Ping Pong Game Documentation

Running the code

There are three main programs that can be run: the realtime game, the signal capture demo, and the video playback scoreboard. Note that for the first two you need two microphones and an audio interface that can capture two channels at once. No packages outside of those used in class were used so requirements should be met already.

To run the main game (`pingpong_game/__main__.py`), make sure the audio interface with two channels is the default input and from the code repo (or wherever pingpong_game is stored) run:

```
python –m pingpong_game
```

To run the signal capture demo, make sure the audio interface with two channels is the default input and run:

```
python –m pingpong_game.signal_capture_demo
```

Both of these programs will produce a wave file output which can be used for post-processing and analysis.

To run the video playback version, make sure you have set the fields indicating the audio and video files in `pingpong_game/config.py` and then run:

```
python –m pingpong_game.devtools.play_video
```

Sample files have been supplied with the project software.

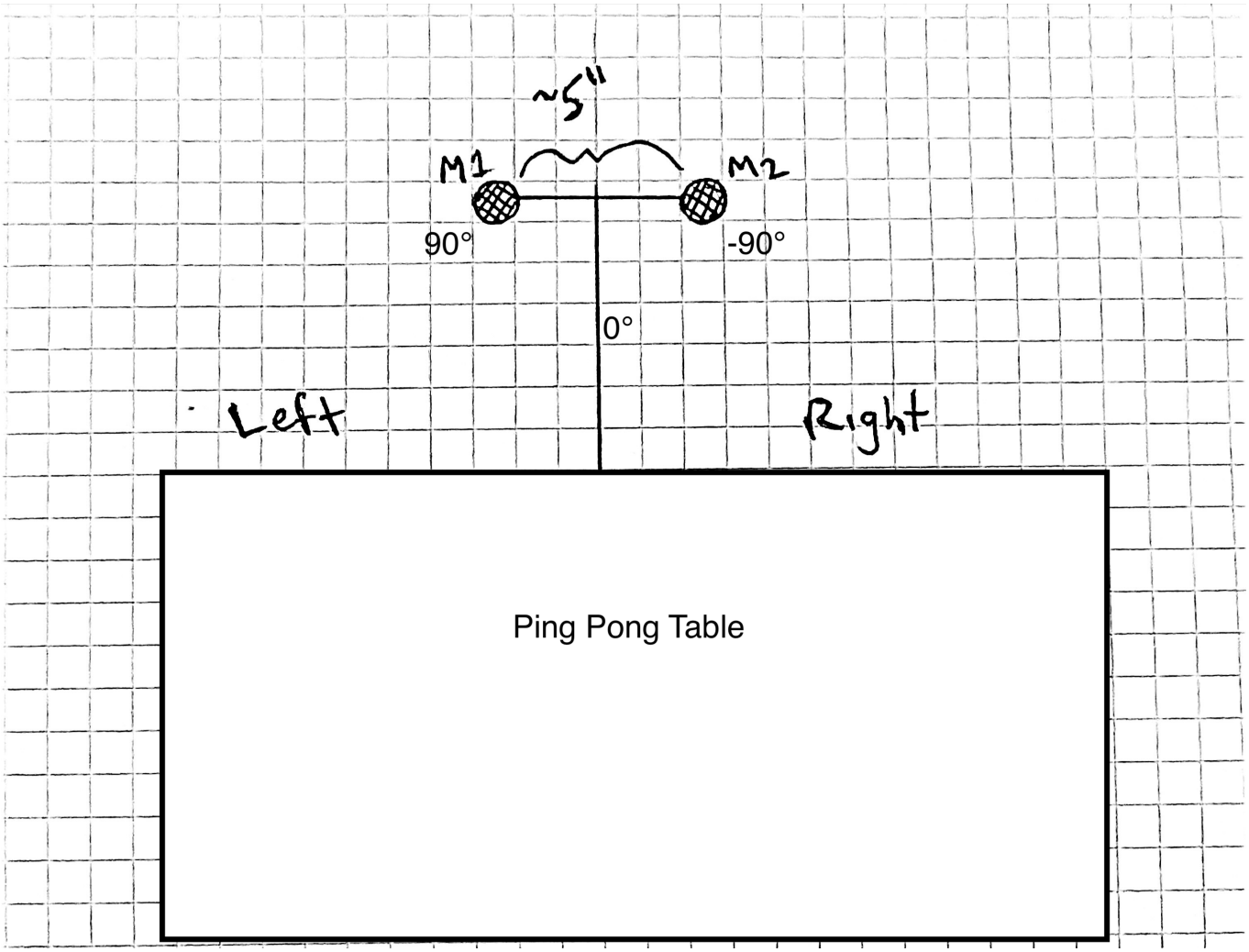
Note that for this last program the results may vary as a side effect of the audio-video synchronization, see the notes in `pingpong_game/devtools/play_video.py` for more details.

