Group 309
Ryan Zimmitti & George Kopf
rz6542@princeton.edu & gk3946@princeton.edu

For speed control, navigation, and the final project, your team should upload two files to Canvas (one report per group): **Lab report (PDF)** & **A .zip file containing all code & designs** (including .cydsn for PSoC, .ino for Arduino, etc.)

Statement of Objective:

Statement of Objective (5 pts.) We want to build an electronic LED lightsaber with fully functional lights and sounds. Addressable LEDs allow for precise control of extension and retraction animations. We will use an internal gyroscope sensor to link audio cues to swinging and hitting motions. We will also use angular acceleration data to modulate the brightness of the blade and audio output for dynamic response.

Overview of Key Subsystems & Components:

Overview and brief explanation of key sub-systems and components (30 pts.)

Arduino Nano

Our Arduino Nano was the main brains of the lightsaber. We went with an Arduino because of its broad capabilities and strong collection of libraries. This allowed us to set up the IMU and the DFPlayer into our software quite quickly. Moving away from PSoC took us from analog circuitry to digital processing, which was a big sacrifice in terms of raw speed. However, the Arduino Nano has one huge advantage that PSoC could never give us: it's tiny. The hilt of a lightsaber is a very restricting space, and it would be impossible to physically fit a different microcontroller inside. Using the Nano allowed us to avoid having to attach some sort of tether or umbilical cord to the lightsaber, which would not have represented the essence of a true lightsaber. In the event we required higher processing power, we could switch from a Nano to a Teensy, but the Nano was powerful enough for our lightsaber (and much cheaper than a Teensy).

The Nano receives power from a DC/DC boost converter attached to the battery. It sends signals down pin D4 to the addressable LED strip, determining what color and brightness each individual LED will shine with. It sends and receives signals to and from the DFPlayer via the TX and RX lines respectively on D11 and D10. It also receives the DFPlayer's active status via the busy pin on D9. The IMU's SCL signal is plugged into A5 and the SDA signal is plugged into A4. And lastly, a button runs between pin D3 and ground. See *Schematic Diagrams* for the circuit diagram.

NeoPixel LEDs - WS2812B

WS2812B LEDs are digitally addressable RGB LEDs. Each unit on the strip of 144 contains three LEDs (red, green, and blue) and its own personal tiny IC, which interprets data sent from the microcontroller. The strip works as a daisy chain of smart pixels. The Nano sends out a continuous stream of digital pulses through the signal wire into the first LED. It reads the first 24 bits of the message (latches data into internal registers), then forwards the rest to the second LED. This repeats down the entire strip. Thus the LEDs avoid having any sort of clock line or index – addressing is implicit by pure position in the data stream.

Each unit reads in 24 bits as three color channels. The first 8 bits code for green, the next 8 code for red, and the final 8 code for blue. These 8-bit values code for a value from 0 to 255. Each of the three LEDs in each unit then lights up at the appropriate value. To vary brightness, they use PWM to flicker the lights at different rates set between 0 and 255.

What's nice about the NeoPixels is that they use a single-wire, self-clocking protocol based entirely on pulse timing. Each value takes a set amount of time (roughly 1.25 us). If the signal value is low for most of the time, it's a 0. If the signal value is high, it's a 1. As can be seen in the *Data* section, oscilloscope outputs of this signal show these individual bits flickering between short and long pulses – indicating that 0 and 1s are being sent through.

After all bits are sent, the microcontroller sets the data line to 0 for about 50 us, which tells all the LEDs to stop listening for commands and update simultaneously. The strip will remain visually constant until the next update. The FastLED() library used in our software essentially just automates this process in the code:

```
for each LED {
        send 8 bits green
        send 8 bits red
        send 8 bits blue
        hold line low for 50 us
}
```

In our project, the LEDs are set to flicker seemingly randomly like a real pulsing lightsaber. To create a realistic pulsing effect, we use a sine wave with random 'jitters' in value as one factor in our brightness calculation, giving a pulsing/flickering appearance. The other factor is the IMU values, which proportionally controls the brightness. Adding these factors together creates an effect where the faster you swing the blade, the brighter it will be while keeping a pulsing appearance. This creates a very 'realistic' lightsaber blade effect, and while it can be subtle, it makes the lightsaber feel very alive when being held as it responds dynamically to movements. Additionally, there is code to generate an impact effect when the lightsaber is 'struck' (high IMU values), as the white flashes enhance the bright pulse of the blade.

Inertial Measurement Unit - MPU6050

Our build uses an MPU6050 in order to sense angular velocity as the blade is swung around. To measure acceleration, multiple tiny proofmasses are suspended by small springs, one for each axis of measurement. One each side of them are capacitive plates. When acceleration occurs, the masses deflect, changing the capacitance and triggering internal circuitry (analog voltage shifts). An Analog-to-Digital block inside the IMU converts this to a digital signal that represents the amount of deflection. To measure rotation with the gyroscope, each axis contains a vibrating proofmass that is driven by an electrode at a steady state. When rotated, the Coriolis effect changes the vibration patterns, triggering sensing electrodes that detect and record this shift in vibration. Again, an ADC block converts this to a digital signal.

Our Nano interfaced with the IMU through two channels: the SCL line is the clock, and the SDA line is the data. Unlike our LEDs, this line is clocked because of the precision of data being sent through. As can be seen in the *Data* section, the SCL line carries repeating clock pulses, dropping from low to high. These are created by the microcontroller and sent through to the IMU. SDA shares bidirectional data flow from the microcontroller and the IMU, with each taking their turn as they go. Both lines are internally pulled up (open-drain/open-collector MOSFET setup). The fundamental rule is that the SDA line can only change while SCL is low. Functionally on the oscilloscope, we see this as SDA switching each clock pulse pulls the SCL voltage down to 0. There are only two exceptions to this rule: start and stop conditions.

To send the “start” signal, SCL and SDA start high in their idle state. The microcontroller then explicitly drops SDA to low. When this happens, the IMU realizes it is being triggered. Every device on the bus in idle will be watching for SDA to fall while SCL remains high.

Next, the microcontroller sends a 7 bit address followed by a read/write bit. To transmit bits, the microcontroller changes what SDA is while SCL is low, allowing it to send data that's linked to the timing of the clock. SDA’s value after SCL returns to high is recorded as a bit. After this data burst, the microcontroller releases the SDA line, allowing it to float back to high. The IMU will dip the SDA line in response as an acknowledgement of a successful transmission.

After this handshake, the microcontroller sends a full byte detailing the register address that the IMU should use (it sets an internal pointer in the IMU). These bytes make up the majority of the data bursts shown in the oscilloscope outputs in the *Data* section. It then sends the “start” signal again by dropping SDA while SCL is high. Subsequent behavior depends if the microcontroller sent a read or a write bit earlier during the handshake.

