Dakota Barnes
Nik Belle

ECE 253 Final Report: Mountain Bike Trail Grade Monitor

Introduction

For our ECE 253 Final Project we set out to develop a mountain bike trail grade monitor prototype. Both of us grew up mountain biking and jumped on an opportunity to build a product that we would love to use ourselves. Given our timeline and the focus of this course, we placed our efforts into creating electronic functionality with an intuitive and responsive user interface. In the following report, we aim to provide a thorough overview of the product starting from the user experience, followed by the engineering that enables tracking a rider's bike incline angle.

User Journey

Our trail grade monitor product is a small, lightweight device that attaches to a mountain bike's handlebars to enable a user to track their incline throughout their ride. When turning on the device, the user sees their current incline displayed as a number, above a simple display of the slope. The user has access to a button that allows them to start recording their ride. Pressing this button again pauses the ride, while a second button lets them reset the ride. They may view their current ride and its metrics by pressing the third button. On entering the ride display, a user can see their maximum incline, minimum incline, average incline, total incline reads, and a diagram showing the stream of incline changes over the course of their ride. During a bike ride, a user can toggle between viewing their current incline and their full ride metrics by pressing this third button. Finally, a user can turn the encoder to increase and decrease the sensitivity of the device based on their planned ride. If they are expecting significant bumps, setting the sensitivity to 1 or 2 will reduce noise. For a smooth trail, they can increase the sensitivity to 5 or 6 for more responsive incline reads.

Hardware Overview

The hardware setup for our project was centered around interfacing an MPU 6050 inertial measurement unit with our NEXYS A7 FPGA through I2C. Additionally, we included the SPI based LCD Display, Rotary Encoder, and onboard buttons for a comprehensive user interface. While our NEXYS A7 contains an ADXL accelerometer, which can be used to calculate incline through linear acceleration data, relying on that data alone would fail to meet the expectations for an effective mountain bike trail grade sensor. Riding over rough terrain leads to vibrations and sudden impulses that are all picked up by an accelerometer as noise. Using the MPU 6050 allows us to filter out noise by combining data from an accelerometer and a gyroscope, discussed in more detail later.

To facilitate communication with the MPU 6050, we added an IIC IP Core to our Vivado block design. After connecting the IP block to our AXI Interconnect, we generated the HDL wrapper, which handled creating tri-state buffers for SDA and SCL and exposed the IObuf primitive. This primitive gave us the pin names to map SDA and SCL to the appropriate PMOD ports. Additionally, we added pullup resistors in the .xdc file to the data lines to ensure data integrity during transmission and reception.

Hardware/Software Interface:
To communicate with our MPU 6050, we utilized the open-source driver library provided at libdriver/mpu6050. The driver library was incredibly helpful in providing a layer of abstraction to initialize, read accelerometer and gyroscope data, and run a series of tests on the device. To bridge the gap between these available functions and our hardware, we had to write an interface that connected this driver library to our Xilinx IIC core. This involved initializing, deinitializing, performing a data read, and a data write using the functions provided in the xiic.c file. This part took time to debug and get right since we hadn't written an I2C interface before using the Xilinx IP for the Block Design. One significant roadblock we faced and overcame was figuring out that Xilinx expects 7 bit addresses for I2C and handles the bit shift itself. This driver library expects an 8 bit address, taking into account an added zero as the LSB. Once we figured out this was the culprit of our issues trying to communicate with the IMU, we were able to add a line of code to handle the extra bit shift in our interface.

Software Overview
We utilized the QPNano framework to convert our software into a state machine for a pleasant user experience (Figure 3). Our hierarchy includes an On super-state, with the HomeView, and the RideView states inside of it. Additionally, we utilized five signals to handle user interaction from the buttons and encoder, and two software signals posted on events. The On super-state handles all the user interaction signals that are thrown, and sends software signals to the "View" states when specific changes are needed. These two software signals, UPDATE_INCLINE and INCLINE_RIDE are handled in the HomeView and RideView differently.

Our code is able to asynchronously read, filter, and display the MPU data, while also maintaining fast response time from user input. Because we did not have a hardware stream grabber, we used the UPDATE_INCLINE signal that was first posted on entry to the ON. The UPDATE_INCLINE signal was then thrown and placed back into the signal queue each time it was handled. This enabled reading MPU data, and UI updates occurring as fast as the framework could handle signals. Please reference the "Challenges Faced and Lessons Learned" for more information.