In addition to these main programs, you might also want to run just the signal capture code on an audio file to see how it performs. This can be achieved by updating the input file at the bottom of `pingpong_game/sig/signal_capture.py` and then running:

```
python –m pingpong_game.sig.signal_capture
```

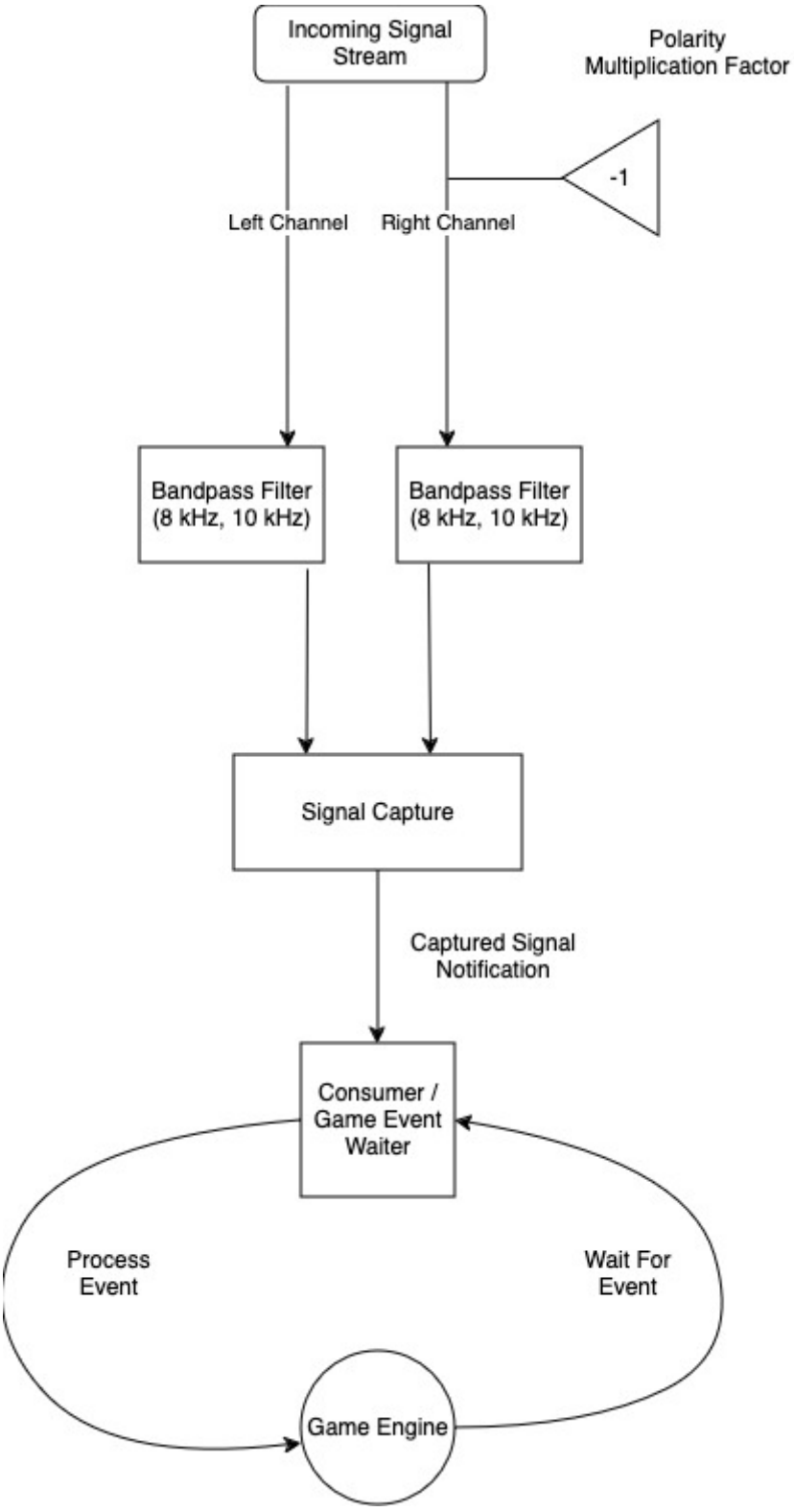
This will by default output the first 15 captures detected in the file, but the code can be taken and made into a full script if desired.

