1600 Final Project: Motion-Based Synthesizer

Group Members: Sophia Lim, Asia Nguyen, Roberto Gonzales Matos, Julie Chung, Sarah Richman
Github: https://github.com/RobertoGonzalesMatos/HandMotionSynthesizer/tree/main

A short introduction giving an overview of your project and what assumptions you are making about the user.

Our project is a motion-based synthesizer that allows users to control sound purely through hand orientation, gestures, and finger contact. Using a glove embedded with an accelerometer, gyroscope, and capacitive touch inputs, the user can generate and manipulate musical output. Vertical hand motion (rotation over the X axis) influences pitch, oscillation (rotation over the Y axis) adjusts vibrato, and rotating hand movement along the Z axis will silence the output. Additionally, finger-to-thumb contact triggers up to 3 harmonies. We also have an interactive UI that allows you to switch to a drum environment with 6 gesture-based drum sounds. The interface also complements the hardware instrument by displaying the musical score with active notes, and enables recording and looping.

The system assumes that the user is able to perform basic rotational and translational hand gestures while wearing our embedded glove. The user is not assumed to have prior experience with music, but should be comfortable interacting with a simple UI workflow and all the features accompanying it.

Revised and fleshed out requirements from part 1 of the progress milestone report.

Core audio control requirements:

1. Pitch Changing: The system shall map the accelerometers' X-axis angle to pitch such that an increase of 7.5 degrees results in an upward change of 1 note. (180 possible degrees over 24 notes)
    a. Confirmable by: holding the hand at 45, 90, 135 degrees, and verifying that the pitch is different
2. Vibrato Control: The system shall increase the vibrato when the Y-axis rotation increases. Vibrato level 0 is from range [0, 15), level 1 is from range [15, 35), level 2 is from range [35, 60), and level 3 is from [60, 90].
    a. Confirmable by: rotating the hand 45 degrees and measuring the change in vibrato
3. Silencing Gesture: The system shall mute output when Z-axis rotation exceeds 25 degrees
    a. Confirmable by: rotating the hand past 30 degrees and checking audio output drops to 0.
4. Harmony Input: The system shall add a harmonic note when the thumb makes contact with another finger, with harmonic intervals defined as:
    a. Thumb-index: +4 semitones
    b. Thumb-middle: +7 semitones
    c. Thumb ring: +12 semitones
    d. Confirmable by: logging detected finger-pair contact and the added harmony note.

Mode Requirements

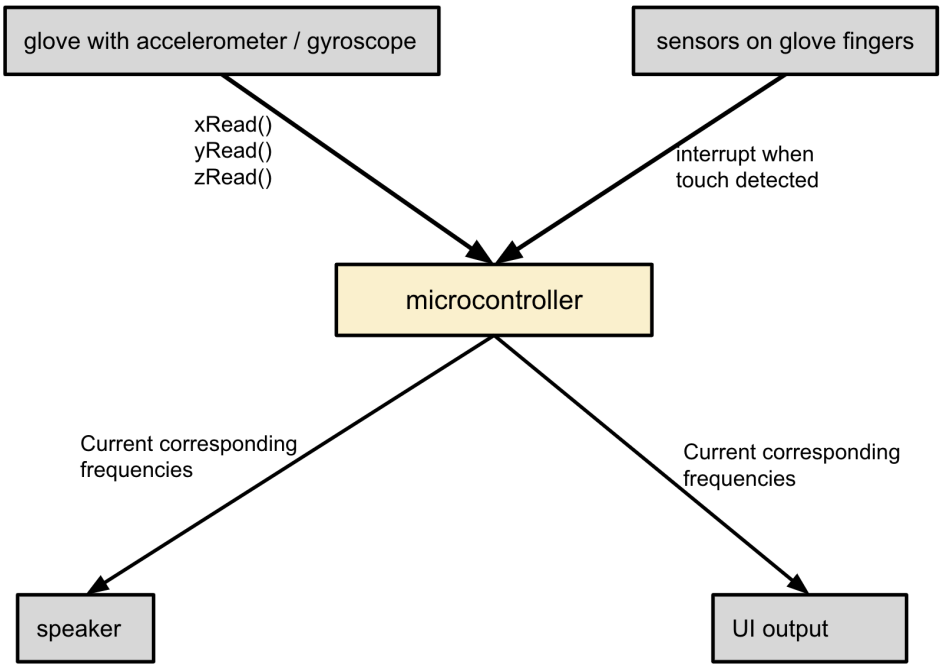
5. Mode Switching: The system shall switch between Standard Mode and Gesture Mode when the mode button on the UI is clicked. (Note: the formal name is gesture mode, but the UI prints statements that indicate the user is in drum mode, so that the functionality feels intuitive to the users. Therefore, gesture mode and drum mode are synonymous for our current use case.)
    a. Confirmable by: clicking the button and checking the change in UI and how sound output responds to gestures.
6. Gesture Mode gestures: In Gesture Mode, the system shall trigger six different drum sounds based on directional accelerometer motion exceeding x degrees:
    a. +X → snare
    b. -X → kick
    c. +Y → tom
    d. - Y → hi-hat
    e. +Z → ride
    f. - Z → cymbal
    g. Confirmable by: performing each motion and verifying that it triggers the designated sound.