Gathering and Filtering Data:

Mountain biking introduces unique challenges for measuring incline due to the unpredictable terrain and frequent vibrations. As mentioned before, using accelerometer data alone would not be possible because it provides linear acceleration, which would get corrupted by sudden movements when calculating pitch with the equation:

$$Pitch = arctan(\frac{-Accel_x}{Accel_z})$$

The video provided below (Lab Demo) showcases our device operating using only accelerometer data to compute incline and the data is incomprehensible even when small movements are introduced.

A gyroscope, on the other hand, excels in detecting rapid changes in orientation by measuring angular velocity. This makes it less affected by high-frequency vibrations, however it has no absolute reference to angle or position. To calculate orientation, the angular velocity must be integrated over time. This means that the calculated incline will become increasingly inaccurate over time if there is any drift, which a gyroscope is prone to have. While a gyroscope and accelerometer alone are insufficient, data from both can be combined to complement each other in the area each sensor lacks.

To achieve this fusion, we implemented a Kalman filter, which is designed to estimate the system state by weighting sensor data based on their respective uncertainties. We adapted an open source Kalman filter algorithm we found online to fit into our software architecture and help us obtain clean incline data.

At its core, the filter predicts the system's next incline angle using gyroscope data and then corrects the prediction with accelerometer data. During the first stage, the gyroscope's angular velocity is used to estimate the new angle while accounting for its bias:

$$angle_{new} = angle_{current} + (gyroRate - bias) \times dt$$

The $gyroRate$ corresponds to the retrieved angular velocity in the x direction and dt is passed into the function as the time between calculations (retrieved from a hardware timer). The error covariance matrix is updated in parallel to model the uncertainty in this estimate. This step ensures that the filter tracks how much trust to place in the gyroscope-driven prediction as errors compound over time.

In the second phase, correction, the algorithm uses the accelerometer measurement to adjust the predicted angle. It acts as a reference point to correct drift. The difference between the accelerometer and the predicted angle is first calculated. The innovation covariance then

quantifies the combined uncertainty of the predicted state and the accelerometer measurement. Next, the Kalman Gain determines how much weight to give the accelerometer and the gyroscope's predictions. This value dynamically adjusts, based on the relative noise levels of the two sensors. Finally, the predicted angle is adjusted using the Kalman Gain and the innovation, followed by the bias and the error covariance both being updated as well.

While the Kalman filter effectively reduced noise and drift, shown in the demo video below, sudden changes in terrain could still cause abrupt shifts in incline readings. To improve the user experience, we applied exponential smoothing to the filtered incline values with the equation:

$$Smoothed\ Incline\ =\ \alpha\ \times\ Raw\ Incline\ +\ (1\ -\ \alpha)\ \times\ Previous\ Incline$$

The sensitivity determines the weighting of the new incline versus previous incline. To obtain α quickly, we use a LUT with $\alpha\ =\ sensitivity\ lut[user\ \ sensitivity]$ (Figure 4). A lower sensitivity displays incline changes more gradually, while a higher sensitivity reacts to changes more quickly. We decided to let the user define their desired sensitivity level with the encoder dial. The user can set the sensitivity to Level 1 or 2 if they expect to be on rough terrain, or bump it up to 5 or 6 if they are on a smooth road and want more responsive incline tracking. This functionality can also be seen in our demo video below.

To demonstrate the effectiveness of our filtering, we included a No Filter mode that can be toggled with the push of the encoder button. The No Filter mode removes the Kalman Filter and the exponential smoothing algorithm from the MPU data. This mode highlights the necessity for both filters which can be seen in the demo.

User Interface

The LCD display acts as the primary interface for the trail grade monitor, providing users with an intuitive way to view their inline and ride metrics. With the "Home View" offering real-time incline monitoring and the "Ride View" providing detailed ride statistics, each came with their own challenges in displaying the desired information.

In the "Home View" we wanted to give the user a real-time view of their incline, which can range from [-90, 90], displayed as a line that has a slope ranging from [-1, 1] with the area below it colored in (see Figure 1). Not only did this require scaling a series of rectangles across the display, increasing in height by a rate defined by the normalized incline, but it also meant we needed to only draw updated parts of the screen since we needed it to display incline changes as quickly as possible. We achieved this by tracking the previous incline and writing the background color or the block color, depending on a bigger or smaller new incline, from the new calculated height to the previously calculated height for each thin rectangle. We additionally reset the color for each rectangle based on an incrementer, creating a gradient that fades from

green to red. We believe this gives useful feedback to the user, signifying a steeper incline when more red is showing, and a steeper decline with more green being present.