For use with a ping-pong set up and two microphones, the mics should be set up according to the following diagram (very much not to scale) with the microphones about 1m from the table:



System Overview

The following diagram provides an overview of the main components of the pingpong game system.



The incoming signal is split out into the two channels, one channel is multiplied by the polarity factor to account for possible sign differences between the two microphone's signals. In the

case of my set up this is a value of -1 [1].

After multiplying the right channel by this factor, the signal is filtered by a band-pass filter. Brief experimentation indicated that a good passband is 8-10 kHz. In all of the programs these steps occur in the audio processing thread.

The filtered signal is then passed to the signal capture object. This object listens to the incoming signal and picks out segments that have high power, since this occurs after the filtering is done these are almost exclusively ping-pong ball sounds.

When a new sound is detected, a notification is sent to the game loop which then determines if it was a valid sound. If the sound is valid (currently that just means it has high enough power), the angle of arrival is estimated and therefore which side of the table it originated from. This information is passed into a state machine which processes it as an in-game event.

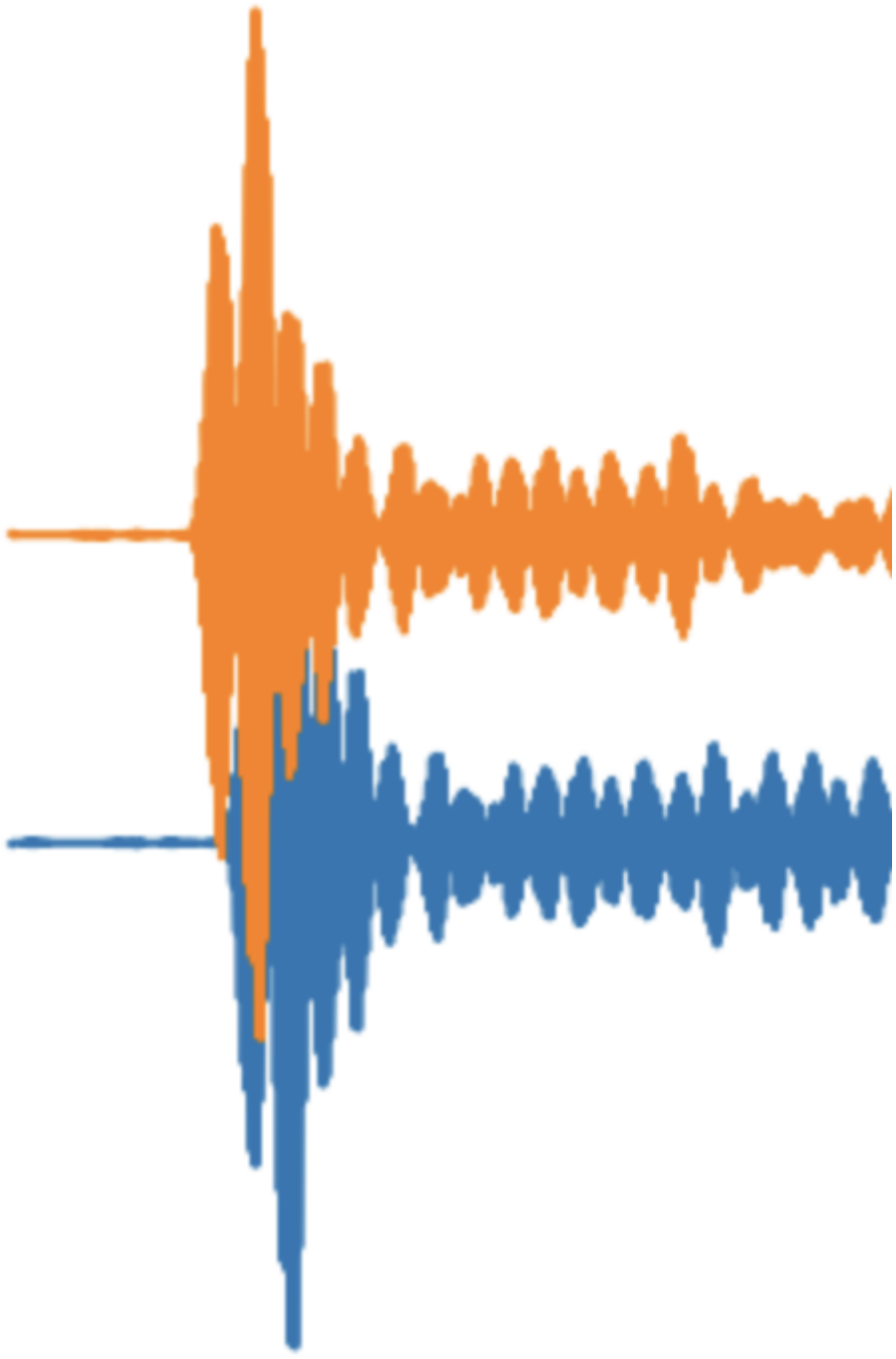
If enough time goes by without a sound being detected, a timeout event occurs and is passed to the state machine which handles it accordingly, usually by awarding a point to a player.