UI Requirements

7. Chosen base note: The UI shall allow the user to choose the base note to be played from the notes available on the piano display.
    a. Confirmable by: checking that the note chosen is logged and the speaker outputs the correct note frequency.
8. Musical Score: The UI shall display the main note currently being played, showing both the note letter and its position on a musical staff. If harmonies are active, the UI shall also display their noteheads on the same staff.
    a. Confirmable by: checking that notes are being added to the on-screen musical score.
9. Recording and looping: The UI shall support starting a recording, ending a recording, replaying the recording, and downloading the recording as a JSON.
    a. Confirmable by recording a track, checking the same pattern plays, replaying the pattern, saving or discarding the recording using on-screen buttons, and ensuring that the recording downloads a JSON (if saved).

Our group does not have any should/optional requirements.

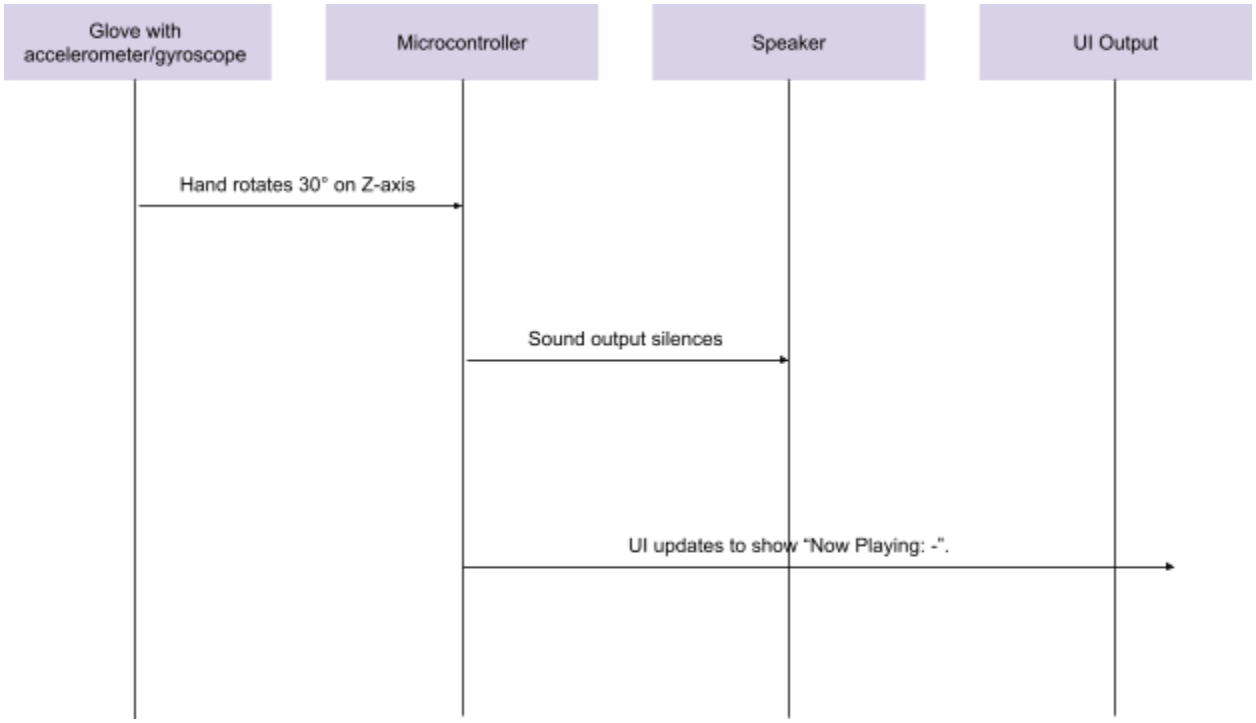
Revised architecture diagram from part 2 of the milestone report.



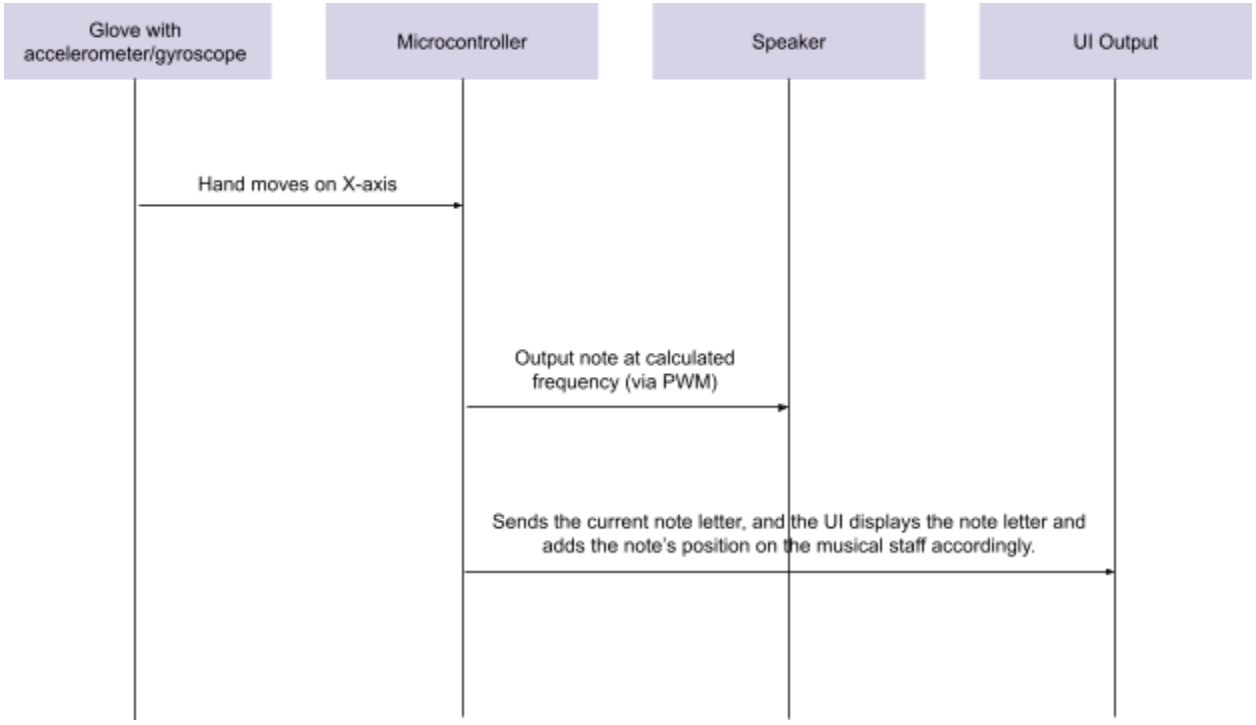
Three sequence diagrams for reasonable use case scenarios of your system (if there are more than 3, you should pick the 3 most likely use case scenarios to diagram, and include a note as to what scenarios you left out). Before *each diagram, you should write a short (1-2 sentence) description of what the use case scenario is.