In the “Ride View”, our goal was to give users the ability to track a certain ride, then display information about the ride over time. To do this, we had a button to start/pause the ride, and a button to reset the ride. The first thing displayed was the current incline in the top center of the screen that was visible if you were in a ride, or not. Below that, we had ride statistics, which included the maximum incline, the minimum incline, the average incline, and the total number of incline measurements that were taken over the course of the ride. Below the ride, there is a chart that is able to live time update and display your incline over measurements. For responsiveness of the UI and quick chart updates, the chart only draws the difference between the incline values for each point in the array, simulating a moving plot over time of the user’s incline.

Both views also had similarity with the top of the screen showing current settings. The top left displayed the sensitivity value which could be 1-6, or “No Filter” for turning off the filtering algorithms. Below the sensitivity value, there is the current ride status, which could be “ON”, “OFF”, or “PAUSED” for the ride. The ride status can be toggled no matter which view for easy access to the biker. Finally, the top right displays the label for which view the user is currently in.

Challenges Faced and Lessons Learned

While we were eventually able to develop a device that met our goals, it didn’t come without challenges. Setting up the IIC IP Core in Vivado was the first difficulty we faced, as we’d never done it before and struggled to find resources online. After gaining a better understanding of the need for tri-state buffers, given I2C’s bidirectional communication, we were able to create a working hardware setup. Writing the I2C interface was the next challenge, even with the driver library we found. Connecting the functions to our IIC IP Core was new to us, but relying on the provided Xilinx library helped us understand the requirements for initialization and sending/receiving data.

Getting and displaying the I2C data rapidly also proved to be difficult. We didn’t have a hardware stream grabber setup like in the FFT lab, so we had to read the I2C data on request. Our initial attempt had a hardware triggered signal UPDATE_INCLINE (U_I) thrown every 1ms. The signal was handled in the super state, and read the I2C MPU data, applied the filtering, and updated the cur_incline variable. However, this introduced a queue overflow problem where QP_nano would fail based on too many signals thrown in the queue. We started slowing the U_I signal, and optimizing parts of our code, but our system was always breakable with fast and repetitive button presses, filling the signal queue. Finally, we realized that timer based signals were not the way to update the cur_incline variable, and pivoted towards an asynchronous methodology.

To solve the problem, we called the U_I signal only once on initialization, and threw the U_I signal each time the U_I was handled. The U_I signal gets the cur_incline at the fastest rate possible, depending on what other signals have to be handled in the signal queue. For example when a ride is in progress, the cur_incline is updated slower due to the signal queue handling storing and displaying the ride information. When we implemented this change, our UI became much more responsive, and we were able to display the incline at much faster speeds (10-100x) which was incredibly rewarding.

Calculating and filtering the incline data came with its challenges as well. The initial setup of the Kalman filter provided some improvements, but wasn't nearly stable enough to match our requirements. We thought of implementing a Schmitt trigger, but decided it would be better to take an approach that takes the current incline into account and pushes it in a direction that is influenced by the newly calculated incline, rather than directly assigning it. We then discovered that exponential smoothing does exactly that and saw significant improvements when integrating it.

Future Steps

We believe the current implementation of the trail grade monitor successfully demonstrates the core functionality of real-time incline measurement and user interaction. However, there are several ways we believe our project can be optimized in performance and usability.

First, we could explore more advanced filtering techniques to improve responsiveness, while still filtering out noise. Our current implementation introduces a tradeoff between fast incline updates and minimal noise. Allowing the user to get closer to having both at once would be optimal.

After ensuring the prototype on the FPGA withstands rigorous testing, we can design a PCB that uses only the essential components to get a more compact device. After creating housing that can mount onto a bike's handlebars, we can go through user testing and get a better understanding of how it performs on an actual ride.

Conclusion

Developing our trail grade monitor offered an opportunity for us to apply what we've learned in ECE253 into a practical application where we focused on meeting tight real-time constraints to create seamless user experience. We had the chance to research and evaluate algorithms that could optimize our product's performance, such as Kalman filtering and exponential smoothing. Finally, we gained valuable experience working through problems where an answer wasn't certain to exist, so we had to create hypotheses and test them.

Appendix

Lab Demo: 253FinalDEMO.MOV

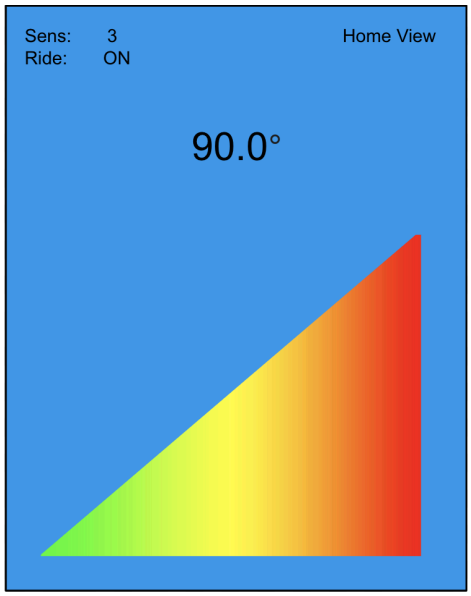

Figure 1. Home View Mock Up Drawing.