In a write transaction, the microcontroller drives SDA, configuring the IMU and identifying registers. In a read transaction, the IMU drives SDA, transmitting the acceleration and gyroscope data it has collected. It runs through the addressed register and outputs the values stored there. After each byte, the listening party confirms with the sender by dropping SDA low.

To end this round of communication, the microcontroller sends the “stop” signal by pulling SDA high while SCL is high. This signals the IMU to go idle.

DFPlayer

The DFPlayer is a UART-controlled finite state machine. It has a built-in audio encoder and Digital-to-Analog block, as well as its own small amplifier (which we used). Because all audio data is held on the microSD card in the chip (we had .wav audio files), the microcontroller never streams audio. It simply tells the DFPlayer what to do over UART.

TX is the transmission line and RX is the receiver line (from the microcontroller's perspective). Both run through 470 Ohm resistors between the Nano and the DFplayer, which limit current (overcurrenting can cause latch ups) and help smooth edges. Every DFPlayer command is sent through by the microcontroller as a 10-byte chunk. The DFPlayer does nothing until all 10 bytes have been received (and it ignores the command if any bytes are corrupt). Each command shifts the internal state of the DFPlayer. In this project, the commands we used were Play [track], Stop, Loop, and Set Volume. The DFPlayer chip contains all the circuitry and logic for handling the actual audio playing. It has its own internal processing and functions as a tiny microprocessor on its own – think of it like an autonomous CD player. The microcontroller sends its commands, but the DFPlayer receives and executes them by itself. The busy pin is internally pulled up, sitting high when idle. While the DFPlayer is playing an audio track, the busy pin is set low. This is particularly important during the hum loop condition in our lightsaber. While idle, the blade should hum passively (like a real lightsaber). However, this track is only 10 or so seconds long. When in the hum state, as long as the busy pin reads low we know we're still playing the hum audio. Once busy pops back up to high, we can immediately command DFPlayer to play the hum track again. The gap between the two ends of the audio are noticeable if you know what to look for, but luckily the pause is mostly obfuscated because the audio file is intentionally full of random cracks and pops that break up the continuous drone. The busy pin is very consistent and allows us to directly check in on what DFPlayer is doing – it even functions if RX is disconnected.

Usually, there is a 10-byte acknowledge response over RX after every command where the DFPlayer acknowledges it has received, but we disabled this by using a custom library called DFPlayerMini_Fast. This allowed our Nano to ignore whether or not the DFPlayer had made its acknowledgement, which made the program run noticeably faster. By using the busy pin instead, we avoided having to wait for this handshake without sacrificing functionality.

The audio output is sent as a compressed burst of data to a decoder before the DAC converts it to analog values the speaker can play. The internal amplifier then amplifies these analog values to a more reasonable level, although it is much weaker than typical amplifier blocks are. Thus the audio on the SPK outputs of the DFPlayer can be a little quiet. To offset this, we used an 8 Ohm 3W speaker instead of the 1W speaker we had initially planned to use. The higher wattage meant we were able to play louder, but we were still limited by the amplifier.

We encountered a major issue when using the DFPlayer that threatened the viability of the project; DFPlayer is very sensitive to power dips. Because of the nature of the lightsaber, we wanted to play an extension audio sound at the same time as the LED strips were igniting when the button was pressed. However, the LEDs draw a lot of current, and would temporarily dip our

voltage when igniting as the supply compensates for the increased draw. This exactly corresponded with when DFPlayer needed to be receiving its 10-byte package, which resulted in the command failing to send completely and DFPlayer refusing to activate. Sometimes it would start and then immediately brown out instead, just playing random noise instead of the correct track, likely due to the power dropping temporarily below the minimum threshold to run the DFPlayer. When power returned, the DFPlayer would be stuck not knowing where it left off and blast out static instead. This was frustrating, but we ended up solving it by adding a 1000 uF capacitor across the DFPlayer VIN and GND pins (as close to the chip as possible) and a 0.1 uF capacitor across the speaker + and - pins. These capacitors helped smooth our sharp current draws, keeping the 5v rail stable enough for DFPlayer to run simultaneously as the LEDs were changing.

Power System

The VIN on the Nano requires 7 to 12v to operate. An internal voltage regulator lowers that to the board's internal operating level of 5v. This higher VIN gives the regulator headroom to stabilize the incoming power, so the Nano receives a consistent 5v.

The MPU6050, DFPlayer, and WS2812B LEDs all operate with 5v inputs. However, because of the high amperage draws of the LEDs, it was not feasible to directly power them from the Nano's 5v VCC pin. Because LEDs draw so much power to run, especially in brighter colors, we had to choose a very energy dense lipo battery to power the LEDs directly. This can get awkward, because now we need a 7v *and* a 5v battery. The simple workaround is to use a 7v battery to power the Nano, then use a buck converter (that is spec'ed for higher amperage) to step down to 5v for the LEDs. However, at this stage we ran into an unusual problem: the hilt of the lightsaber is tiny. With such little room to fit electronics, we physically could not fit any of the batteries that had both high enough voltages and enough amperage to run the LEDs for a reasonable length of time. The goal was to create a product that could be used consistently over the course of an event without needing constant charging.

This is where we decided to cheat somewhat. We ended up choosing a 18650 lithium polymer battery with 9800 mAH of power at 3.7 volts. This was the most powerful battery we could find that fit into the hilt of the lightsaber without compromising our capacity to fit the rest of the circuit. We then connected it to a boost converter that could step our voltage up to 5v. We ran this directly into the VCC pin of the Nano, which meant we were powering the Nano directly from its 5v line. This was on the other side of its internal voltage regulator, which also meant we were bypassing the Nano's internal protection circuitry. Large bumps in the voltage level would negatively impact the microcontroller. To combat this, we did careful testing of our full circuit under load and found it to be quite safe in most conditions. The only situation where we identified that the power line to the Nano had a risk of blipping and causing a reset was when the LEDs were set to full white at max brightness while being rapidly turned on and off in unison. In this configuration, enough rapid switching could cause a big enough blip to reset the Nano.