This loop continues until either the game is quit, an error occurs, or one of the players wins.

System component details

Signal capture details

The definition of a captured signal is any segment of an incoming signal stream with "high" power. Intuitively this is just any part of the signal that isn't quiet. For example if we look at the following image of a subsection of an incoming audio stream we see that the first section has no significant audio content, but then there is clear signal afterwards. The way this is translated into code is by looking for subsections of the signal where the RMS value is higher than a desired threshold.



To achieve this task, the signal capture object maintains a five second circular buffer storing the signals observed in the left and right channels. When a new input signal is passed in to be processed, the signal capture processor iterates over the block in chunks of size `window_len`, in this case equal to `.01 * Fs`. For each sub-block, if the RMS value of either the left or right channel is greater than the desired threshold (50 for my demo) then the sub-block is added to the current capture, otherwise the current capture is halted and the captured signal is added to the list of captured signals.

After a new captured signal is added to the list, a notification is sent to the game engine to tell it to process the event. The signal capture code continues in this way as long as new data is fed into it.

In pseudo-code the flow is as follows:

```
for window in signal_block:
        if RMS( left_channel_window ) > 50 or RMS( right_channel_window) >
50:
                current_capture_left.append( left_channel_window )
                current_capture_right.append( rightt_channel_window )
        else:
                captured_signals.append(current_capture_left,
current_capture_right)
                current_capture_left = []
                current_capture_right = []
                notify(game_engine)
```

