TARGET IMPACT LOCATION DETERMINATION: AN ANALYTIC METHOD

by

Wyatt Andresen

Honors Thesis

Appalachian State University

Submitted to the Department of Computer Science
and the Department of Mathematical Sciences
in partial fulfillment of the requirements for the degree of

Bachelor of Science

August, 2019

Approved by:

_____

Frank Barry, M.S.E.E., Computer Science Thesis Director

_____

Alice McRae, Ph.D., Honors Director, Department of Computer Science

_____

Holly Hirst, Ph.D., Mathematical Sciences Thesis Director

_____

William J. Cook, Ph.D., Honors Director, Department of Mathematical Sciences

Abstract

This thesis constructs and implements a model for 2-dimensional impact location determination and applies it to a simple experimental setup. In seeking a versatile solution, only minimal data is used to determine the impact location. In practice, this includes sound waveform data collected by microphones and the location of the microphones themselves. Using this data, the time of arrival (TOA) of the sound can be computed. Then, using TOAs from multiple microphones, time deltas, known as time differences of arrival (TDOAs), can be found. Equipped with TDOAs and the locations of the microphones, hyperbolae can be constructed and intersected to determine the sound's point of origin.

Strikingly, hyperbola construction and intersection proved to be simple to define and implement. Instead, the generation of TOAs and choosing a final point of origin proved to be much more sophisticated problems; the results of which further reinforced the importance of choosing accurate TDOAs. It was found that location determination had varying degrees of success, with the best results found near the center of the experimental apparatus where all microphones are roughly equidistant. Finally, ideas are discussed for further enhancement and areas needing additional iteration.

Contents

List of Figures

Listings

Chapter 1

Introduction

Suppose you had a slingshot and attempted to strike a target on a range. Though it would be reasonably clear whether or not you hit the target with your sling, it would be hard to determine where exactly you had struck the target. Would it be possible to determine the impact location of your sling upon the target using an array of microphones around the target? How? And to what degree of accuracy?

Determining the location of an object using sound waves or electromagnetic waves is a very relevant and modern topic in many fields such as security and air-traffic control. One method of sound location determination involves the generation of hyperbolae using time deltas, which are then intersected in order to determine the originating location of the sound. This method is particularly useful with limited information about the signal's origin.

This thesis will abstract the above scenario by presenting, implementing, and analyzing the results of a model for the determination of a sound's location in 2-dimensional space using low-cost microphones modules to provide real-time data collection.

Chapter 2

Background & Theory

2.1   A 1-Dimensional Model: Time Deltas

A good starting point to consider is a 1-dimensional model where impacts will occur on a single axis. This will provide a model from which to expand upon in the 2-dimensional model.

In this model we have two microphones, one at each end of an axis. The two microphones are separated by some distance d. Thus, the location of microphone m_1 is $(0, 0)$, and the location of microphone m_2 is $(d, 0)$.

Without loss of generality, suppose that an impact occurs at some point closer to m_1 on the axis between the two microphones, as shown in Figure 2.1.

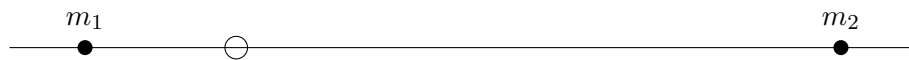Figure 2.1: A 1-dimensional impact.

Each microphone will output time data representing when the sound was registered in that microphone. This data can be extracted to form a time delta, Δt. The distance covered during this time, $d_{\Delta t}$, can be calculated by the product of Δt and v, the speed of sound in the material.

$$\Delta t = t_2 - t_1$$

$$d_{\Delta t} = \Delta t \cdot v$$

Then, the distance between the impact and the first microphone, d_{m_1} is some unknown distance and the distance between the impact and the second microphone, d_{m_2}, is the sum of this unknown distance and $d_{\Delta t}$.

$$d_{m_2} = d_{m_1} + d_{\Delta t}$$

This is visualized graphically in Figure 2.2, below.
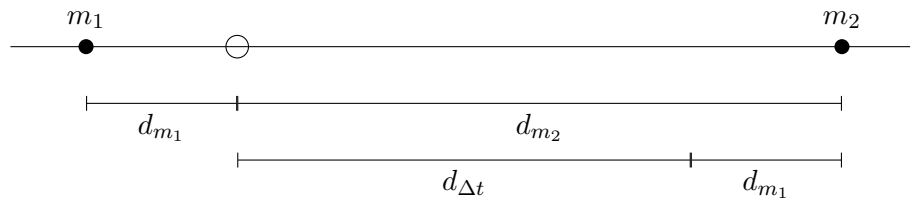
Figure 2.2: A 1-dimensional impact broken into its constituent parts.

From here the problem becomes quite simple to solve algebraically. $d_{\Delta t}$ is known, and the sum of d_{m_2} and d_{m_1} is d. Then, all that is required to pinpoint the impact is d_{m_1}.

$$d_{m_1} + d_{m_2} = d$$
$$d_{m_1} + (d_{m_1} + d_{\Delta t}) = d$$
$$2d_{m_1} + d_{\Delta t} = d$$
$$2d_{m_1} = d - d_{\Delta t}$$
$$d_{m_1} = \frac{d - d_{\Delta t}}{2}$$

Now that d_{m_1} is known it can be concluded that the impact occurred at $(d_{m_1}, 0)$.

2.2 A 2-Dimensional Model: Circles

One approach to this problem is to triangulate the point of impact with circles. Given the time that the signal occurred, along with the time that the signal reached at least three stations, three or more circles can be built which may be intersected to find the origin of the signal.