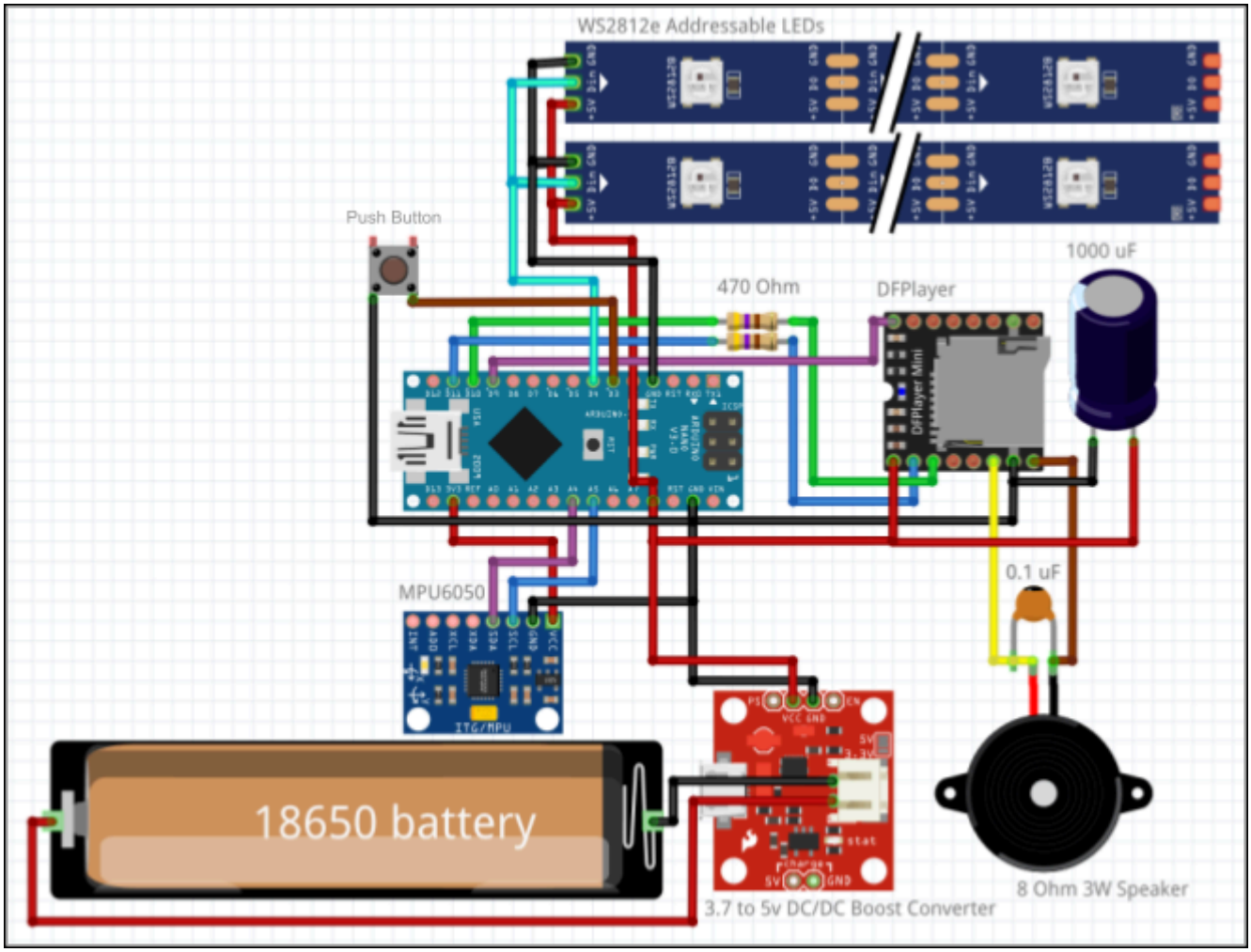
Luckily, this was easily worked around. First of all, the LEDs were set to never come on in unison. The natural extension/retraction animation of the lightsaber meant that LEDs were always set to turn on sequentially. We also programmed in a sweeping animation for the color changing functionality, so LEDs wouldn't rapidly switch between colors either. Secondly, we removed the option to use the color white from our color selection. This wasn't strictly necessary, but we wanted to make sure the lightsaber was fully safe. Lastly, we also considered attaching a small capacitor between the VCC pin and ground, but in the end concluded that this was not worth the space it would occupy (adding an additional element to the Nano's pin would require adding some kind of PCB to connect the lines, which would be a big sacrifice of the limited space inside the hilt).

Using the DC/DC boost converter that we chose also gave us the benefit of built in charging circuitry (based on the SW6008 IC), allowing us to use the port on the converter board as a charging port. This was critical because it meant the lightsaber was rechargeable without having to fully disassemble the hilt to reveal the battery itself.

The converter we used also had a configuration to enable an integrated protection circuit, which would protect the lipo from a short higher up in the lightsaber's circuitry. Lipo batteries have a high risk factor of igniting fires when shorted, meaning that enabling this protection was invaluable to our build's safety – especially on an item built to be swung around and hit into things.

Schematic Diagrams:

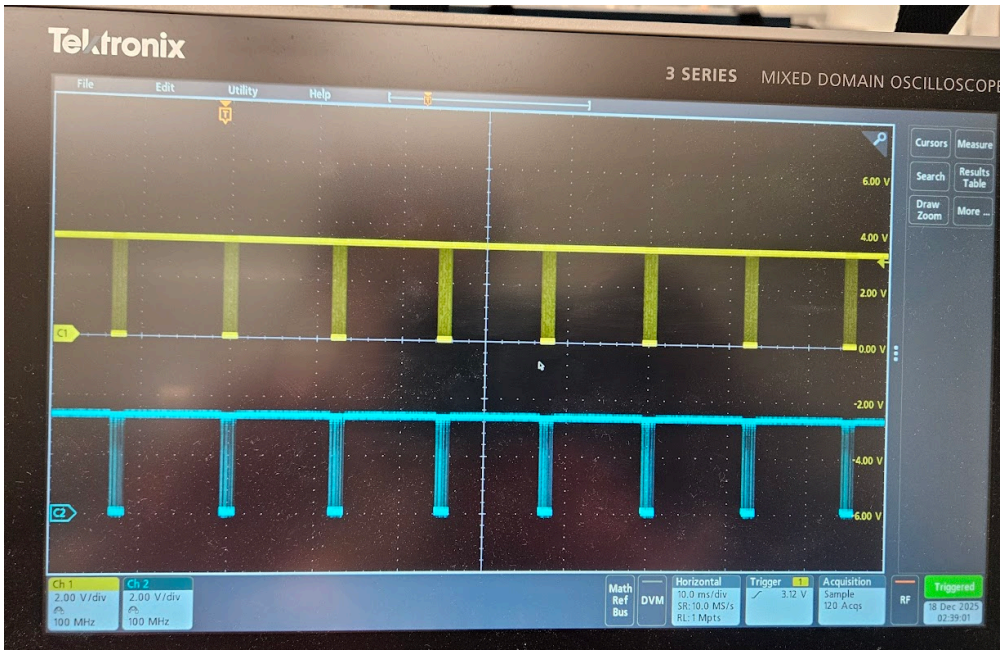
Schematic diagram of circuits (computer-drawn, try CircuitLab, etc.) (15 pts.)



