *most* human acceleration falls within the frequency band of 0-10Hz. Both devices sample at 25Hz or higher, preventing the possibility of an aliased signal.

With the user both holding the tablet and wearing the watch with the coordinate bases more or less aligned, we viewed that the X and Y accelerations more or less matched up at the same

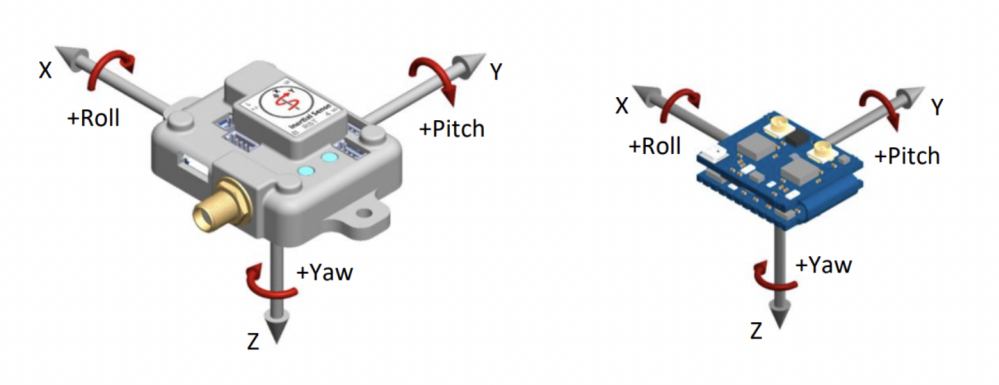

Fig. 5. Example Frame of Reference of IMU device

time on both the tablet and the watch with small error (qualitatively observed).

This was a good way to test, but the constraint was that comparison was all done qualitatively. If comparison was to be done quantitatively, we would have to do all of the processing offline and compare a metric such as mean-square error.

The second methodology we used to test was through using the NVIDIA shield's built in GPS to serve as the ground truth. We took in GPS coordinates (provided to us as latitude and longitude points) and converted these points to accelerometer data through the haversine formula, as described below:

The haversine formula calculates the great-circle distance between two points on a sphere:

$$d = 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \tag{8}$$

Where:

- d is the great-circle distance between the two points
- r is the radius of the sphere (typically the Earth's radius)
- ϕ_1, ϕ_2 are the latitudes of point 1 and point 2 (in radians)
- λ_1, λ_2 are the longitudes of point 1 and point 2 (in radians)
- $\arcsin$ is the inverse sine function

An alternative, equivalent form of the formula can be written as:

$$d = 2r \arctan 2\left(\sqrt{a}, \sqrt{1-a}\right) \tag{9}$$

Where:

$$a = \sin^2\left(\frac{\phi_2 - \phi_1}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right) \tag{10}$$

As the distance is calculated through this formula, velocity can be calculated from the change of distance, and acceleration can be calculated from the change of velocity.

Instead of having two devices, we only needed one device, and on the tablet we were able to plot the real-time values for the acceleration provided through the Kalman filter on one graph, as well as the real-time values for the acceleration provided through the double derivative of distance calculated using the haversine formula on another graph.