The radius of each circle is the product of the velocity of the signal with the time taken to reach the

station. Then, given a time of origin, t_o, a time of arrival for each station, t_i, and the velocity of the signal in the medium, v, the radius of each circle can be found.

$$r_i = v \cdot (t_i - t_o)$$

Suppose three stations are placed at known locations, each at the center of their own circle. Clearly, the circles will intersect at the point of the signal's origin as shown in Figure 2.3. All that is left is to determine the intersection points of each circle with the other circles; there will be one or two intersection points per pair of circles; then the impact is the common point of intersection.
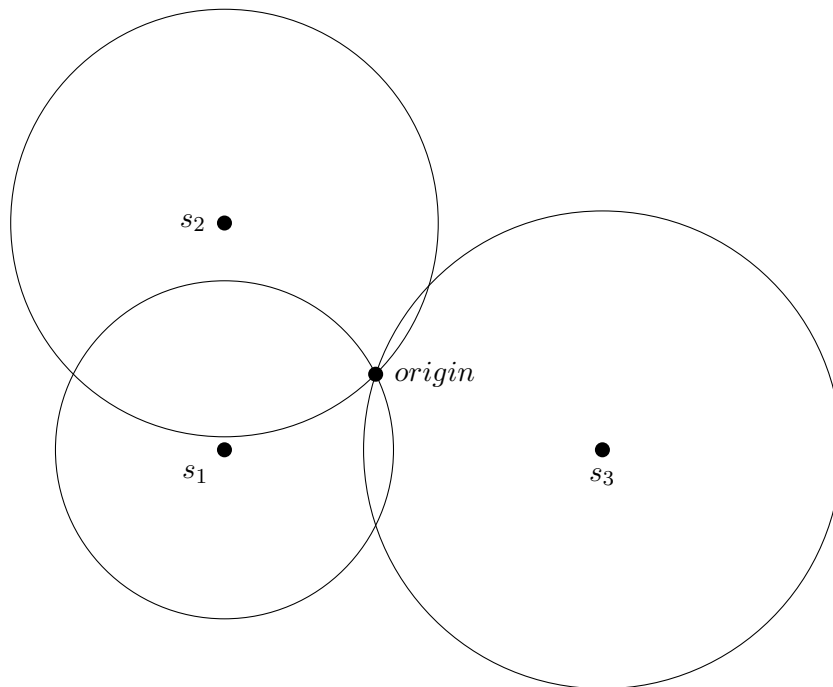
Figure 2.3: Triangulation using circles.

However, what happens when there is no data regarding the time of the signal's origination? Immediately the problem becomes significantly more complex because the radii of the circles are unknown and the available data, the time of arrival of the signal to the station, is insufficient in determining these radii. A review of literature found that it was far simpler to build and intersect hyperbolae using time deltas [2, 3, 5, 6]. However, discussion of this methodology was limited, necessitating the following section.

2.3 A 2-Dimensional Model: Hyperbolae

A hyperbola is the set of points such that the difference in distances between each point and two fixed points, the foci, is constant. Thus, the logical basis for this model is that $d_{\Delta t}$ is that constant difference.

2.3.1 Preliminaries

A hyperbola with center (h, k) can be defined by

$$\frac{(x-h)^2}{a^2} - \frac{(y-k)^2}{b^2} = 1$$

for a hyperbola with a horizontal transverse axis (horizontal hyperbola), or

$$\frac{(y-k)^2}{a^2} - \frac{(x-h)^2}{b^2} = 1$$

for a hyperbola with a vertical transverse axis (vertical hyperbola) where a is the shortest distance from the hyperbola's center to a vertex and b is the distance, perpendicular to the vertex, from the vertex to its asymptotic line. Additionally, a and b give rise to a trigonometric relationship with a third characteristic, c, defined as the distance from the center of the hyperbola to one of its foci.

$$c^2 = a^2 + b^2$$
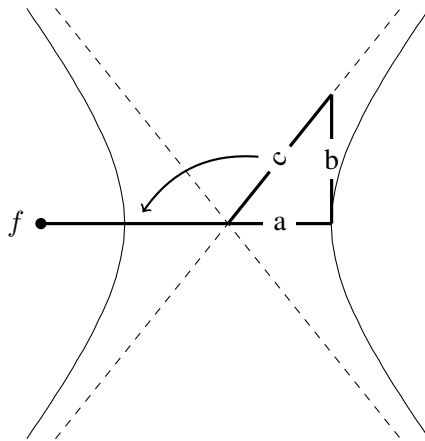
Figure 2.4: Characteristics of a hyperbola.

2.3.2 Construction of Hyperbolae From Time Deltas

Let m_1 and m_2 lie at (x_1, y_1) and (x_2, y_2) respectively. Without loss of generality, suppose that an impact occurs at some point closer to m_1 just as before. Then Δt and $d_{\Delta t}$ can be determined as in the 1-dimensional case. However, this time it cannot be presumed that the location of the impact occurred upon the axis between the two microphones. Instead, a hyperbola can be generated from this data with the impact lying somewhere upon it. In essence, $d_{\Delta t}$ acts as the constant difference that defines the hyperbola. Note that while the following example visualizes a horizontal hyperbola, this will also work for a vertical hyperbola.

First, h and k can be found from the midpoint of m_1 and m_2.

$$(h, k) = \left(\frac{x_1 + x_2}{2}, \frac{y_1 + y_2}{2} \right)$$

Then, by assuming that the impact occurred horizontally between the two microphones, $d_{\Delta t}$ can be shifted onto the line between the two foci, showing that the point of impact found in the 1-dimensional model was actually the point where the hyperbola built from $d_{\Delta t}$ intersected the transverse axis. Then a is simple to calculate.

$$a = \frac{d_{\Delta t}}{2}$$

The visual for this calculation can be seen in Figure 2.5.
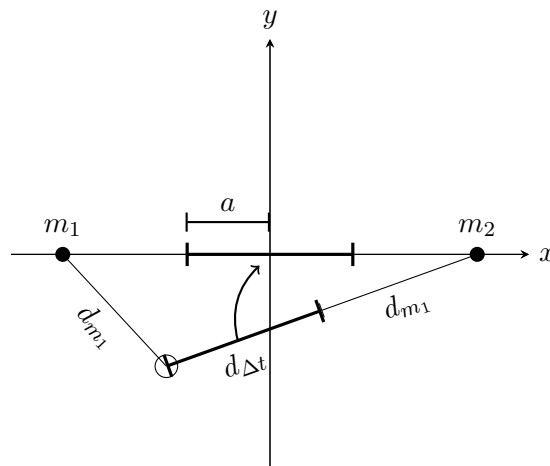
Figure 2.5: Finding a from $d_{\Delta t}$.

6

c is clearly half the distance between the two microphones and a is now known. Then,

$$b^2 = \left(\frac{d}{2}\right)^2 - a^2$$

With h, k, a, and b^2 found, the construction of the hyperbola upon which the impact lies is complete as shown in Figure 2.6.
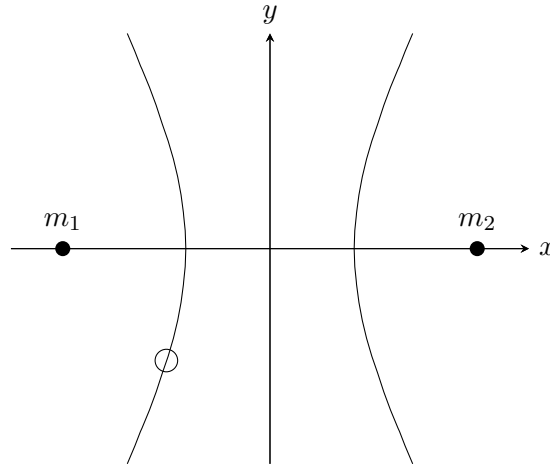
Figure 2.6: A hyperbola constructed using a time delta.