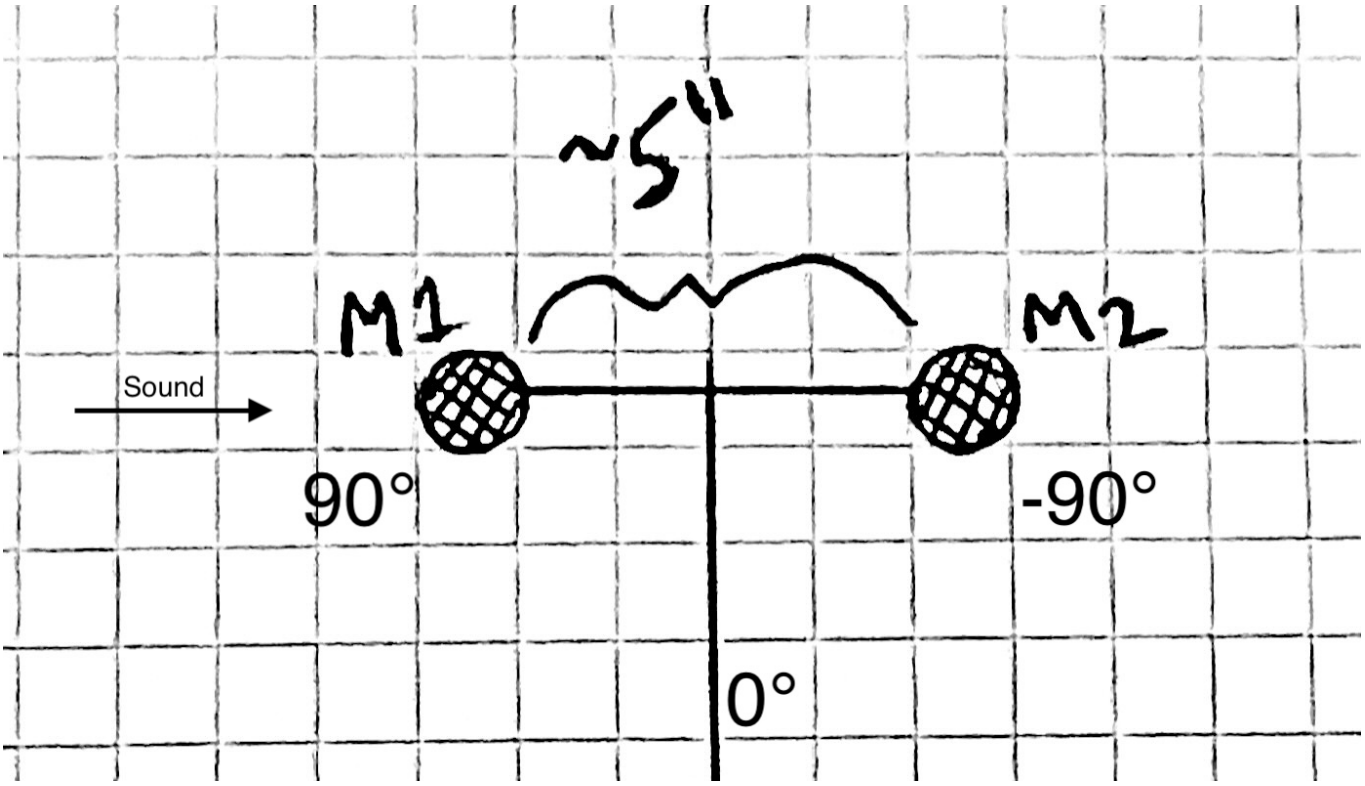
Delay and angle estimation

When a new sound is detected, the filtered signal is sent to the game engine which first estimates the delay, and then uses the delay to estimate the angle. There are two possible techniques for estimating the delay available in the code: cross-correlation and beam-forming. They perform similarly but have a slightly different mechanism.

Cross-correlation

The cross-correlation technique is done by finding the delay d that leads to the highest correlation between the signals. I.e. one signal is delayed by some candidate delay d_i and then the correlation is taken between the signals, whichever candidate delay leads to the maximum value is considered to be the best estimate for the delay between the signals.

In the case of this project, there is a maximum expected delay between the two microphones which is a function of the distance between the mics. This occurs when the sound is traveling along the same axis upon which the mics are lined up, thus the sound first hits one microphone, then travels the distance between them and hits the next one.



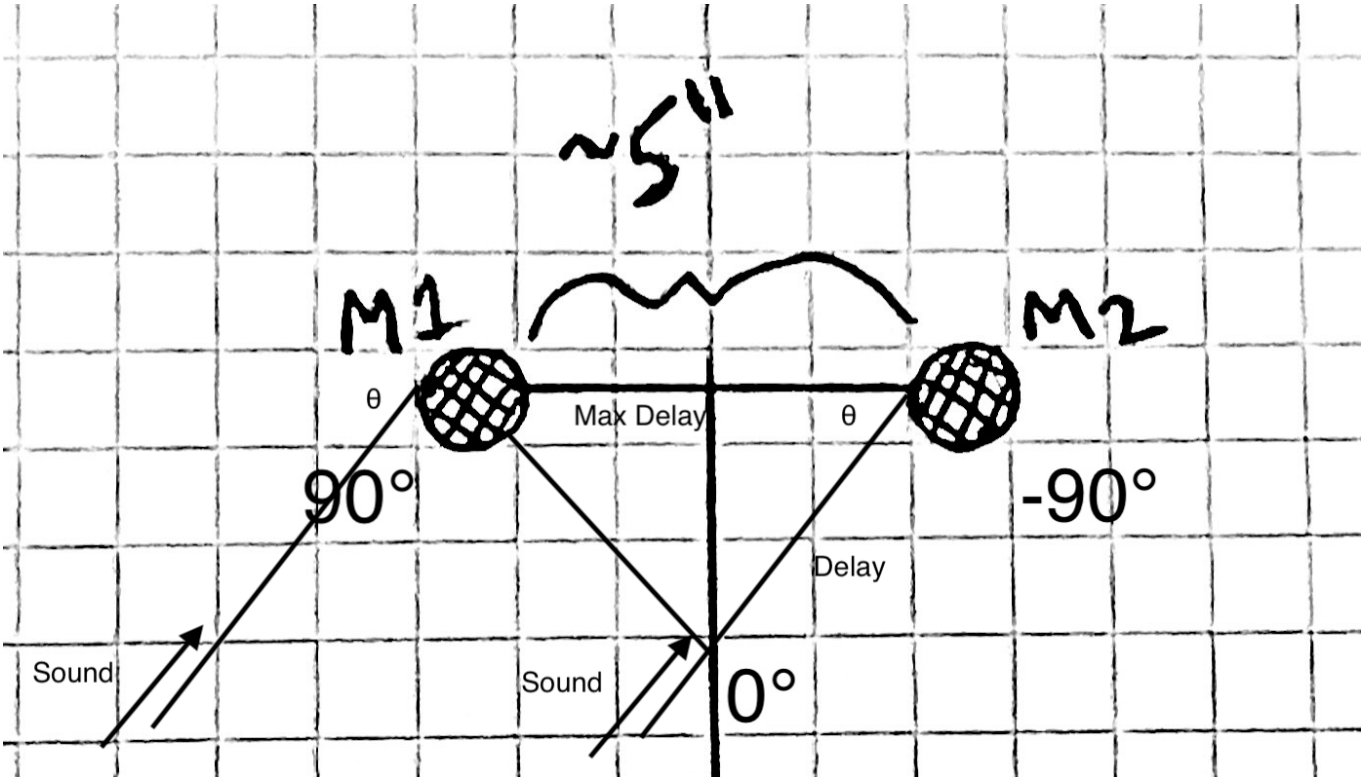
For example, most of the time the distance I used was 5 inches, so the max delay in time between the mics is $\frac{\text{dist}}{\text{speed of sound}} = \frac{(5/12)*.3048m}{340m/s} \approx 0.00037$ seconds, which is floored to 17 samples when using a sampling rate of 48 kHz. Thus the max delay between two signals that we expect is 17 samples. For this reason I only look at the correlation between ± 17 samples, and took the argmax of this range as the estimated delay.