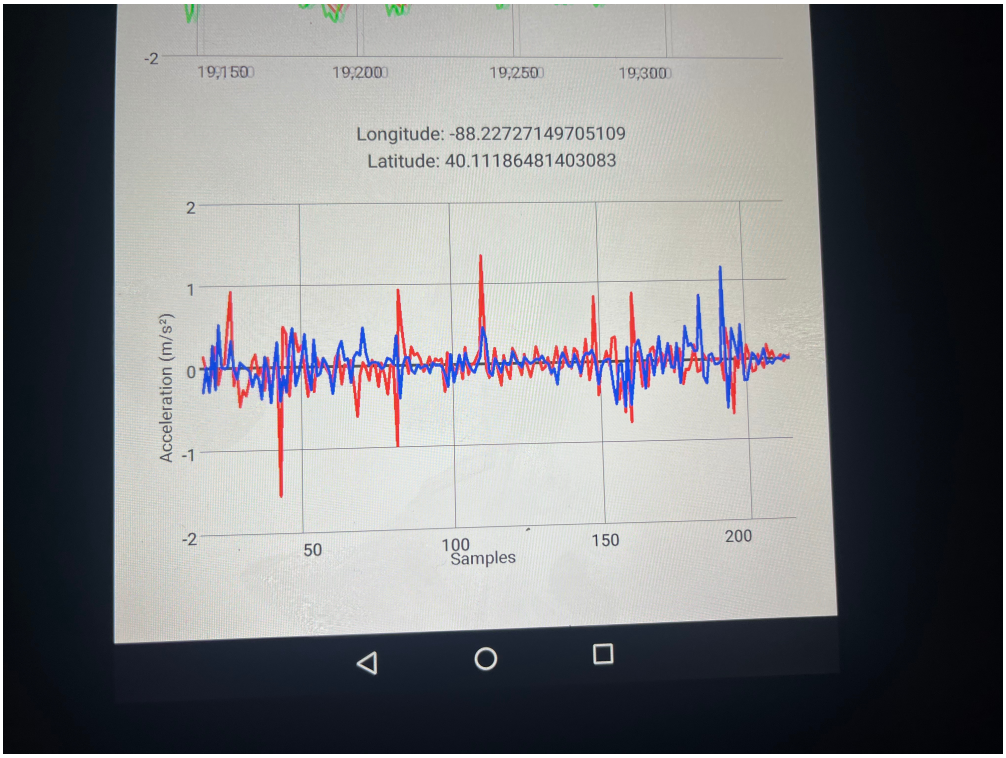

Fig. 6. acceleration from GPS source plot

Using the GPS (from android tablet) to serve as the ground truth also presented unique issues. For the GPS, the sample rate was extremely low and varied based on the connection stability, struggling to receive reception indoors. Upon testing the GPS outdoors, we noticed that the GPS coordinates would still only update every second or two if not even slower. This rate was ***much*** less than the rate at which we would receive samples from the Shield's accelerometer, so

inherently, our accelerometer values would be inherently inaccurate as the sample rates would be mismatched.

Could we have interpolated the accelerometer values calculated from the NVIDIA shield? The answer is yes. However, doing so, especially to match the GPS' low sampling rate to the much higher accelerometer sample rate is also inaccurate, as acceleration is a highly fluctuating variable with the possibility to drastically change within milliseconds. Trying to solve this problem using this method could have potentially introduced even more errors and dug a deeper hole in the process of comparing different interpolation methods (Linear, Nearest-Neighbor, Sinc, etc;).

The third method which we used to verify the integrity and efficacy of our Kalman Filter was through using a constant and unchanging, "known" ground truth.

I would place the tablet on an unmoving surface (a still table, for example).

This verification process was arguably the easiest to implement but the most accurate, statically speaking, to prove the filter's accuracy. At each timestep, I computed the Mean-Square Error (MSE) between the estimated Kalman Filter acceleration value in the X direction, and zero (a still device should have zero acceleration, in theory).

I repeated the same MSE calculation again, but this time with the ***measured acceleration value*** in the X direction at each timestep, and zero (this whole procedure could have been repeated in a similar manner using the Y direction, as well).

By performing these calculations, I was able to visually plot both MSEs on a graph. The MSE of the Kalman Filtered acceleration was clearly much lower as timesteps progressed in comparison with the MSE of the raw accelerometer data, and the Kalman Filter MSE even descended to values where it maintained ***local*** stability noticeably faster than the raw acceleration.

Is this methodology of testing 100 percent perfect? No. Of course there is always some negligible acceleration present acting on objects that we believe to be completely at rest. The air blowing from an AC vent and a tap on the table acts as some miniscule force on our NVIDIA Shield. Perhaps even the internal components of the tablet may affect its own measurements to a negligible degree.

However, for the sake of comparing against a known ground truth, this is our best known bet, because despite all these microscopic elements of noise, our acceleration, ***basically is*** - Zero!

