Week 11:

Date: 11/4/2022

Hours: 12

Description of design efforts:

Microphone Array Trilateration:

I continued to devise the hardware for this system, stripping ribbon cables for signal transmission from the microphones to the microcontroller. As each sensor has a ground pin and an output pin, we need 2 lines per sensor, or 8 lines per side (16 lines total). For now, as there are only 4 microphones wired to the table for feasibility testing, we will only be wiring with 8 lines, as depicted by the blue wires in Figure 1. As this image is marked with 1 unit as 3.5 cm, we can calculate the estimated length needed for each wire from any given sensor to the microcontroller, which will be loosely placed next to the central microphone on one side of the table.

One half of the table is a perfect square, measuring 77 cm on each side. With each microphone positioned approximately 5 cm in from each corner, Figure 1 shows almost a perfect model of the table. Each “straight” wire between sensors should then be 67 cm; I’ve increased this to 80 cm, as a safety net, if some needs to be removed. Then, from the central sensor, I added 10 cm to reach the microcontroller.

In the bottom right of Figure 1, there is a purple figure which shows the ribbon cable split. We know that some sensors are closer to the microcontroller than others, and thus need longer wires. If the wires are cut following the purple figure, where 1 unit is 10 cm, we should fit the wires accordingly. To set this up for only 1 side of the table, we simply use one half of the purple image.

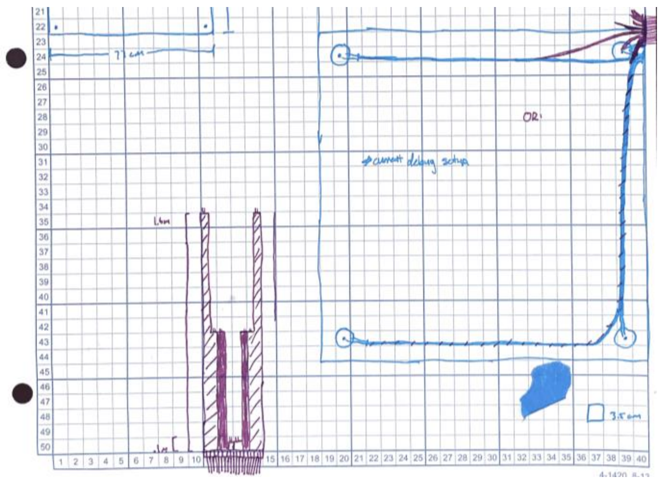


Figure 1. Wiring layout.

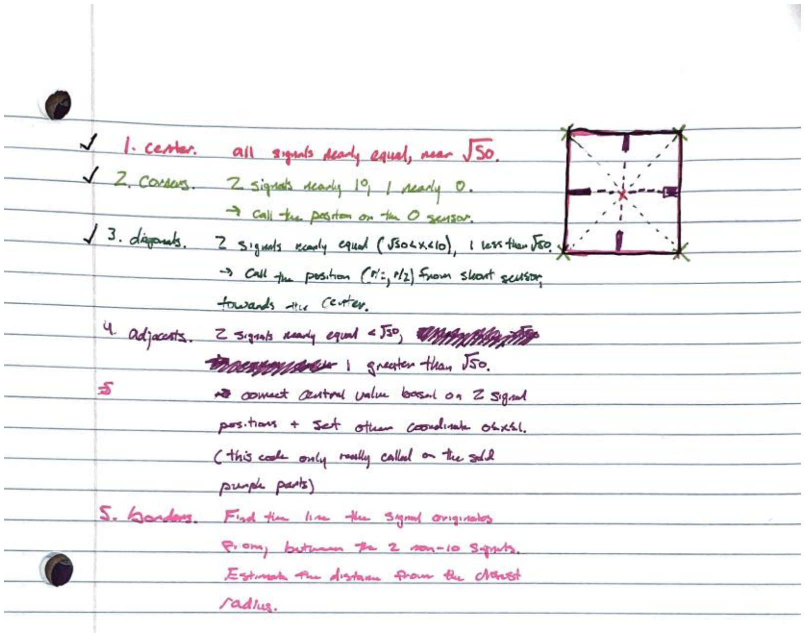
I began prototyping the logic in a Python script, where I input the coordinates of the bounce, then estimate where it was. To do so, we first convert the real coordinates into circles from each sensor with a radius equivalent to the distance to the bounce. Then, we simulate the "delay", where we know that the closest sensor's delay will be 0 and subtract this from each signal. Now, we have the true data that would have been read by the microcontroller. Of course, on the microcontroller, we will need to convert the number of samples before the signal is collected into a numerical value, representing the delay in milliseconds. The data is then normalized, to 10 in the simulation, but preferably to 72 cm on the table, as this is the maximal true distance from the sensor to the bounce.