Beam-forming

This technique is similar to cross-correlation in that a calculation is done for each possible delay from -17 to $+17$, however in this case instead of calculating the correlation between each pair of signals, they are added together and the power is calculated for each of the summed signals. The delay is estimated to be whichever offset led to the signal with the highest power.

Angle estimation

Once the delay is estimated, we can estimate the angle as demonstrated in the following diagram.



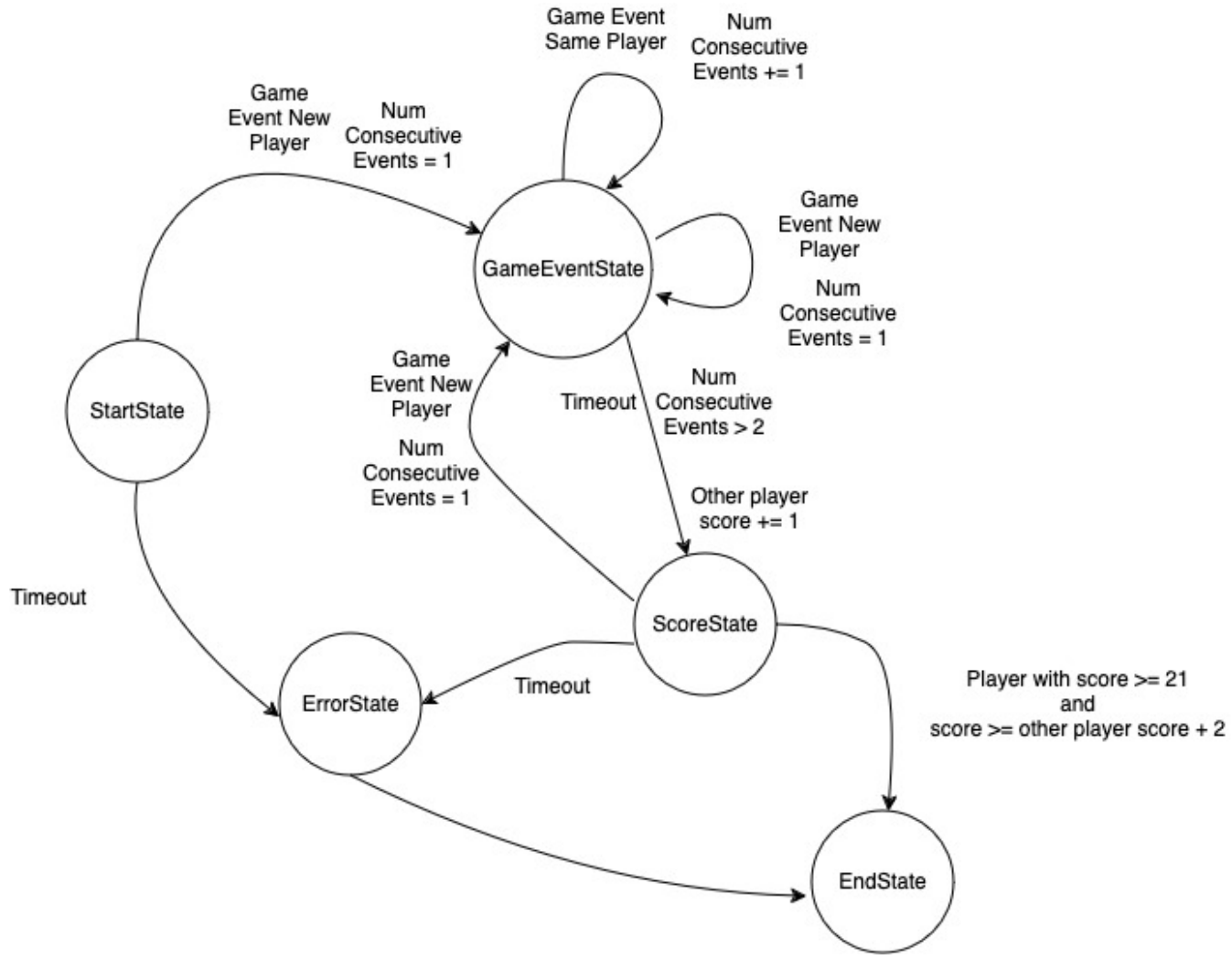
From this we can tell that the angle of the sound can be estimated by $\theta = \cos^{-1} \frac{\text{delay}}{\text{max delay}}$. After computing this angle, the only other step the code takes is to convert the result into degrees for a more user-friendly output to the console and then maps the angle to the corresponding side of the table.