Sequence Diagram 1: The user rotates their hand such that the glove's Z-axis angle exceeds the mute threshold (25°). The microcontroller interprets this gesture as a silence command, stops audio output

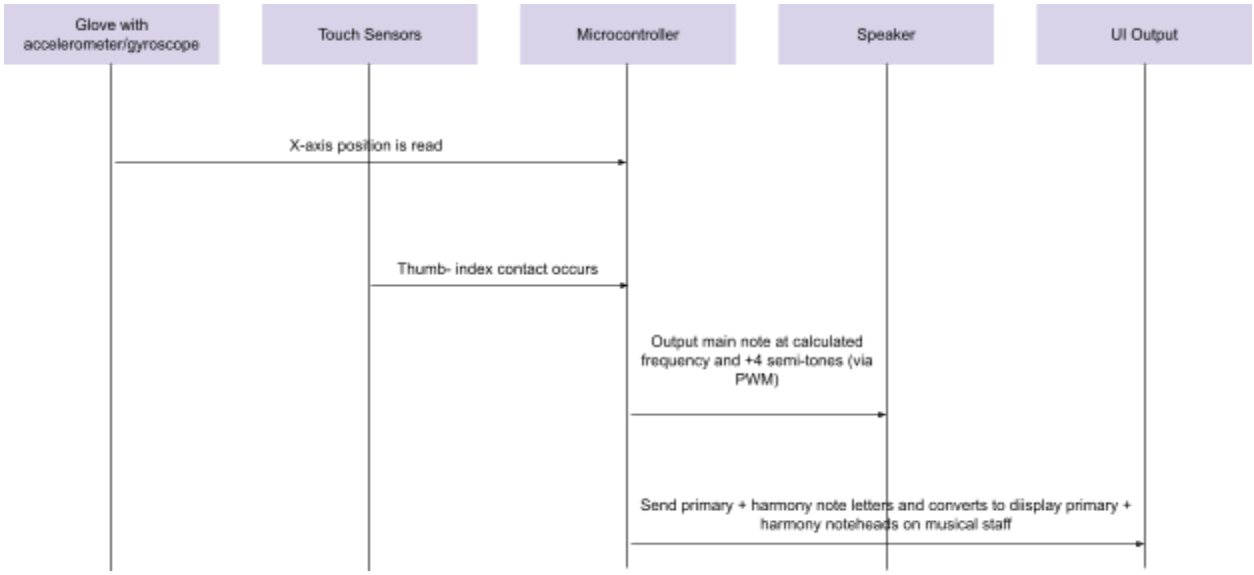
through the speaker, and updates the UI to reflect mute status.



Sequence Diagram 2: The user moves their hand up and down in standard mode, and the system interprets X-axis motion as pitch, outputs a corresponding note, and updates the UI display.