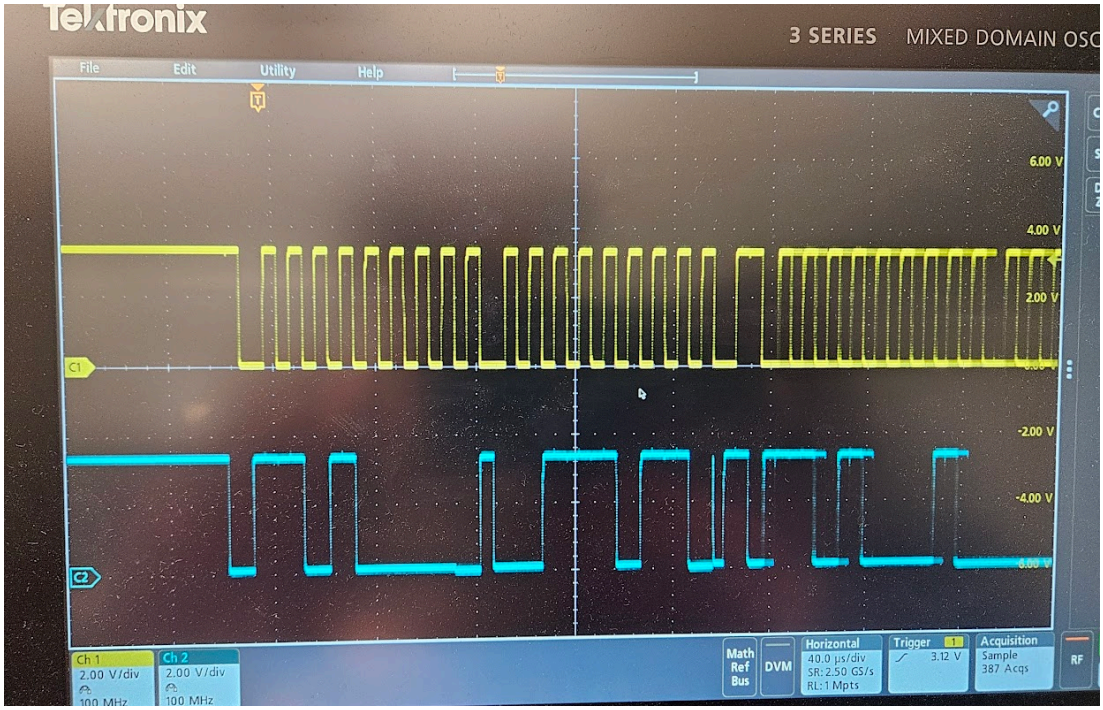
Data:

Data showing that it is working as you expect. Include explanations for this. (25 pts.)

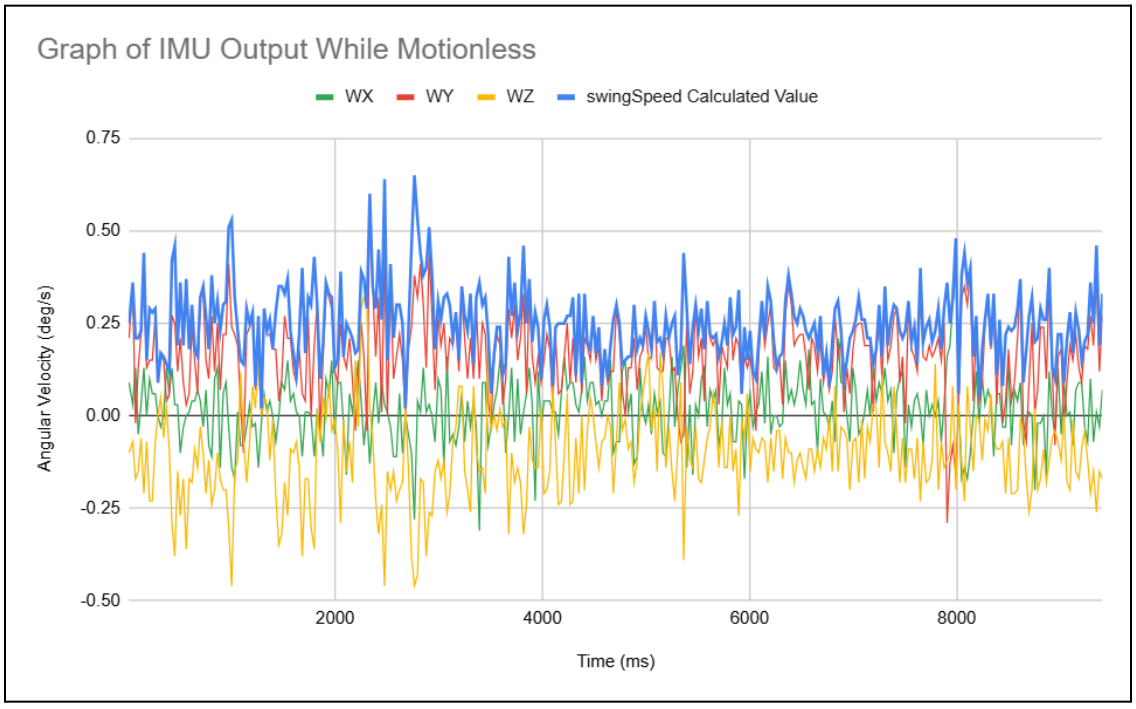
IMU Data:



Channel 1 is the SCL clock line, and channel 2 is the SDA data line. As discussed in the *Overview* section, SCL sends bursts of clock pulses while the microcontroller and IMU are communicating. This times the data line, telling the processors how to interpret the data being sent over SDA. If we zoom in a little further we can see a clearer picture of what's happening during these bursts of information being sent back and forth.