Game State Machine

The scorekeeping component of the program uses a very simple state machine to keep score. The brief overview is that it waits for game events or timeouts, if 3 or more game events occur on the same side of the table, a point is awarded to the other player, if a timeout occurs after a game event, a point is awarded to the player who did not touch the ball last, e.g. if someone hits the ball over the table, a timeout would occur and a point would go to the other player.

This state machine diagram attempts to explain the state flows as described above.

For example one sequence of events could be: the game starts at the StartState, if a serve happens that is a Game Event for a new player, so num consecutive events = 1. If another event occurs on the same side, num consecutive events = 2, if yet another event occurs on that side num consecutive events > 2 and the score state is entered. At this point either a new game event occurs and the process repeats, or a timeout occurs which for now is considered an error and the game ends.



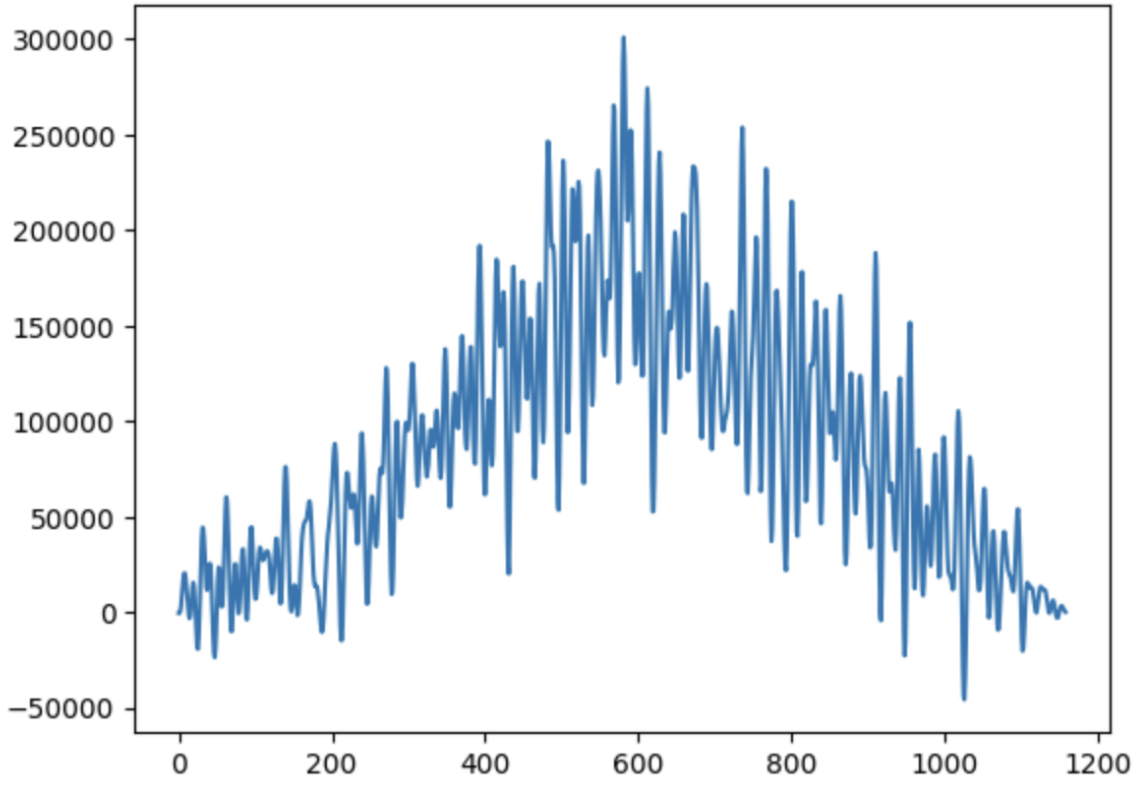
One of the key errors that the program makes while scoring is awarding multiple points if the ball bounces too many times on one side. I tried to mitigate this by turning off signal captures for a small period of time after a point is scored, but there are probably some other business rules that could be applied to reduce this.