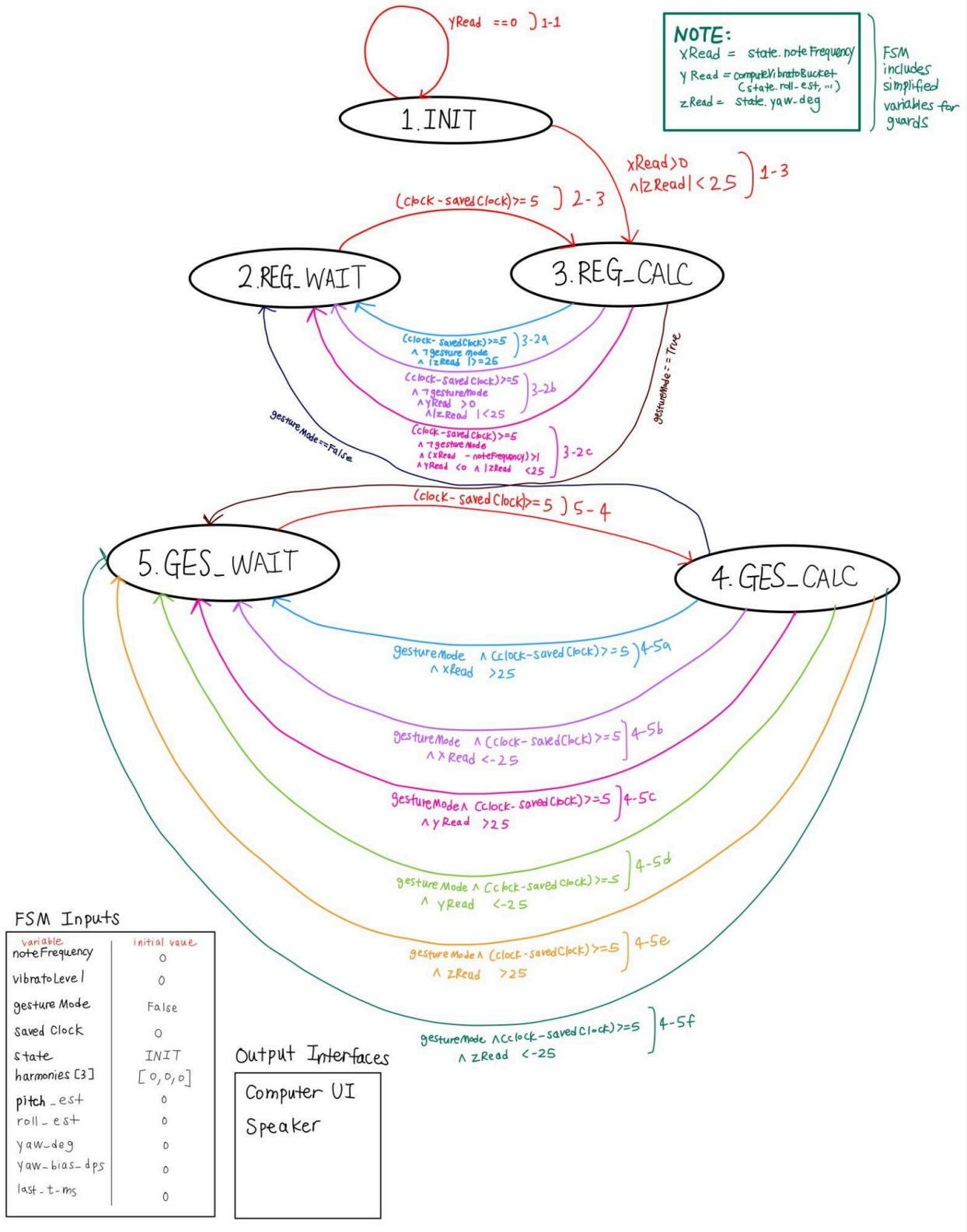
Sequence Diagram 3: The user touches their thumb to their index finger, triggering a harmony. The microcontroller detects the contact, generates the corresponding harmony along with the primary note, sends the audio to the speaker, and updates the UI with all active notes on the musical staff.



Scenarios we left out:

- Recording, replaying, and saving
- Vibrato changes via Y-axis rotation
- Gesture mode and subsequent gestures
- Triggering the base note
- Switching to gesture mode

***Revised finite and/or extended state machine from part 4 of the milestone report. See part 13 below for directions on including your peer review feedback as an appendix.**


yRead ==0) 1-1
1. INIT
NOTE:
xRead = state.noteFrequency
yRead = computeVibratoBucket(state.roll_est, ...)
zRead = state.yaw_deg
FSM includes simplified variables for guards
xRead>0 ∧|zRead|<25) 1-3
(clock - savedClock)>=5) 2-3
2. REG_WAIT
3. REG_CALC
(clock - savedClock)>=5 ∧ ¬gestureMode ∧ |zRead| >=25) 3-2a
(clock - savedClock)>=5 ∧ ¬gestureMode ∧ yRead >0 ∧ |zRead| <25) 3-2b
(clock - savedClock)>=5 ∧ ¬gestureMode ∧ (xRead - noteFrequency)>1 ∧ yRead <0 ∧ |zRead| <25) 3-2c
gestureMode == False
gestureMode == True
(clock - savedClock)>=5) 5-4
5. GES_WAIT
4. GES_CALC
gestureMode ∧ (clock - savedClock)>=5 ∧ xRead >25) 4-5a
gestureMode ∧ (clock - savedClock)>=5 ∧ xRead <-25) 4-5b
gestureMode ∧ (clock - savedClock)>=5 ∧ yRead >25) 4-5c
gestureMode ∧ (clock - savedClock)>=5 ∧ yRead <-25) 4-5d
gestureMode ∧ (clock - savedClock)>=5 ∧ zRead >25) 4-5e
gestureMode ∧ (clock - savedClock)>=5 ∧ zRead <-25) 4-5f
FSM Inputs
variable
initial value
noteFrequency 0
vibratoLevel 0
gestureMode False
savedClock 0
state INIT
harmonies[3] [0,0,0]
pitch_est 0
roll_est 0
yaw_deg 0
yaw_bias_dps 0
last_t_ms 0
Output Interfaces
Computer UI
Speaker

** Transitions 3-2(a), 3-2(b), and 3-2(c) go from REG_CALC to REG_WAIT, but different functionality gets triggered (stop(), vibrato(), and playNote() depending on priority conditions in order of xRead, yRead, and zRead. We decided to put separate conditions to distinguish the features.*

*** Similarly, transitions 4-5(a), 4-5(b), 4-5(c), 4-5(d), 4-5(e), and 4-5(f) go from GES_CALC to GES_WAIT, but likewise different functionality gets triggered (snare(), kick(), tom(), hat(), ride(), cymbal()) depending on xRead, yRead, and zRead. We decided to put separate conditions to distinguish the features.*