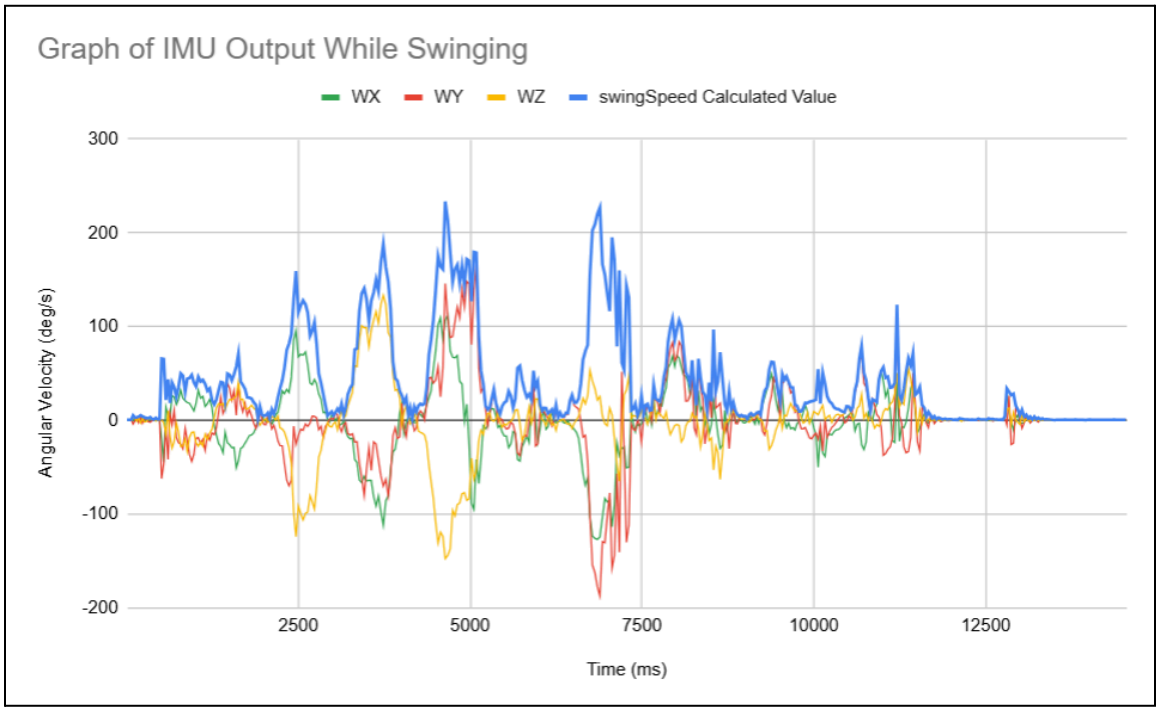


Zooming in, we see the first thing that happens is SDA drops low while SCL is still high. This is the only time that SDA changes while SCL is high (except for at the very end to signal "stop"). This signals "start". Then, SDA is modulated when SCL dips low. It only changes while SCL is low. The value of SDA after SCL returns to high corresponds to the 0 or 1 being transmitted. As can be seen, SCL is not perfectly consistent. This is likely due to clock stretching, where the IMU extends clock pulses to give it time to keep up. This doesn't affect functionality.

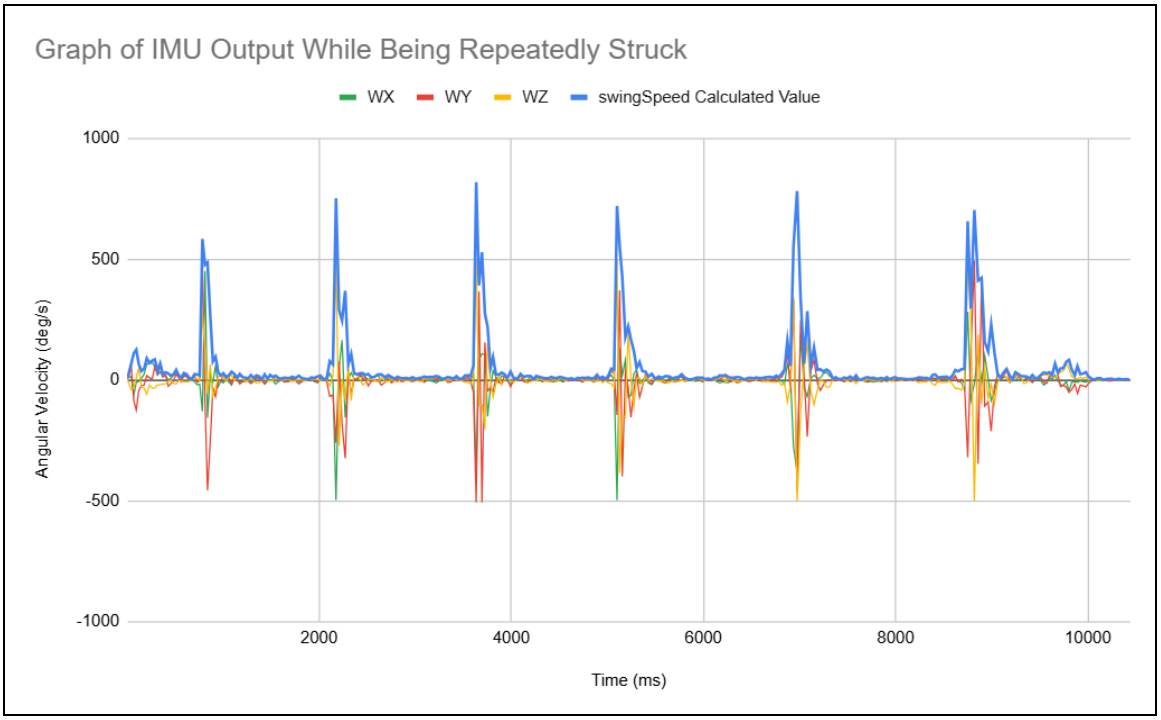


This graph shows the angular velocity of the three axis as well as swingSpeed, a variable calculated from all three to create a total angular velocity term (swingSpeed = sqrt(wx*wx + wy*wy + wz*wz)). This swingSpeed value is the main signal of the IMU that we're reading and is used to identify what range of motion is happening to the blade, as well as to what degree that motion is happening at (a light swing and a heavier swing affect the brightness of the blade differently because the sine wave that drives brightness level is proportional to swingSpeed).

In the above graph, we see baseline noise for the IMU while the lightsaber is laying motionless on the bench. It's quite accurate, even though it looks noisy; the y-axis shows us our passive error is about half a degree per second in either direction. This is tiny compared to the values we work at, so the IMU can be assumed to have negligible error.