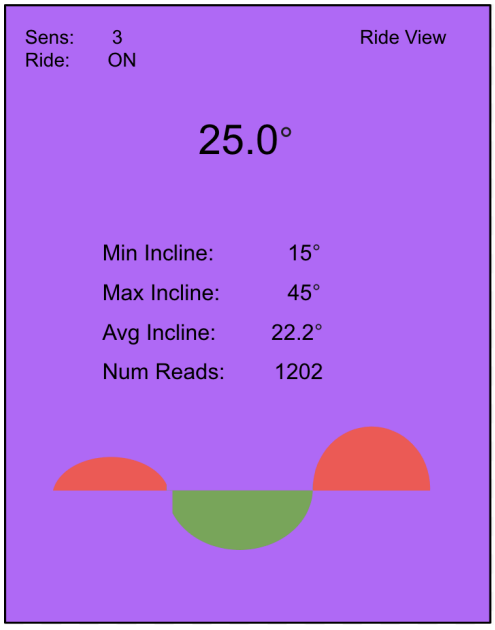

Figure 2. Ride View Mock Up Drawing.

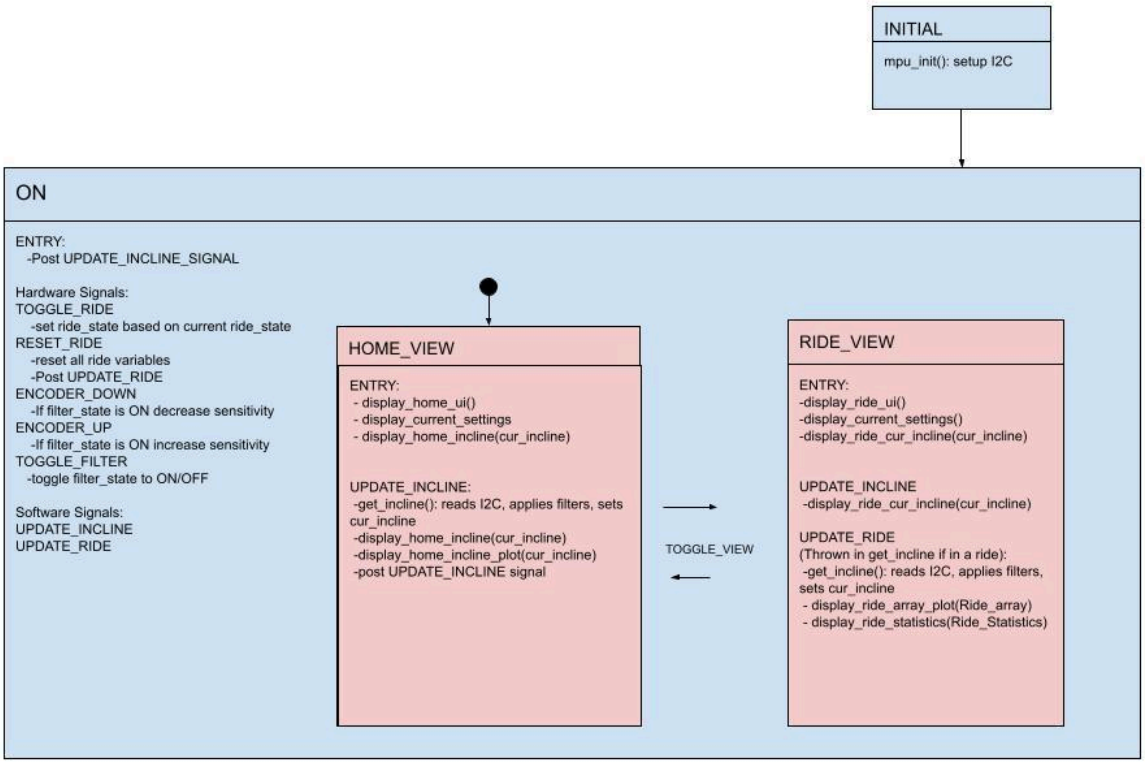

Figure 3. Software State Machine.

```
float sensitivity_lut[SENSITIVITY_LUT_LENGTH] = {
    0.0005f,
    0.005f,
    0.05f,
    0.25f,
    0.5f,
    1.0f,
};
```

Figure 4. Sensitivity LUT for α for Exponential Smoothing