Transition	**Guard**	**Output**	**FSM variables**
1-1	yRead == 0		
1-3	xRead > 0 ∧ |zRead| < 25	-	noteFrequency := msg.frequency
2-3	(clock - savedClock) >= 5		clock = savedClock
3-2 (a) *	(clock - savedClock) >= 5 ∧ ¬gestureMode ∧ |zRead| >= 25	stop()	noteFrequency := 0
3-2 (b) *	(clock - savedClock) >= 5 ∧ ¬gestureMode ∧ yRead > 0 ∧ |zRead| < 25	vibrato()	vibLevel := computeVibratoBuckets(yRead())
3-2 (c) *	(clock - savedClock) >= 5 ∧ ¬gestureMode ∧ (xRead - noteFrequency) > 1 ∧ yRead < 0 ∧ |zRead| < 25	playNote()	noteFrequency := xRead()
3-5	gestureMode == true	Serial.Println(“You are in Drum Mode!)	gestureModeOn := 1
4-2	gestureMode == false	Serial.Println(“You are in Regular Mode!)	gestureModeOn := 0
4-5 (a) **	gestureMode ∧ (clock - savedClock) >= 5 ∧ Xread > 25	snare()	
4-5 (b) **	gestureMode ∧ (clock - savedClock) >= 5 ^ Xread < -25	kick()	
4-5 (c) **	gestureMode ∧(clock - savedClock) >= 5 ∧ Yread > 25	tom()	
4-5 (d) **	gestureMode ∧ (clock - savedClock) >= 5 ∧ Yread < -25	hat()	
4-5 (e) **	gestureMode ∧ (clock - savedClock) >= 5 ∧ Zread > 25	ride()	

4-5 (f) **	gestureMode ∧ (clock - savedClock) >= 5 ∧ Zread < -25	cymbal()	
5-4	(clock - savedClock) >= 5		clock = savedClock

A traceability matrix of how the transitions and states of the FSM in part 5 trace to the requirements in part 2. If your system is made up of multiple modules, you will not have full traceability from just one FSM, so you are not required to have a check in every column (requirement) of the matrix. Traceability matrices were covered in the FSM lectures.

	R1	**R2**	**R3**	**R4**	**R5**	**R6**	**R7**	**R8**	**R9**
1-1							x		
1-3							x	x	
2-3	x	x	x	x					x
3-2(a)			x	x				x	x
3-2(b)		x						x	x
3-2(c)	x			x				x	x
3-5					x			x	
4-2					x			x	
4-5 (a)						x			
4-5 (b)						x			
4-5 (c)						x			
4-5 (d)						x			
4-5 (e)						x			
4-5 (f)						x			
5-4						x			

***An overview of your testing approach, including which modules you unit tested and which integration and system (software and user-facing/acceptance) tests you ran. You should explain how you determined how much testing was enough (i.e. what coverage or other criteria you used, how**

you measured that you achieved that criteria, and why you think it is sufficient for you to have met that criteria).

We unit tested each of our FSM transitions, and with each transition, we tested all the possible inputs. This approach allowed us to test all the possible state changes in isolation. Specifically, we have 16 unit tests that test that the chosen input note, X, Y, and Z inputs from our accelerometer/gyroscope, the gesture mode button, and our loop accurately causes the correct transition to occur. We tested this by mocking inputs that would trigger each transition, and were guided by all possible transitions that could be taken. We also implemented integration testing for each of our sequence diagrams and a few other cases. These tests verify that when certain actions are taken, both the expected state is reached and the UI visually updates as expected. We created a mock updateFSM function that takes in mock inputs and triggers state changes based on our FSM and visual outputs in the UI. We were guided by our sequence diagrams and the different actions you can take on the UI. Additionally, there are a few test transitions in the file with the unit transitions that ensure multiple transitions in a row can take place (ex, 3-4-5 when pressing the gesture mode button and then inputting a gesture to cause one of the drum sounds).

Unit tests in FSMtesting.ino - all modules are tested (accelerometer, touch sensors, MCU, speaker, UI).

- Transition 1-1: stays in s_INIT until there is an input note frequency
- Transition 1-2: input note frequency value becomes frequency value
- Transition 2-3: Test that after 5ms, state changes
- Transition 3-2: Test that Y rotation of accelerometer causes a change in vibrato
- Transition 3-2: Test that X rotation of accelerometer causes a change in note
- Transition 3-2: Test that Z rotation of accelerometer is causes the speaker to go silent if more than 25 degrees
- Transition 3-2: Test that Z rotation of accelerometer does not cause any change in sound if less than 25 degrees
- Transition 3-5: Test that button press in normal mode changes to gesture mode
- Transition 4-2: Test that button press in gesture mode changes to normal mode
- Transition 5-4: Test that after 5 ms, state changes to gesture wait mode
- Transition 4-5: Test that X rotation of accelerometer (up) causes snare sound
- Transition 4-5: Test that X rotation of accelerometer (down) causes kickdrum sound
- Transition 4-5: Test that Y rotation of accelerometer (left) causes tom sound
- Transition 4-5: Test that Y rotation of accelerometer (right) causes hi-hat sound
- Transition 4-5: Test that Z rotation of accelerometer (left) causes ride sound
- Transition 4-5: Test that Z rotation of accelerometer (right) causes cymbal sound

Integration tests in FSMtesting.ino

- Transition 3-5-4-5: test that when you press the button in regular mode, you switch to gesture mode and then can make a gesture
- Transition 4-5-4: test that it loops between wait and calc in gesture mode
- Transition 2-3-2: tests that it loops between wait and calc in regular mode

Integration testing that we tested physically:

- Test that, in regular mode, X rotation up causes a change in note (increase in pitch)
- Test that, in regular mode, X rotation down causes a change in note (decrease in pitch)
- Test that, in regular mode, Y rotation of accelerometer causes a change in vibrato
- Test that, in regular mode, Z rotation of accelerometer causes speaker to go silent
-
- Test that, in gesture mode, X rotation of accelerometer (up) causes snare sound
- Test that, in gesture mode, X rotation of accelerometer (down) causes kickdrum sound
- Test that, in gesture mode, Y rotation of accelerometer (left) causes tom sound
- Test that, in gesture mode, Y rotation of accelerometer (right) causes hi-hat sound
- Test that, in gesture mode, Z rotation of accelerometer (left) causes ride sound
- Test that, in gesture mode, Z rotation of accelerometer (right causes cymbal sound
-
- Test record, loop, replay: record note, loop, replay it
-
- Thumb/index finger in normal mode plays 3 note up harmony
- Thumb/middle dinger in normal mode plays 5 note up harmony
- Thumb/ring finger in normal mode plays 8 note up harmony

UI integration tests in the integrationTests.spec.js file

- Test that in the initial state, no input shows no note
- Test that in the initial state, a valid input triggers the 1-2 transition
- Test that a change in note is reflected visually in the UI
- Test that a change in stop is reflected visually in the UI
- Test that a change in harmonies is reflected visually in the UI
- Test that the button press for mode change also triggers the 3-5 and 5-3 transition
- Test that the button press for recording reflects visual changes and receives the recorded notes

How did we determine how much testing was enough (i.e. what coverage or other criteria you used, how you measured that you achieved that criteria, and why you think it is sufficient for you to have met that criteria).

Our updateFSM function, which determines which state to transition to and what musical gestures to trigger, is compatible with using the MC/DC coverage. To elaborate, we have different cases for each state, and within those states, we have conditions for what output to trigger. Based on this, we made sure to have a unit that could go into every state and every condition by giving it inputs that would trigger that output.

We think this is sufficient because it covers all aspects of our FSM.

The integration tests we chose covered the most common use cases with our synthesizer, which also triggered a sound output that we could then verify. We did this so we could test multiple aspects of our project working together.

For testing our front end, there is a test for every interaction you can take with our UI, both in isolation and when interacting with each other. We believe this is sufficient coverage given that it uses all of the functionality in the system.

At least 2 safety and 3 liveness requirements, written in propositional or linear temporal logic, for your FSM. Include a description in prose of what the requirement represents. Each requirement should be a single logic statement, made up of propositional and/or linear temporal logic operators, that references only the inputs, outputs, variables, and states defined in part 5 of this report. We will discuss safety and liveness requirements at the end of November. You can also refer to chapter 13 of Lee/Seshia or chapters 3 and 9 of Alur's textbook (Brown login required).

Safety Requirement 1: When in REG_WAIT state and the Z-axis reading exceeds 15 degrees, the speaker must stop immediately. This makes sure there is no sound output when the hand is moved beyond the mute threshold.
G ((state = REG_WAIT $\wedge$ zRead() >= 15) $\rightarrow$ stop())

Safety Requirement 2: The drumMode variable must never indicate both modes simultaneously. If gestureMode = true, the system is in Gesture Mode, and if gestureMode = false, it is in Regular Mode.
G $\neg$(gestureMode = true $\wedge$ gestureMode = false)

Liveness Requirement 1: When in REG_WAIT and the X-axis reading differs from the current noteFrequency, the system must play the new note in the next state REG_CALC.
G (state = REG_WAIT $\wedge$ xRead() != noteFrequency $\rightarrow$ X (playNote()))

Liveness Requirement 2: When in REG_WAIT state and the Y-axis is positive, the system should apply a vibrato effect in the next state REG_CALC corresponding to the appropriate vibrato bucket and update lastVibLevel to match the selected rate.
G (state = REG_WAIT $\wedge$ yRead() > 0 $\rightarrow$ X (vibrato() $\wedge$ lastVibLevel $\in$ {0,1,2,3}))

Liveness Requirement 3: In GESTURE_WAIT state, if a drum gesture (e.g. x/y/z axis beyond threshold) occurs, the corresponding drum output must be played in the next state GESTURE_CALC.
Example for kick: G (state = GESTURE_WAIT $\wedge$ Xread() < -25 $\rightarrow$ X kick())

A discussion of what environment process(es) you would have to model so that you could compose your FSM with the models of the environment to create a closed system for analysis. Composing state machines was covered during the modeling lectures. For each environmental process, you should explain and justify whether it makes sense to model it as either a discrete or hybrid system and either a deterministic or non-deterministic system (for example, a button push could be modeled as a discrete, non-deterministic signal because it is an event that could happen at an arbitrary time, but a component's position could be modeled as a hybrid, deterministic signal based on some acceleration command). You are not being asked to actually do this modeling or any sort of reachability analysis, just to reason about this sort of modeling at a high level.

To analyse our FSM as a closed system, it would need to be composed with models of each environment process that provide inputs and feedback. These are the components that we have identified would directly influence transitions and outputs:

Human Hand Motion (Accelerometer/Gyroscope inputs): hand motions generate xRead, yRead, and zRead values, which trigger state transitions (pitch change, vibrato, silencing)

- Model type: Hybrid, non-deterministic.
    - Hybrid: Hand movement is a continuous motion, but sampled discretely by the MCU
    - Non-deterministic: motion timing is user-driven and unpredictable

Touch sensor (Harmony Detection): touching the thumb to fingers initiates harmonies inside regular mode (states 2 and 3)

- Model type: Discrete, non-deterministic
    - Discrete: contacts are events (either touch or no touch), not continuous values
    - Non-deterministic: user may press inconsistently/unpredictably

Mode Toggle Button UI (standard/gesture modes): Button press causes state change between reg_wait and ges_wait.