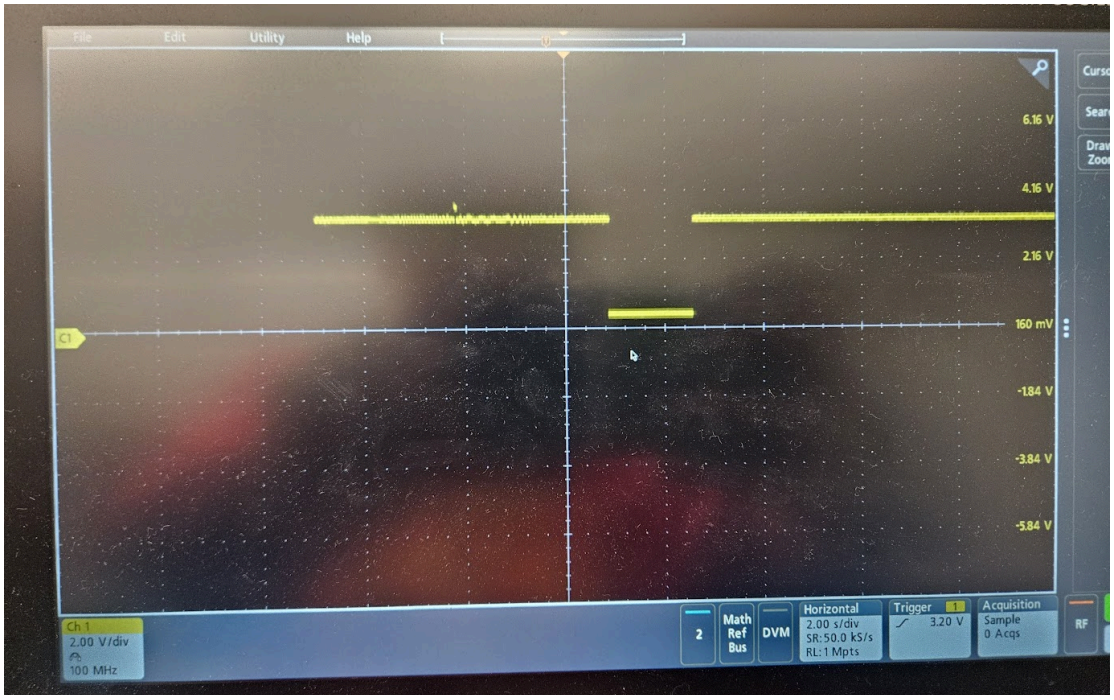


This graph shows the IMU while being swung around slowly. Even passively holding the lightsaber puts our deg/s into double digits, but large swings can clearly be seen in the data. Four main swings done during this test show four clear peaks from 2500 ms to 7000 ms, ranging from 150 to 230 deg/s. Using this data, we calculated the range for a "medium" movement to be from 130 to 380 deg/s. When in this range, the lightsaber will trigger swinging noises and pulse slightly brighter.

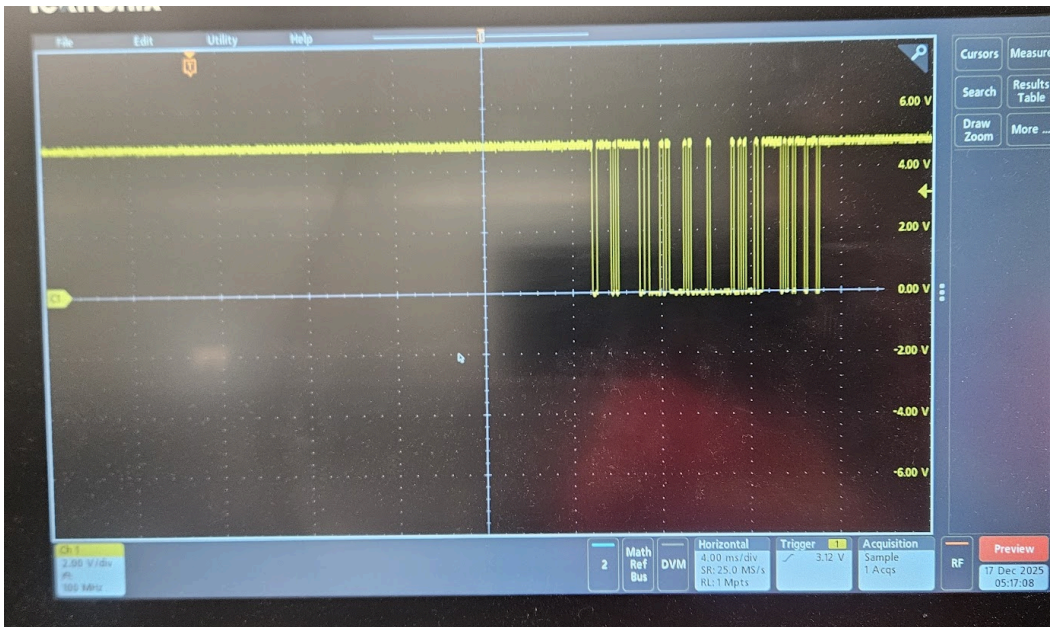


This graph shows the IMU while the blade is being firmly struck. The six strikes are clearly visible on the graph as massive and sharp peaks, all spiking over 500 deg/s. Our threshold for swinging is set to 380, so most physical strikes on the blade should easily be enough to trigger a “large” movement. When in this range, the lightsaber will trigger clashing noises, pulse brightly, and flicker with white pixels.

DFPlayer Data:

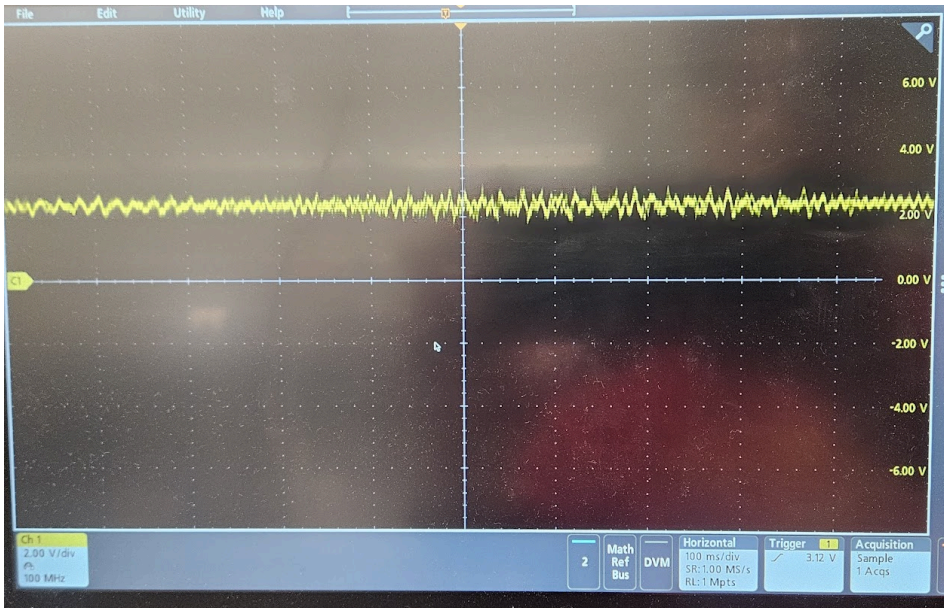


This first graph shows the Busy pin. The signal is pulled up to high by default, but when playing audio it drops low. In this scope trace, the DFPlayer was set to manually play a short sound bite, and clearly showed up on the trace as that low block in the signal.