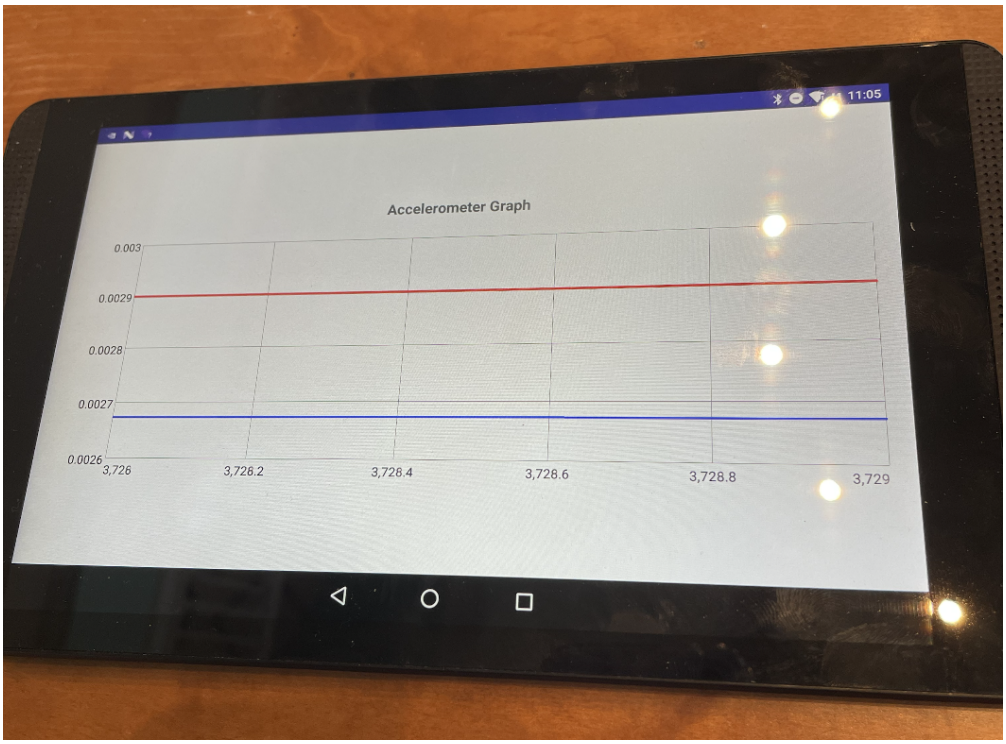

Fig. 7. Testing of the MSE of the unfiltered acceleration (red) against the filtered acceleration (blue) - comparing X direction for both.

Extensions and Modification

The following enhancements would extend the scope of the project beyond its current functionality (the MVP):

Position Estimation

Incorporate additional sensor data, such as gyroscope or compass readings, to determine the orientation of the device and estimate position more accurately as a total distance traveled.

Control Input Integration

Adapt the Kalman filter to use acceleration as a control variable rather than assuming constant acceleration. This approach would introduce dynamic control to the filter, improving flexibility and accuracy.

Multi-Measurement Variable Kalman Filter

Leverage the Kalman filter's versatility to integrate data from multiple sources, such as IMU, gyroscope, compass, and GPS, for a more accurate state estimation. Implement sensor fusion to corroborate predictions and further reduce uncertainty.

SOFTWARE/HARDWARE DOCUMENTATION

The class `SensorReader.java` handles the main computation logic. It is responsible for the initialization of matrices pertaining to state and covariance upon the startup of the application.

At each sample of the accelerometer sensor, this class handles the process of calculating an updated estimate of state at each timestep as these accelerometer measurements are received.

The class `KalmanFilter.java` is a class that represents the Kalman Filter object that is instantiated in the SensorReader. The Kalman Filter class contains functions for the predict and update stage, as well as a function to retrieve the state. It also features a function to construct the object upon specifying initial state dynamics and initial covariances.

The class `GraphManager.java` handles plotting the data in the user interface. It has a function that updates the graphs allowing for new points to be plotted - called in SensorReader as new values are recieved.

The class `GPSManager.java` handles retrieval and access of the GPS data from the android tablet after permissions are accounted for succesfully. This file handles the updates of the GPS coordinates when detecting a change and has a function to calculate the distance, velocity, and acceleration from the change in GPS coordinates using the haversine formula to convert it from longitude and latitude.

To handle matrix calculations, we utilized the Efficient Java Matrix Library (EJML)'s DMatrixRMaj class to handle matrix creation and operations.

The tablet used for testing was an NVIDIA Shield. On this hardware, the computation speed was sufficient for the signal processing that was done by our filter. The sensors of this device used in the MVP were the accelerometer and GPS. While the accelerometer sensor was pretty fast and sufficient for the purposes of this project, the GPS hardware did not hold up to ideal standard for our usage, as its sample rate was slower than the accelerometer's and varied depending on stable connection in the environment.

REFERENCES

[1] L. McGee *et al.*, "Discovery of the kalman filter as a practical tool for aerospace and industry," *N/A*, 1985, available at: https://ntrs.nasa.gov/api/citations/19860003843/downloads/19860003843.pdf (Accessed: 20 Nov 2024).

[2] G. Revach *et al.*, "Neural network aided kalman filtering for partially known dynamics," *N/A*, 2022, available at: https://ieeexplore.ieee.org/document/9733186 (Accessed: 20 Nov 2024).