- Model type: discrete, non-deterministic
    - Discrete: button press is an instantaneous event signal
    - Non-deterministic: press timing is user-driven and can't be predicted by the system

First Note Piano Key UI: Button press on the piano causes a state change between INIT and REG_CALC.

- Model type: discrete, non-deterministic
    - Discrete: button press is an instantaneous event signal
    - Non-deterministic: press timing is user-driven and can't be predicted by the system

Timer clock (sampling period): every major transition (REG_WAIT↔REG_CALC and GES_WAIT↔GES_CALC) uses clock - savedClock ≥ 5 ms to progress.

- Model type: discrete, deterministic
    - Discrete: Although time is physically continuous, the MCU timer is discrete.
    - Deterministic: loop runs every 5ms

A description of the files you turned in for the code deliverable. Each file should have a high-level (1-3 sentence summary) description. For each of the assignment requirements (PWM, interrupts, serial, etc), you should indicate which file(s) and line(s) of code fulfill that requirement.

- Files
    - AccelerometerLibrary.ino:
        - This file continuously reads in data from the IMU using I2C and converts it to our sound control signals: hand movement on the x-axis determines pitch, on y-axis determines vibrato, and on the z-axis silences it. It filters these readings to get stable angles, then feeds those values into our FSM that decides what the

sound output should be (start, stop, gesture mode) and updates the sound engine accordingly.

- baseFinalProject.ino
    - This is our main control module and initializes the IMU, sound engine, harmony, sensors, and timer system. It reads the serial keyboard input to control (base note, gesture mode, recording), which is how we have connected the UI to our Arduino logic, and was also used to test the Arduino before integration. It also updates harmony controls, drum controls, and polls the IMU every loop cycle. It also has the set up and use for the ADC button to toggle modes.
- DrumLibrary.h
    - This header file declares the structs needed for gesture mode, mainly the envelope needed to produce the different sounds and the current state of the drum. It also provides the constants and function declarations needed.
- DrumLibrary.ino
    - This file implements our drum engine. Each drum trigger sets a frequency and then shapes the amplitude over time using an envelope to control PWM duty cycle. The gestures are: snare, kick, tom, hi-hat, ride, cymbal.
- fsmTesting.ino
    - This file includes all of the unit tests for each transition in isolation and a few tests that simulate transitioning between multiple states. We used the same method as Lab 6, filling out a testing table and generating 16 unit tests. These confirm that states change accordingly based on inputs and our FSM. For integration tests, we have one going through states 3-4-5, 5-4-3, and 3-2-3.
- harmonyControls.ino
    - This file uses three capacitive touch sensors and updates three harmony flags based on thresholded sensor readings. When a harmony flag changes, it toggles the corresponding output pin and plays the harmony. It also prints harmony notes to serial so that they can be translated over to the UI.
- IMUHelpers.h
    - This header file declares all of the functions needed to properly transform the accelerometer/gyroscope readings into values we can use to play notes, silence, and apply vibrato.
- IMUHelpers.ino
    - This file defines each of the functions declared in IMUHelpers.ino. Notably, the functions help process and convert raw IMU data into orientations (pitch, yaw, and roll) and musical outputs (note frequency and vibrato).
- Notes.h:
    - This header file declares the note name array and the frequency-to-MIDI conversion function so other code modules can use them.
- Notes.ino:
    - This file converts a frequency in Hz to its corresponding MIDI note number and provides a lookup table for note names.
- SoundEngine.h

        - This header file defines the FSM state struct and declares functions for recording, playback, and harmony control.
    - SoundEngine.ino:
        - This file implements the core sound engine for our project, including live note generation, harmonies, vibrato, and gesture modes, using the MCU's GPT timers and pins. It also helps handle recording and playback. This code uses interrupts to produce precise ADC output for audio in real time.
    - WDT.ino:
        - This file holds the setup and associated functions with the Watchdog timer. The code is based on that of Lab 4 and modified for the use of the project.
    - UI.html:
        - This file implements the UI for our project that connects to an Arduino via the Web Serial API. It renders controls (connect, drum, record, replay, save, discard), a clickable piano keyboard that starts the basenote, a visual score with a "Now Playing" display, and a log area. The embedded JavaScript manages the serial connection and message parsing (notes, harmonies, recording chunks), updates the score and UI state, and handles recording buffering and saving as JSON.
    - Style.css:
        - This file defines the visual layout, styling, and interactive states for the web UI, including the three-column layout, piano/key and score visuals, and buttons.
    - integrationTests.spec.js
        - This file uses Playwright for integration/end-to-end tests. It implements a mockUpdateFSM function to simulate the updateFSM function from AccelerometerLibrary.ino. This sends the same Serial communication messages as updateFSM based on certain inputs, such that the UI changes appropriately. The tests then check that the changes in the UI are reflected and the state is updated accordingly based on the transition(s) taken.

- Requirements
    - PWM:
        - We use PWM to output the note frequency on our piezo speaker, similar to how we did in Lab 4. This is done in playNote() (lines 78-126), stopPlay() (lines 129-134), and noteISR() (209-223) in SoundEngine.ino.
        - In drumLibrary.ino, specifically around line 66, the duty cycle is set so that PWM can be used. PWM is needed to create the envelope sound shape needed for the drum sounds.
    - ADC:
        - In AccelerometerLibrary.ino, specifically the function mpuReadRaw() from lines 130-150, we read in the raw analog values from our accelerometer and gyroscope and convert them to digital signals we can use for our gestures. We do the processing through the functions in IMUHelpers.ino (all lines).
        - We use the ADC to read the analog voltage from a button. The ADC converts this voltage into a digital value, which we threshold in software to toggle between

regular mode and drum mode. This allows the system to operate independently of the UI (lines 12 - 32 and 67).