However, this singular hyperbola does not provide enough information to pinpoint the location of the impact.

2.3.3 Intersecting Hyperbolae

In order to find the point of impact, additional information is required. Continuing with hyperbolae additional information can be obtained through more microphones and thus obtain more hyperbolae. Suppose then that two additional microphones are added, creating a square formation. Microphones are axis aligned to allow for simple hyperbola generation. Then additional hyperbolae may be constructed from the additional microphones, as in Figure 2.7.

While algebraic solutions are simple for some cases of intersecting hyperbolae, there are numerical methods that are equally simple and consistent across all cases. To this end, each hyperbola is first
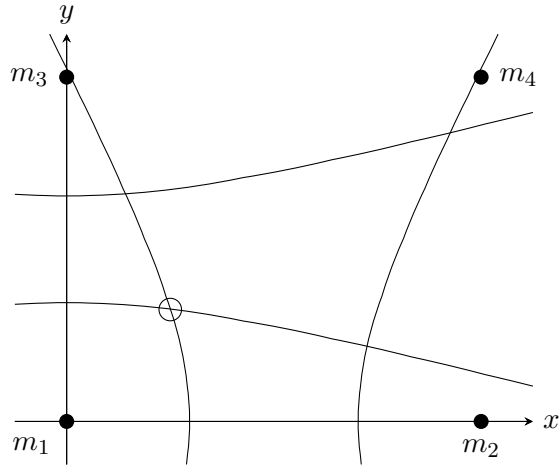
Figure 2.7: Two hyperbolae intersecting at the impact.

solved for y. First, for horizontal hyperbolae,

$$y = \sqrt{-\left(1 - \frac{(x-h)^2}{a^2}\right)b^2 + k}$$

and then for vertical hyperbolae.

$$y = \sqrt{\left(1 + \frac{(x-h)^2}{b^2}\right)a^2 + k}$$

Of course this form does not form the whole hyperbola alone. Half of the hyperbola will be generated by selecting the positive solution to the square root and the other half will be generated by selecting the negative solution to the square root.

Horizontal hyperbolae will be split on their transverse axis while vertical hyperbolae will select only one branch of the hyperbola at a time. This is shown in Figures 2.8 and 2.9. This requires the numerical method used for intersection to cycle through all four possibilities, gathering all intersection points between the hyperbolae.
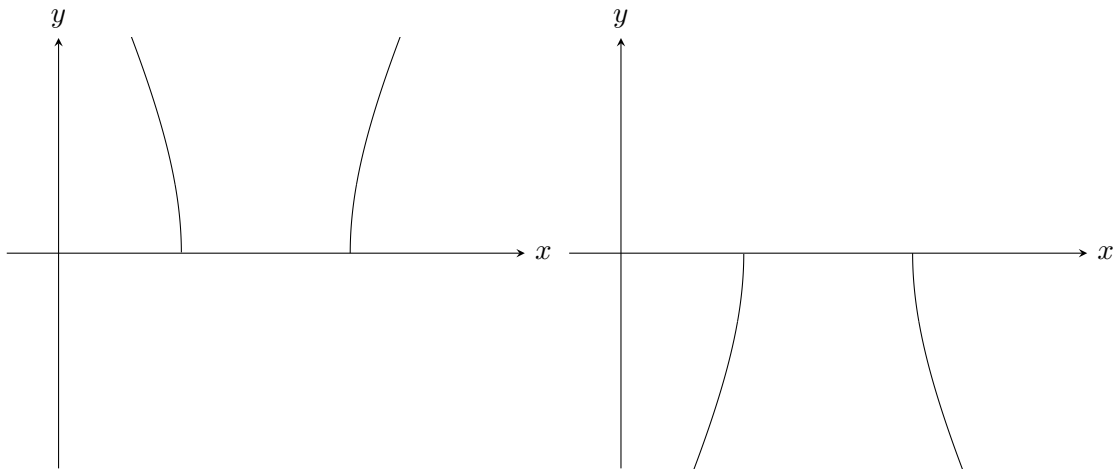
Figure 2.8: Solutions of horizontal hyperbolae.
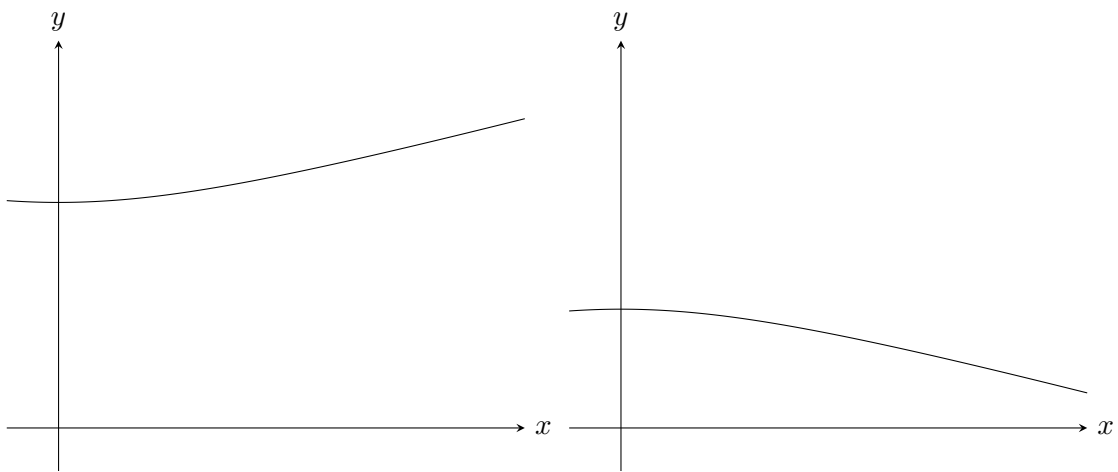
Figure 2.9: Solutions of vertical hyperbolae.

9

Chapter 3

Implementation

3.1 Experimental Model

To get an idea for how the model performs in practice, an experimental model is necessary. This will allow easy testing of the algorithms discussed in following sections. The experimental model discussed below was developed by the Computer Science Thesis Director, Frank Barry, in coordination with this thesis' research goals.