Next, we must process the circles to determine the location. We do so first through an algorithm to determine the intersection points of each combination of "useful" circles. That is, the 3 nearest circles to the bounce. If 2 circles are equally furthest away, we remove either. Here, there are 6 cases that require different processing. Each of them are depicted in Figure 2, and further explained below:

> *Note that "signals" below refer, in general, to the value of the delay seen at a particular sensor. While the true values will be in ms, we can generalize them to approximately be the true distance in cm. Of course, this will require an algorithm to convert the two later.

0. A "standard bounce". This will make more sense as each other cases are explained, but in essence, this is the base case. Any bounce that does not fall into other cases is calculated here, with the algorithm I explained in Progress Report 9.
1. A center bounce (red). Here, we find all signals nearly equal (as we may find error in a non-simulated environment), at around sqrt(2*[72/2]^2), or the distance from the sensor to the center of the table (on the given side). Here, we can estimate the location to be (77/2, 77/2).
2. A "corners" bounce (light green). Here, a bounce has occurred on a corner of the table. We will see 1 signal nearly 0, and 2 nearly equivalent at around 77. Estimate the location to be that of the sensor with minimal reception (this is actually a subset of case 3).
3. A "diagonals" bounce (green). Here, a bounce has occurred on a diagonal of the table. As such, 2 signals will be nearly equal, between sqrt(2*[72/2]^2) and 77, and 1 will be something between 0 and sqrt(2*[72/2]^2). Set the location to be (r/2, r/2) towards the center, from the sensor with smallest signal (whose value is r).
4. An "adjacents" bounce (purple). This bounce occurs on the dividing gridlines that split the table into quadrants. The logic is similar to case 3: we know 2 signals are nearly equal and less than sqrt(2*[72/2]^2), and 1 signal is greater than sqrt(2*[72/2]^2). In practice, this case only occurs near the edges of the table, thus, we can estimate the "table-side" coordinate to be ~5cm or ~72cm, and the other coordinate to be 77/2, choosing the correct combination of these values based on which 2 sensors have the small signals.

5. A “borders” bounce (pink). This occurs when a ball bounces near the edge of the table on any side. We should see a signal with a maximal, or normalized radius, 77, and 2 signals that are not. Those 2 sensors can be used to pick the edge that the bounce has occurred on, and we can estimate the distance from the shortest sensor using its radius and determine that the position is r points towards the second sensor from the first.



✓ 1. center. all signals nearly equal, near $\sqrt{50}$.

✓ 2. corners. 2 signals nearly 10, 1 nearly 0.
→ call the position on the 0 sensor.

✓ 3. diagonals. 2 signals nearly equal ($\sqrt{50} < x < 10$), 1 less than $\sqrt{50}$.
→ Call the position $(r/2, r/2)$ from short sensor, towards the center.

4. adjacents. 2 signals nearly equal $< \sqrt{50}$, 1 greater than $\sqrt{50}$.
5 → correct central value based on 2 signal positions + set other coordinate $0 \leq x \leq 1$.
(this code only really called on the solid purple parts)

5. borders. Find the line the signal originates from, between the 2 non-10 signals. Estimate the distance from the closest radius.

Figure 2. Bounce cases.

I have some concerns about implementing this. Most possible bounce positions have low error in the simulator, but not all. I have some bug fixes left to make in the algorithm. I have already made many and am running out of ideas on what may be going wrong. As of now, it appears that my intersection algorithm may be erroneous, as “mirrored” x and y coordinates sometimes result in different (mirrored) results, which is unexpected. All that is okay – if we are fairly accurate, that is good enough. My main concern, though, is how quickly we can sample the ADC. I increased the sample rate for some positioning testing and was impressed with the results. But when I later tested our newly constructed button matrix PCB (which was successful!), I noticed that we weren’t reading properly, or sending the data to the laptop properly. It appeared that the microcontroller was spending too much time in the ADC interrupts to work on others. I need to test changing interrupt priorities to hopefully resolve this. If it doesn’t, this idea may be in trouble.

Next Week:

Next week, I intend to pursue completion of the following tasks:

1. Test ADC sample rate limits for proper user interaction.
2. Continue debugging trilateration algorithm.
3. Continue wiring the trilateration setup.