- Watchdog:
    - The watchdog is pet in every transition of our FSM. The watchdog logic is in WDT.ino, and the watchdog is specifically set on lines 205, 233, 254, 275, 291, 301, 327, 351, 366, 383 of AccelerometerLibrary.ino.
- Timer/counter:
    - Our project uses a timer to count up to 5ms to transition between our CALC and WAIT modes.
    - We also use a timer to configure note frequency and duration. Similar to lab 4, based on a note frequency, we configure a timer to count up to the corresponding value, and trigger the interrupt once it overflows.
    - We also use a timer in gesture mode to shape the envelope, specifically used to determine the length for how long a specific frequency is played.
    - We also use a timer for vibrato. We map the function so it oscillates, it goes from zero to a number, then negative, then up and down. This oscillation is what is used to make the vibrato sound.
    - You can reference this in SoundEngine.ino, specifically 137-185.
- Interrupts:
    - Similar to lab 4, we use interrupts to help with playing notes at a frequency and duration. This can be found in noteISR in SoundEngine.ino (lines 209-223) and in gptISR() (lines 136 - 142).
    - Harmonies also use interrupts, such that when you make contact with the capacitive sensor, the logic for playing harmonies will execute. This can be found in gptISRHarmony(), gptISRHarmony2(), and gptISRHarmony3() (lines 145-158 in SoundEngine.ino).
    - We also use interrupts for drum mode, similarly to regular play. We see the interrupt in playbackISR (lines 299-308 in SoundEngine.ino)
- Serial communication
    - ReadFromArduino(), handleArduinoMessage() in UI.html, and our basefinalproject.ino file are all integrated to allow for communication between our Arduino Serial output and our front-end interface. Specifically, you can see line 91-9 of UI.html, the Serial port is being requested and written into our UI.

A procedure on how to run your unit tests (for example, if you have mock functions that are used by setting a macro, similar to lab 6, make sure to note that).

To run the unit tests, call testAll() and then while(true) in the setup function in baseFinalProject.ino. Also be sure to #define TESTING in SoundEngine.h. To run the end-to-end/integration tests, run “npx playwright test”.

A reflection on whether your goals were met and what challenges you encountered to meet these goals. You should only include challenges met or newly addressed since the milestone.

All of our goals were met.

Challenges:

1. We sometimes have problems with z-axis drift. For example, if you move to stop, you might need to move further back in the original direction to begin playing again. We believe this might be because gravity and slight sensor misalignment make the z-axis especially prone to apparent movement. This issue is negligible.
2. For the UI, we send messages back to the Arduino with a writer object; however, because our project has multiple components that can happen concurrently, the writer effectively has a queue. This means if there are multiple messages needing to be sent, they will be handled in order. This caused small issues with trying to switch between drum and regular mode very quickly using the UI, since it can't process those messages instantly to the Arduino, which then also needs to send information back. However, this isn't an issue if you aren't trying to switch modes rapidly.
3. Due to the overall structure of the glove requiring lots of movements, in particular in gesture mode, in which quick changes in angles must occur to trigger a sound, we encountered wire errors in which wires would get loose and have unexpected behaviors. This would sometimes result in the synthesizer getting stuck in one mode and printing "I2C endTransmission err." However, this was not a major functionality issue and required precaution to prevent it.

As an appendix, include the review spreadsheets you received after the milestone demo. Each defect should be marked as "fixed" or "will not fix" with a justification.

Feedback from milestone demo and report for our FSM:

Transition/State	**Defect**	**Fixed/Not Fixed**
	Gradescope: Defines inputs	Fixed
	Gradescope: Defines outputs/output interfaces	Fixed
	Gradescope: Defines ESM variables	Fixed
2-3 5-4	Milestone spreadsheet: Does check watchdog count as an output? Not included among output variables. Perhaps this should count as a function rather than an output. Unsure if pet_watchdog() is something that belongs on the FSM? Pet watchdog doesn't belong in the FSM Pet watchdog doesn't belong in the FSM I do not think watchdog should be in FSM Same watchdog error.	Fixed
3-2 (b and c)	Milestone spreadsheet:	Fixed

	Transitions are not mutually exclusive; both vibrato and note frequency can be updated at the same time	
3-2a/b/c	Milestone spreadsheet: Since the guards each only check for one of zRead, xRead, and yRead, these guards could hypothetically all be true at once, then the FSM wouldn't know which state to transition to Each guard inspects only one of zRead, xRead, and yRead. All three conditions can therefore hold at the same time, which leaves the FSM without a clear choice of next state.	Fixed
	Milestone spreadsheet: All transitions out of a state are mutually exclusive	Fixed
	Milestone spreadsheet: Only outputs and variables are set in actions	Fixed
	Milestone spreadsheet: FSM behavior aligns with the behavior/requirements put forth in the team presentation	Fixed

Other - comments from Gradescope

Section	**Comment**	**Fixed/Not Fixed**
Requirements	Why are all of these “should” requirements? Under what circumstances would these requirements not be met?	fixed
Requirements	These should be confirmable! What defines the behavior?	fixed
Architecture diagram	Does the MCU send signals to the glove?	Not fixed - not, it does not send signals to the glove
Test Plan	Add more on if mocks will be necessary, what artifacts will guide testing.	fixed