The experimental model consists of a 1/8" steel plate with a 32×32 cm grid engraved upon its surface. Microphones (mics) are mounted to the plate at the corners of the grid. The grid allows for reasonably accurate data gathering and testing. Sounds are generated by tapping a nail on a location of the grid. The nail is precisely directed to a location using an impromptu aiming device: the empty barrel of a marker. A photo of the complete model can be seen in Figure 3.1 alongside an accompanying diagram in Figure 3.2.

Each mic emits a pulse indicating that a received sound exceeds some threshold. These signals then enter into a microcontroller, triggering an interrupt on the port associated with that mic. The interrupt handler then begins sampling from each mic's port, collecting and storing the raw data. The raw waveform data can then be used to find time difference of arrival (TDOA) data.

3.1.1 Data Collection

A large set of data was collected for analysis using this experimental model. This was done by recording the microphone output at each grid location, three times each, creating three sets of microphone data
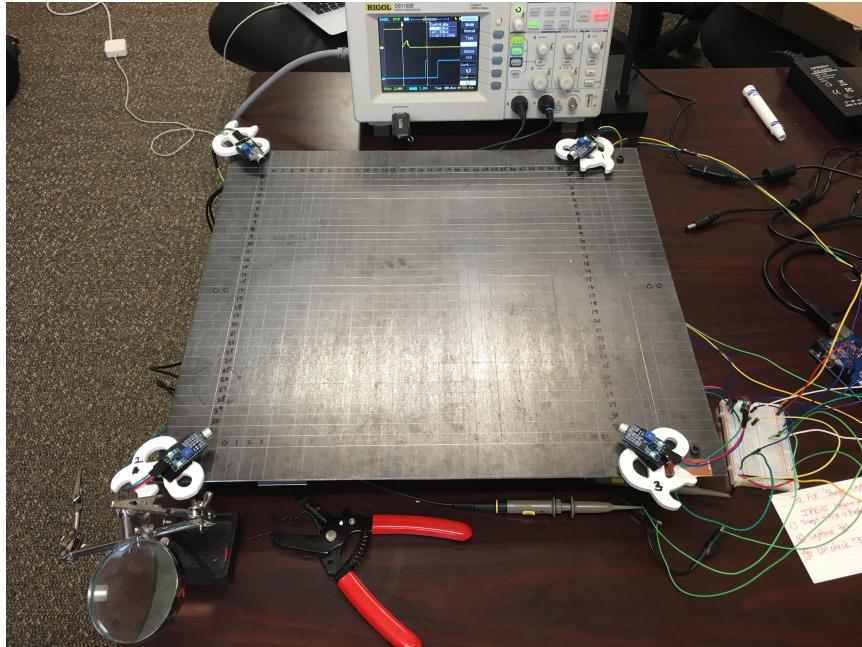
Figure 3.1: The experimental model.
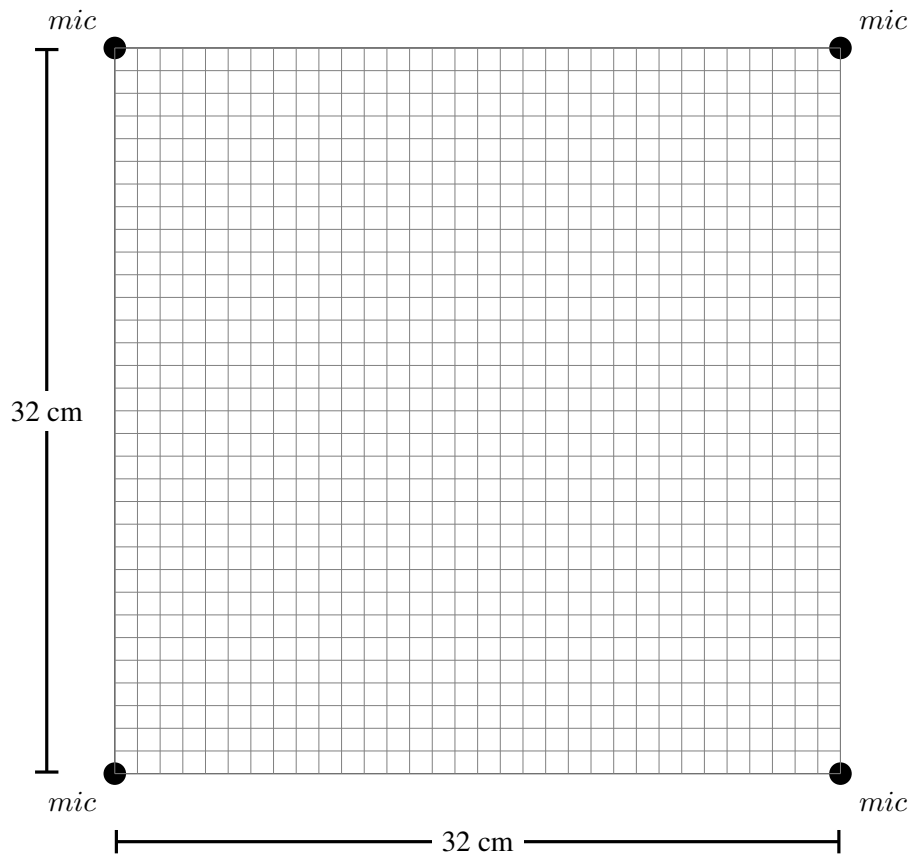
Figure 3.2: The experimental model.

for each location on the grid. Additionally, an oscilloscope was attached to two of the microphones and used to verify the validity of the data at a glance as it was collected.