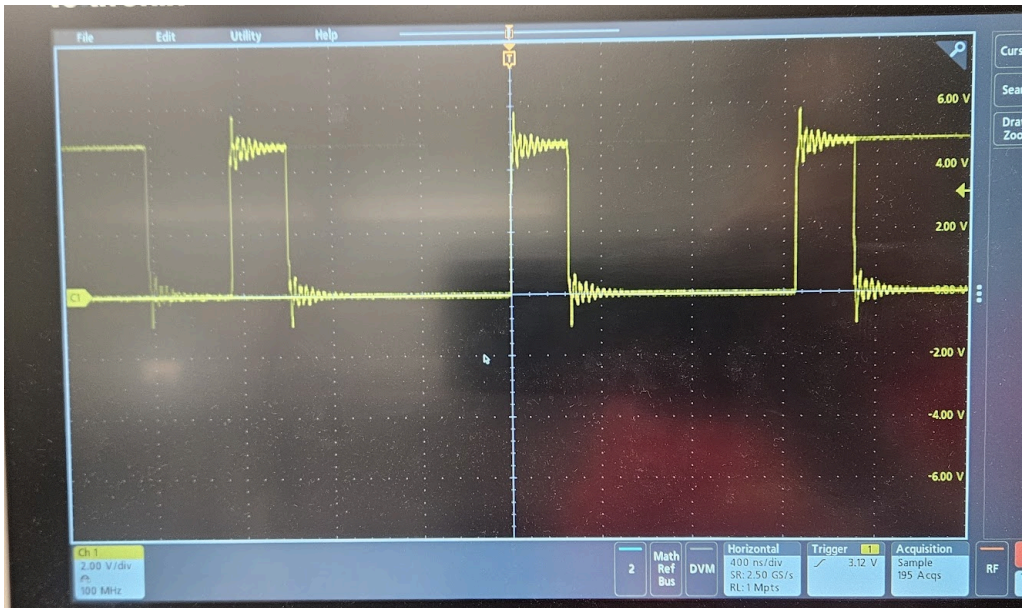
This scope trace is the TX pin. It shows a burst of 10 bytes of data, the microcontroller transmitting a command to DFPlayer. This command should be the Play Track_1 command. Once all 10 bytes of data are sent, only then does the DFPlayer react to the instructions. If any of the bits are corrupted or don't send through, the DFPlayer will ignore it.

The DFPlayer acts as its own semi-autonomous microprocessor, meaning it receives straightforward UART commands and makes its own decisions about what to execute. It handles the audio data on the microSD card and sound functionality without needing to consult the microcontroller. These 10 byte commands are all that are required to get full functionality.



This scope trace shows the positive output line to the speaker. These rapid electrical pulses mimic the actual sound waves, controlling the speaker's cone to push air (PWM or DAC to convert from digital to analog). This signal is already amplified by the DFPlayer's internal amplifier, which is what gives it enough current to move the physical speaker elements. The microcontroller itself doesn't have the power to do that without an amplifier.

NeoPixel LED Data:



These scope traces show the signal input line of the WS2812 LED strips. A single signal wire communicates all the necessary data to operate the series of LEDs along the strip. As discussed in *Overview*, each pixel on the strip is actually three LEDs of R, G, and B along with a tiny IC that interprets data and controls the LEDs. Data is sent through the signal line in bursts of 24 bits. These bits are identified by the signal's built-in clock timing. Like morse code, short

bursts are zeros and long bursts are ones. The entire signal is defined by 1.25 us clock intervals (note this is very visible on the 2 us division scale in the second image) that the microcontroller and LEDs have hardcoded. The microcontroller just sends long chains of continuous data. The first IC receives the full chain of data, reads off the first 24 bits, then passes the rest along. The second receives the rest, reads off the next 24 bits, and passes what's left. In this way, all 144 LEDs in a full strip receive their instructions.

These 24 bit commands take the form of three 8-bit PWM values from 0 to 255. In the order of green, red, then blue, the LEDs pulse to modulate their brightness on that 8-bit scale. Thus, individual pixels on the strip can be specifically controlled, and the difference between setting them all to one color and setting each one to a unique color is minimal in terms of processing power required. This is a highly efficient way of communicating with the LEDs, and it's enabled in software by the FastLED library in Arduino and other similar libraries elsewhere.

Discussion:

Discussion of challenges and design choices you made (10 pts.)

Challenges:

Challenges we encountered on the project included hardware troubles, software bugs, and physical design limitations. In software, one challenge we encountered was strange behavior on startup where commands would not be received by ICs as expected. To account for this, we added a delay to the startup time for the ICs (1 second) in the setup portion of the code. Other coding issues surrounded the DFPlayer, which would misorder or not find certain audio files that were requested by the Arduino. To map the correct audio files, we iterated through an array to find the indexes of the audio files as ordered by the DFPlayer and set constants associated with the array indices.

Light and power were sticking points in our project as well. To make LEDs that were bright enough to have responsiveness to IMU measurements, we needed sufficient current capabilities. This is why we chose the LIPO battery and bypassed the Arduino's internal power circuitry, which could not supply the power necessary to light the LEDs. During testing, it was difficult to keep the LEDs running because of the power draw. Since our battery did not ship quickly, we used the bench power supply to temporarily power the LEDs and arduino, disconnecting the LEDs when necessary to code. We improvised a quick-disconnect connector on the LED strip for this purpose, with the wires for one end coming up through the hilt, and the other connector side being a much smaller distance from the LED strip itself.

The main limitation of our setup came from size constraints. The hilt had a small inner diameter and the length was also reduced as we needed to nest the tube for the LED strip inside the main cylinder. This manifested in cutting corners (literally) on our hardware. The speaker itself we removed from its plastic casing, and the perfboards had to be cut right up to the edge of the components we soldered. Any wires that could be shortened were, but there was still a tangle

of crossing wires that had to be compressed. Shorting and board breakdown was a natural result of this issue, and each wire had heat shrink at connection points. All boards were also wrapped in electrical tape to prevent shorts. Strained wires still caused a couple issues as well. The Arduino pin for sending data to the LEDs was not doing so correctly and caused strange colors to appear. We traced the issue to a ripped pad on the Arduino itself, moving the wire to relieve the strain on it. The audio wires for the speaker also broke multiple times due to strain, requiring entirely new wires to be soldered directly to the speaker.