Another difficulty in scoring is distinguishing between paddle-strikes, table-strikes and a ball bouncing on the floor. To mitigate this I introduced an extra power threshold within the game engine which rejects captures with too low mean RMS between the channels, this helps reduce the number of bounces on the floor that are captured but does not get rid of all of them. A possible improvement would be to try to classify signals based on their frequency content as one of those options, and using that info to improve the state machine's understanding of the game.

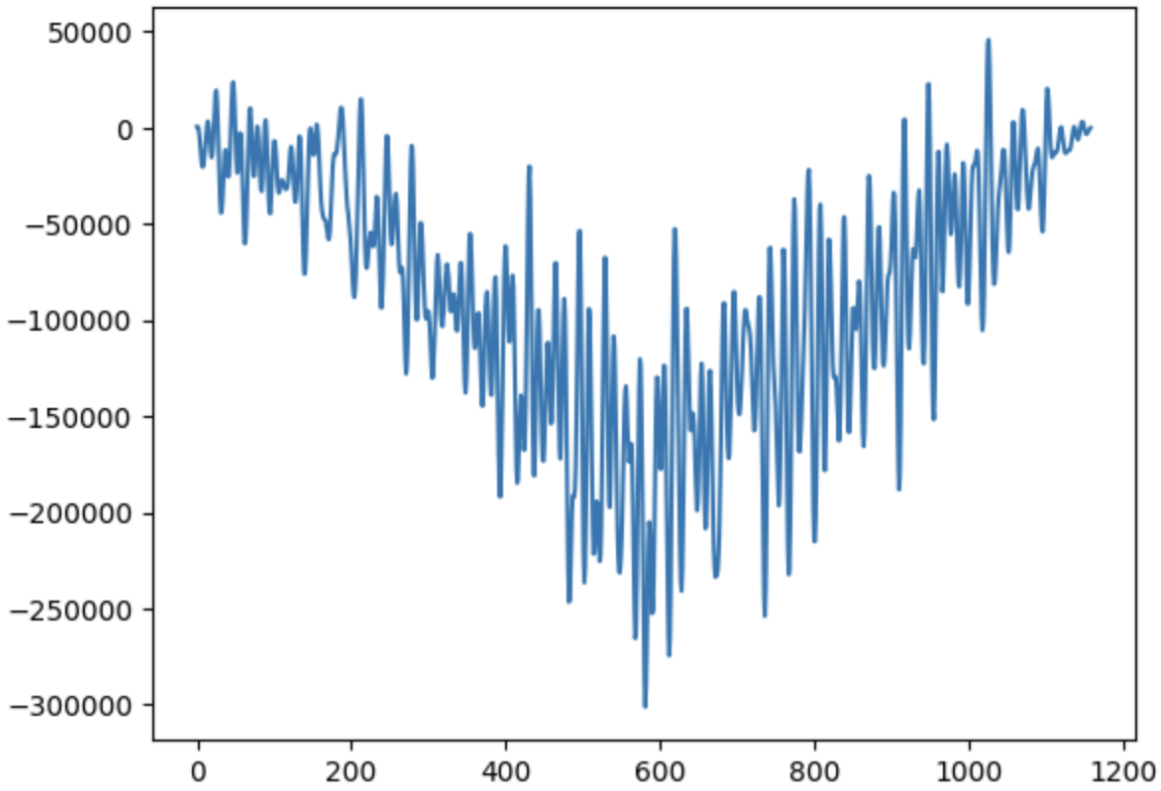
---

1. The reason for the polarity multiplication factor is that the signals generated by the two microphones I used were of opposite sign, so in order to properly compare them I needed to first make sure they had the same sign. This assumption might not always be correct so the game engine double checks the incoming signal polarity at the start by comparing the max of the cross-correlation between the two channels with no sign multiplication against that of the cross-correlation with one channel multiplied by -1.

   In the case of my microphones, here is a plot of this cross-correlation with the multiple of -1



   And here is the same but with no sign change



We can clearly see from this that the two signals have opposite sign. We also note that the correctly modified signal has a much larger max than the incorrect one, so based on this the code just chooses the sign that creates a larger maximum.↩