3.2 Program Control Flow

Figure 3.3 shows the general control flow for the ImpactFinder class as it executes for one impact's worth of microphone data. In practice, this control flow lies within a while loop waiting for additional sets of microphone data. The remainder of this section will discuss each piece of the control flow diagram.

```
          ┌──────────────────┐
          │      Start       │
          └──────────────────┘
                   │
                   ▼
         ╱─────────────────────╲
        ╱    Microphone Data     ╲
        ╲─────────────────────╱
                   │
                   ▼
          ┌──────────────────┐
          │  Determine TDOAs │
          └──────────────────┘
                   │
                   ▼
          ┌──────────────────┐
          │Generate Hyperbolae│
          └──────────────────┘
                   │
                   ▼
          ┌──────────────────┐
          │Intersect Hyperbolae│
          └──────────────────┘
                   │
                   ▼
          ┌──────────────────┐
          │Choose Intersection│
          └──────────────────┘
                   │
                   ▼
          ┌──────────────────┐
          │       Stop       │
          └──────────────────┘
```
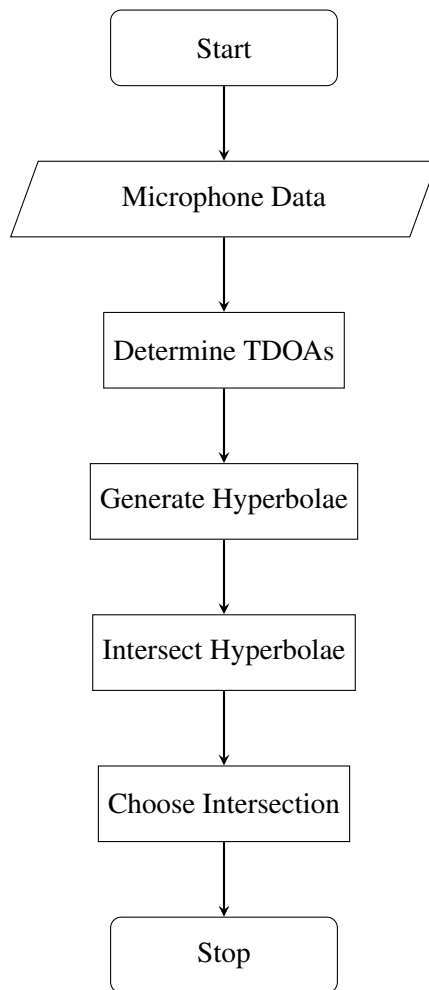
Figure 3.3: ImpactFinder control flow diagram.

3.2.1 Microphone Data

Data Format

The raw data itself consists of an array of four bit numbers, each four bit number representing the state of each mic at the time of that sample, as shown in Figure 3.4. The labels indicate which bit is associated with which mic, as well as the time associated with each row of data. The sample rate is one sample per 2μs, or 500kHz.

b_3	b_2	b_1	b_0	t_{n+0}
b_3	b_2	b_1	b_0	t_{n+1}
b_3	b_2	b_1	b_0	t_{n+2}
b_3	b_2	b_1	b_0	t_{n+3}

Figure 3.4: The raw data format of each sample.

Waveform Analysis

Figure 3.5 contains two real examples of the signal form for impacts at (1,1), close to Mic 0, and (16,16), at the center of the grid, created by feeding the raw data into a python script (See Appendix B.1). Note that in each example the initial bump in the signal is directly correlated to the distance away from the mic that the impacts occurred relative to the first receiving mic.

For instance, looking at the data for point (1,1), it is clear that Mic 0 registered the sound first, followed by Mic 1 and Mic 2 in close succession, and finally by Mic 3. Visually, then, it is clear that the sound occurred closest to Mic 0, equidistant to Mic 1 and Mic 2, and furthest from Mic 3. Already it is easy enough to construct a mental idea of the location of the impacts by observing the waveform data. The next section will discuss how these observations are translated into algorithms.

Figure 3.5: Waveform Examples

3.2.2 Determining TDOAs

The first order of business is to process the raw input data to obtain TDOAs for each microphone. Then hyperbolae can be constructed from these TDOAs. Choosing accurate TDOAs is essential to building appropriate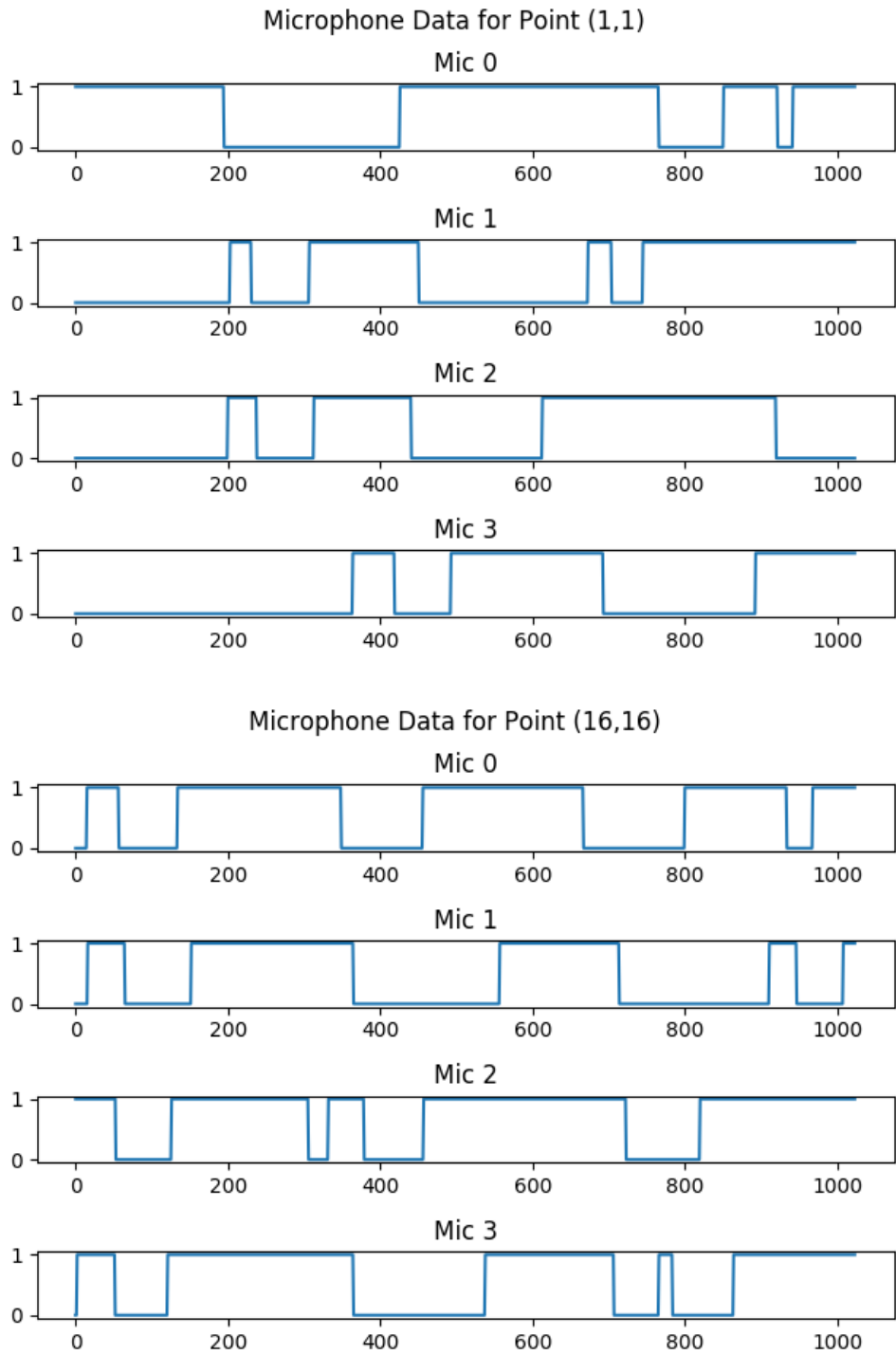 hyperbolae. If an inaccurate TDOA is chosen, then the built hyperbolae will be flawed and intersection will be inaccurate. Three algorithms were tested in order to generate accurate TDOAs: a cross correlation algorithm, a first hit algorithm, and a first sustained hit algorithm.