A smaller issue we encountered was with power delivery. Due to the small space, we could not simply connect many positive and ground wires. As there were many components with power needed to be supplied, we improvised a 'terminal block' of sorts to route power through the hilt. This consisted of a long and thin perfboard with two rails, one for +5v and one for GND. Components then did not need long wires that traveled up and down the hilt to route power to, simply short wires to the terminal block.

Design Choices:

The choices we made for the design reflect the realism we wished to achieve compared to a real lightsaber. In the visual system, we designed it to mimic the flickering, modulation, and hit effects of a lightsaber. The audio system combined multiple audio files to provide responsiveness and support for the visual system to create an integrated experience.

As discussed in the *Data* section, we chose specific medium swing vs heavy impact thresholds on our calculated IMU readings. We wanted to create a very specific user experience where the blade would hum when idle, make buzzy woosh sounds when swinging around, and make sharp clash sounds with bright flashes when physically hit. We used careful data collection combined with subjective use testing to narrow in on thresholds we were happy with: 130 to 380 for medium swings and 380+ for heavy impacts. This actually ended up being slightly on the sensitive side, as in the end we sometimes would trigger heavy impact responses when swinging the blade around quickly. Bumping the threshold to over 400 likely would mitigate this.

Initially, during the setup() block of our code, the IMU had to calculate offsets for several seconds. This behavior could also be seen in other projects; I remember Noah Hiers and Rose Weathers' car had to wait several seconds before parallel parking to calibrate. This is fine for a car on the ground, but for a lightsaber, it's impractical to have to hold the blade perfectly still while the IMU is calibrating. Small movements of the hand can throw off IMU values, shifting the movement thresholds for our control system. To solve this, we took advantage of our unique requirements: we only needed gyroscope data. Because we weren't measuring angle and were only looking at angular velocity, we didn't need the linear accelerometers at all. Calculating offsets is still important for the gyroscopes, but because total angular velocity can ignore the inertial frame of reference it's measured in, we could calculate offsets with a perfectly still IMU beforehand. Using those precalculated offsets (code included in the attached .zip file), we could initialize the IMU quickly and without having to wait for an awkward several seconds every time

we wanted to turn on the lightsaber. We could also be holding it, ready for action, before activating it without worrying about ruining IMU readings.

Next, our project was initially approved with the simple control element of using IMU data to switch between medium swing noises and heavy impact noises. However, we felt as if we were lacking a more complex control system and knew we could make the lightsaber a little more interesting. Thus, we decided to implement a function where the average brightness of the blade (average because the entire thing is constantly flickering) would be scaled to a sine wave. This helped make the blade look a lot more lifelike compared to the random jittery flickers we had before, but we weren't done yet. As discussed in *Overview*, we used our angular velocity measurements to proportionally scale the brightness of the LEDs. This proportional control system made the blade feel much more lifelike, as it would literally glow and respond as you swung it around. The faster you moved it, the brighter it got.

Unfortunately, in order to make this work we had to sacrifice some overall brightness. Before, the blade was constantly just set to glow as its highest level. However, in order to have a brighter response relative to a dimmer one, we had to make the default state of the lightsaber dimmer. Too dim and the effect of the LEDs was ruined, so we did some tuning to find the right balance between minimum dimness when idle and noticeable brightness when swinging. In the end, we leaned towards the brighter end when idle, which made the proportional response effect hard to see in a well lit room. That's why during the demo, we had the lights turned off (and for the cool factor too of course).

Improvements:

What could be improved (10 pts.)

There are a number of improvements to our project that would add to the realism of the blade and improve the durability for repeated use. On the coding side, our algorithm checked for how hard the blade was being swung by examining peaks in total angular velocity. However, we have more data available to us to determine more exact timing for striking and swinging behavior. Currently, the blade averages the angular velocity in all directions, so shaking the blade vertically up and down would trigger the swinging sound effect. To mitigate this we could write a more complex algorithm that takes into account the rotation and direction from the IMU to only playing swinging sounds during a certain range of directions. Another change would be the smash sound effect. Currently, the effect is played at a peak in angular velocity, but a hit (smash effect) should come after the peak in velocity (large change in velocity). We could set a flag at the point of a peak threshold we determine and wait for a large drop in velocity indicating the actual hit to trigger the smash sound effect.

The audio system is also not ideal. Physically, the hilt was closed off and the speaker put too far inside for optimal sound projection out of the blade. Mounting the speaker in the side of the hilt or directly on the bottom would allow for better sound volume. Our speaker is also

underpowered by the small amplifier in the DFPlayer. Adding a dedicated amplifier circuit after the DFPlayer would have alleviated this issue. The DFPlayer itself is not an ideal audio decoder, and was the cheapest one we found on Amazon. Investing in a more expensive audio file player would be necessary for a streamlined audio processing experience. Adding proportional control to the audio volume based on IMU measurements would also reflect a more realistic blade, as a smashing sound would be louder than swinging and idling.

LEDs are the main feature of our project, but the visibility was somewhat lacking in terms of brightness. The tradeoff with brightness is of course how amplified the swinging effects are, as reducing the baseline brightness makes peaks more impressive but dims the idle flickering effect and vice versa for increasing baseline brightness. What we aren't sure of is whether the brightness limits are due to our battery not being able to supply the current, or simply the LEDs not being powerful enough. Adding more LED strips (or replacing them with higher wattage strips) would certainly allow for better visibility, but could require more batteries, or reduce how long we could run the blade before needing recharging.

In our implementation, we were severely limited by the diameter of the hilt, requiring us to cram multiple boards very close together. Instead of electrical tape and hope, our solution could involve combining as many circuits as possible onto one or two custom boards designed to the space of the hilt with a custom enclosure to prevent conductance to the hilt itself. The arduino, MPU, and boosting circuitry could be designed for one board if we sourced the individual chips and only implemented the necessary features in hardware, reducing the physical footprint compared to the boards we received. Machining the hilt to a smaller inner diameter is also reasonable as the walls are significantly thicker than needed.

A significant design change, but a more realistic feature could involve a physically retractable blade. There are a number of ways to accomplish this. Some real-life lightsaber designs use real flames and combine gases together. For us, to keep it lab-safe, we could use motors with curved plates akin to tape-measures that snap closed and roll out with a small motor, but this would break easily with hard enough hits. Using concentric rings that spring outward and retract in through an inner tensioning string would be another option, with a need for LEDs that collapse with the rings or ride along the tensioning string.

Adequate comments in your code (5 pts.)
See submitted code ✅