Cross Correlation Algorithm

Cross correlation is a common signal processing technique for comparing two signals using a sliding dot product [4]. Essentially, the larger the dot product between two signals, the more similar the two signals are.

First, the dot product of the two signals is calculated and stored as the maximum. Next, one signal is chosen and shifted a step to the left. Then, the dot product of the two signals is again calculated. If this new dot product is greater than the previous, it becomes the new maximum. This process is then repeated by continually updating the maximum dot product as well as the number of steps needed to produce that dot product.

Consider the cross correlation algorithm in the context of the microphone data shown in Figure 3.5. The waveforms for each mic look quite similar overall. Then cross correlation should return the number of shifts required for any two signals to match the most. Recalling that each data point is taken every 2μs, the TDOA, in μs, is double the shift.

Listing 3.1: Cross Correlation Algorithm

```
1  /* crossCorrelation - Determines TDOA between two microphone signals using
2   * cross correlation.
3   * @param: uint1024_t c - Raw signal data of the "closest" microphone.
4   * @param: uint1024_t m - Raw signal data of the microphone to be compared
5   *  against.
6   * @return: TDOA between mic c and mic m.
7   */
8  double DataParser::crossCorrelation(uint1024_t c, uint1024_t m) {
9      // Would generally only apply to the comparison of the closest mic
```

```
10      // with itself.
11      if (c == m) {
12          return 0;
13      }
14      else {
15          // bestFit holds the shift at which the highest sum was achieved.
16          // bestSum holds this highest sum.
17          int bestFit = -1, bestSum = 0;
18          // Perform sliding dot product between c and each "iteration" of
19          // m. We will only shift halfway.
20          for (int i = 0; i < 511; i++) {
21              // Temporary sum variable.
22              int tempSum = 0;
23              // Shift m over
24              uint1024_t shiftm = m << i;
25              // Perform a bitwise and between c and m.
26              uint1024_t dotprod = c & shiftm;
27              // Count the number of ones in dotprod. This is the sum. We will
28              // only look at the first half of dotprod.
29              uint1024_t shift = 1;
30              for (int j = 0; j < 1023; j++) {
31                  if ((shift & dotprod) > 0)
32                      tempSum++;
33                  shift = shift << 1;
34              }
35              // If tempSum is higher, we have a new
36              // best fit.
37              if (tempSum >= bestSum) {
38                  bestSum = tempSum;
39                  bestFit = i;
40              }
41          }
42          // bestFit contains the number of "slides" making up the best sum.
```

```
43        // Multiply this number of slides by the time elapsed between each
44        // data point (2 microseconds). This is the TDOA in microseconds.
45        // Then, convert to seconds.
46        return bestFit * 2 * .000001;
47    }
48 }
```

This method appeared most promising at first; however, when it was applied the TDOAs returned from cross correlation were faulty in many cases. In an attempt to figure out what the issue could have been the waveform visuals in Figure 3.5 were constructed for many different data points. Cross correlation, in this case, relies on the two compared waveforms looking quite similar to each other to be successful. However, the nature of the experimental model prevents cross correlation from being effective because this is not always the case. Looking at point (1,1) in Figure 3.5, this becomes clear. Mic 0 has a very clearly different waveform from that of the other mics due to its proximity to the sound. Further, the TDOA returned by cross correlation is relative to the speed of sound through the apparatus not through air as was originally intended. Ergo, experimental determination of the speed of sound through the apparatus is required. An additional downside of the cross correlation algorithm is that it is extremely slow compared to the following algorithms.

First Hit Algorithm

The goal of the first hit algorithm was to remove the previous reliance on ideal waveforms, and it was developed for the purpose of this thesis. Ironically, this resulted in the simplest algorithm. Similar to cross correlation, this algorithm would rely on the propagation of sound through the apparatus which would again require experimental determination of the speed of sound.

Listing 3.2: First Hit Algorithm

```
1 /* firstHit - Determines "true start time" as time of first "on" data
2  * point, and therefore relative to the speed of sound in the apparatus.
3  * @param: uint1024_t m - Raw signal data of the microphone.
4  * @return: True start time of mic m.
5  */
6 double DataParser::firstHit(uint1024_t m) {
```

```
7      uint1024_t one = 1;

8      for (int i = 1023; i > 0; i--) {

9          if ((m & (one<<i)) > 0) {

10             return (double)(1023-i) * 2.0 * .000001;

11         }

12     }

13     return 0;

14 }
```

This algorithm, while still imperfect, presented more consistent results across the grid as a whole.

First Sustained Hit Algorithm

The goal of the first sustained hit algorithm was to identify the true start time of the signal relative to the speed of sound. The initial jolts of activation before the main bump in Figure 3.5 are attributed to the propagation of the sound through the apparatus itself as opposed to through the air. By analyzing multiple examples of the waveform, it is clear that the true start time can generally be found by determining the start time of the sustained bump. Thus, this algorithm was developed as an enhancement to the previous first hit algorithm.

Listing 3.3: First Sustained Hit Algorithm

```
1  /* firstSustainedHit - Determines "true start time" as first sustained

2   * hit, and therefore relative to the speed of sound in air.

3   * @param: uint1024_t m - Raw signal data of the microphone.

4   * @return: True start time of mic m.

5   */

6  double DataParser::firstSustainedHit(uint1024_t m) {

7      uint1024_t one = 1;

8      for (int i = 1023; i > 256; i--) {

9          int count = 0;

10         for (int j = 0; j < 80; j++) {

11             if ((m & (one << (i-j))) > 0) {

12                 count++;

13                 if (count >= 80) {
```

```
14          cout << 1023-i << endl;

15          return (double)(1023-i) * 2.0 *.000001;

16        }

17      }

18    else

19      break;

20  }

21 }

22 return 0;

23 }
```

However, this algorithm suffered from a similar issue to cross correlation in that it again was forced to rely on an ideal waveform. Specifically, for sounds relatively close to a microphone there is no split between the initial jolt and the sustained bump. This caused the true start time for these sounds to be consistently off. On the other hand, the algorithm performed extremely well for well defined waveforms consisting of both an initial jolt and a sustained bump.

3.2.3 Generating Hyperbolae

Generating hyperbolae is simple to do once TDOAs have been determined. All that is required to generate hyperbolae are the corresponding microphone locations (which are the foci of the hyperbola) and the TDOA between those two microphones. Then hyperbola generation is performed exactly as discussed in Section 2.3.2, within the Hyperbola class. The UML class diagram for the Hyperbola class is shown in Figure 3.6 below. Note that the type of hyperbola (horizontal or vertical) is also stored.

3.2.4 Intersecting Hyperbolae

With all hyperbolae now generated, they can now be intersected. The first step of intersection is to determine how the bisection method (which will actually perform the intersection) needs to be called. The reason for this is that horizontal hyperbolae have a gap at their center that is undefined. However, the size of this gap is known because the distance from a hyperbola's center to its nearest vertex is a which is stored within the Hyperbola object.

Once this initial processing is done to narrow the bounds of the bisection, bisection can now be
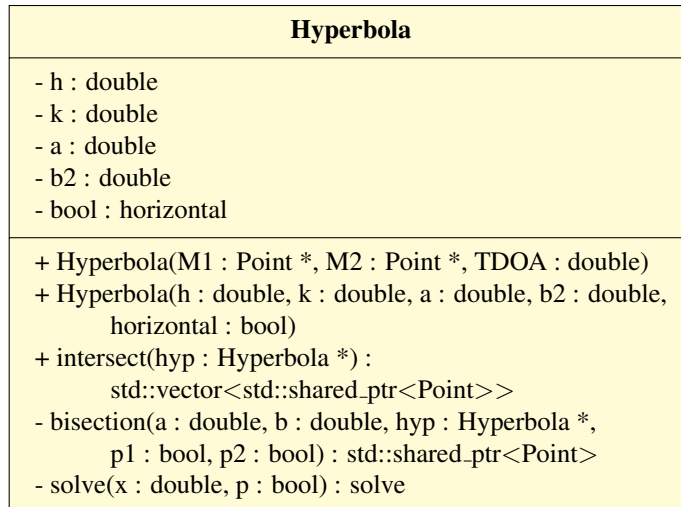
Figure 3.6: Hyperbola UML class diagram.

performed. Recalling from Section 2.3.3 that each hyperbola will have two solutions, we know that there will be four total calls to bisection in order to cover all combinations. The bisection algorithm is shown below. The bisection algorithm is ideal because the bounds of the intersection region are well defined and narrow, allowing for quick and reliable convergence.

Listing 3.4: Bisection Algorithm

```
1  /* Performs bisection method to find intersection point.
2   * @param: double a - left bound
3   * @param: double b - right bound
4   * @param: Hyperbola * hyp - second hyperbola
5   * @param: bool p1 - select positive or negative solution of
6   *    square root for this
7   * @param: bool p2 - select positive or negative solution of
8   *    square root for hyp
9   * @return: Intersection point (if there is one) or nullptr (if there is
10  *    not).
11  */
12  std::shared_ptr<Point> Hyperbola::bisection(double a, double b,
13      Hyperbola * hyp, bool p1, bool p2) {
14      // The "function" that we are applying bisection to is
15      // <this-hyp>. By substracting the two functions, bisection
```

```
16      // should find the point at which the two functions are equal,
17      // i.e. where they intersect.
18      double c, solvea, solveb, solvec;
19      solvea = this->solve(a,p1) - hyp->solve(a,p2);
20      solveb = this->solve(b,p1) - hyp->solve(b,p2);
21      if ((solvea * solveb >= 0) || std::isnan(solvea*solveb)) {
22          return nullptr;
23      }
24      while ((b-a) > EPSILON) {
25          // Calculate midpoint
26          c = (a+b)/2;
27          solvec = this->solve(c,p1) - hyp->solve(c,p2);
28          solvea = this->solve(a,p1) - hyp->solve(a,p2);
29          if (solvec == 0.0)
30              break;
31          else if (solvec * solvea < 0)
32              b = c;
33          else
34              a = c;
35      }
36      return std::shared_ptr<Point>(new Point(c, this->solve(c,p1)));
37  }
```

3.2.5 Choosing The Intersection Point

By now, a vector of intersection points between all the various hyperbolae has been generated. The last step is to narrow the list of intersection points by weeding out those that are clearly out of range and then to average the remaining candidates. A simple way to do this is to ensure that each intersection point is within the quadrant of the grid dominated by the closest mic.

However, even within one quadrant there can be significant variance; so, a simple averaging of all of the remaining points will not be effective. To counter this the outliers must be removed from the pool of intersection points so that the remaining points can be averaged to estimate the final intersection point. Two algorithms were implemented to allow for this; a statistical method and a clustering method.

A Statistical Method for Excluding Outlier Points

The statistical method starts by determining the mean point of all the candidate intersection points. It then calculates the mean distance that each point lies from the mean point as well as the standard deviation from the mean. Finally, it excludes all points greater than two standard deviations from the mean.

The primary issue with this method is that, in some cases, there are no candidate intersection points that pass this test.

Listing 3.5: Statistical Method for Outlier Exclusion

```
1   /* SD - Uses standard deviation of mean distance of average point to
2    * exclude outlier points.
3    * @param: std::vector<std::shared_ptr<Point>> intersections - vector of
4    *  intersection points
5    */
6   std::vector<std::shared_ptr<Point>> ImpactFinder::SD(
7       std::vector<std::shared_ptr<Point>> intersections) {
8
9       // Mean x and y values to determine mean point.
10      double xsum = 0.0;
11      double ysum = 0.0;
12      for (uint i = 0; i < intersections.size(); i++) {
13          xsum += intersections[i]->x;
14          ysum += intersections[i]->y;
15      }
16      double xav = xsum/intersections.size();
17      double yav = ysum/intersections.size();
18
19      // Calculate mean distance from mean point.
20      double avdist = 0.0;
21      for (uint i = 0; i < intersections.size(); i++) {
22          double xval = intersections[i]->x;
23          double yval = intersections[i]->y;
24          double dist = sqrt(pow(xval-xav, 2) + pow(yval-yav, 2));
```

```
25        avdist += dist;
26    }
27    avdist = avdist/intersections.size();
28
29    // Calculate standard deviation of mean distance.
30    double sd = 0.0;
31    for (uint i = 0; i < intersections.size(); i++) {
32        double xval = intersections[i]->x;
33        double yval = intersections[i]->y;
34        double dist = sqrt(pow(xval-xav, 2) + pow(yval-yav, 2));
35        sd += pow(dist-avdist, 2);
36    }
37    sd = sqrt(sd / intersections.size());
38
39    // Now that we have the standard deviation, we will invalidate
40    // all points greater than two standard deviations from the mean
41    // distance. These points should be outliers.
42    for (uint i = 0; i < intersections.size(); i++) {
43        double xval = intersections[i]->x;
44        double yval = intersections[i]->y;
45        double dist = sqrt(pow(xval-xav, 2) + pow(yval-yav, 2));
46        if (dist > (sd * 2)) {
47            intersections[i]->x = -1;
48            intersections[i]->y = -1;
49        }
50    }
51
52    return intersections;
53 }
```

A Clustering Method for Excluding Outlier Points

The clustering method used is the Density-Based Spatial Clustering of Applications with Noise, or DBSCAN [1]. This algorithm chooses points and attempts to assign them to a cluster. For a group of

23

points to be considered a cluster they must be of some minimum density. Points that do not fall within a cluster are considered noise and excluded from the list of candidate intersection points.

The greatest issue with this method occurs when there are very few intersection points. In this case, a cluster can fail to be generated at all.

Listing 3.6: Clustering Method for Outlier Exclusion

```
/* DBSCAN - (Density-Based Spatial Clustering of Applications with Noise)
 * A data clustering algorithm to weed out problem points.
 * @param: std::vector<std::shared_ptr<Point>> intersections - vector of
 *  intersection points
 * @param: double maxdist - minimum distance between points
 * @param: int minpts - minimum number of points to constitute a cluster
 * @return: Cluster with enough points.
 */
std::vector<std::shared_ptr<Point>> ImpactFinder::DBSCAN(
    std::vector<std::shared_ptr<Point>> intersections,
    double maxdist, uint minpts) {

    // Convert intersections to a vector of pairs of <Point, int>.
    // This allows us to assign each point a cluster (int) while
    // maintaining them all within the same overarching data structure.
    std::vector<std::pair<std::shared_ptr<Point>, int>> points;
    for (uint i = 0; i < intersections.size(); i++) {
        std::pair<std::shared_ptr<Point>, int> p(intersections[i], 0);
        points.push_back(p);
    };

    // Cluster labels
    int UNDEFINED = 0;
    int NOISE = -1;
    int C = 0;
    // Determine clusters
    for (uint i = 0; i < points.size(); i++) {
```

```
28      std::pair<std::shared_ptr<Point>, int> * P = &(points[i]);
29      // P is already part of a cluster.
30      if (P->second != UNDEFINED)
31          continue;
32      // Gather points neighboring P
33      std::vector<std::pair<std::shared_ptr<Point>, int>*> Pneighbors;
34      Pneighbors = inRange(points, maxdist, P);
35      // Too small to become a cluster, is instead noise.
36      if (Pneighbors.size() < minpts)
37          P->second = NOISE;
38      // Big enough to be a cluster
39      else {
40          // Cluster label
41          C++;
42          // Make P part of the new cluster.
43          P->second = C;
44          // Expand cluster.
45          uint j = 0;
46          uint Psize = Pneighbors.size();
47          while (j < Psize) {
48              std::pair<std::shared_ptr<Point>, int> * N = Pneighbors[j];
49              // N has already been processed.
50              if (N->second != UNDEFINED && N->second != NOISE) {
51                  j++;
52                  Psize = Pneighbors.size();
53                  continue;
54              }
55              // Add N to this cluster, as well as N's neighbors.
56              else {
57                  N->second = C;
58                  // Gather points neighboring N
59                  std::vector<std::pair<std::shared_ptr<Point>, int>*>
                        Nneighbors;
```

```cpp
                    Nneighbors = inRange(points, maxdist, N);
                    // Extend cluster if dense enough around N
                    if (Nneighbors.size() >= minpts) {
                        for (uint i = 0; i < Pneighbors.size(); i++) {
                        }
                        for (uint i = 0; i < Nneighbors.size(); i++) {
                            if (std::find(Pneighbors.begin(), Pneighbors.end(),
                                Nneighbors[i]) == Pneighbors.end()) {
                                Pneighbors.push_back(Nneighbors[i]);
                            }
                        }
                        //Pneighbors.insert(Pneighbors.end(),
                            Nneighbors.begin(), Nneighbors.end());
                    }
                }
                j++;
                Psize = Pneighbors.size();
            }
        }
    }

    return intersections;
}

/* inRange - Helper for DBSCAN. Returns all Points in a range of maxdist
 * of Point P.
 * @param: std::vector<std::shared_ptr<Point>> intersections - vector of
 *  intersection points
 * @param: double maxdist - minimum distance between points
 * @param: std::shared_ptr<Point> P - point to compare against
 * @return: All neighboring points.
 */
std::vector<std::pair<std::shared_ptr<Point>, int>*> ImpactFinder::inRange(
```

```
91    std::vector<std::pair<std::shared_ptr<Point>, int>> & points,
92    double maxdist, std::pair<std::shared_ptr<Point>, int> * P) {
93
94    std::vector<std::pair<std::shared_ptr<Point>, int>*> neighbors;
95    for (uint i = 0; i < points.size(); i++) {
96       double x = points[i].first->x;
97       double y = points[i].first->y;
98       double dist = sqrt(pow(x - P->first->x, 2) + pow(y - P->first->y,
             2));
99       if (dist <= maxdist)
100         neighbors.push_back(&(points[i]));
101   }
102   return neighbors;
103 }
```

Chapter 4

Results

To analyze the accuracy of this work, the data set created as discussed in 3.1.1 was used. The data for every point on the grid was run through every algorithm combination and parsed by a python script (See Appendix B.2) to generate a heat map. Each heat map shows the difference between the actual and calculated impact points, where a lower number (darker color) indicates greater accuracy. These heat maps are visible in Figures 4.1, 4.2, and 4.3. The goal of these heat maps is to allow for visual comparison of the effectiveness of each algorithm for determining TDOAs and for each method for excluding outliers.

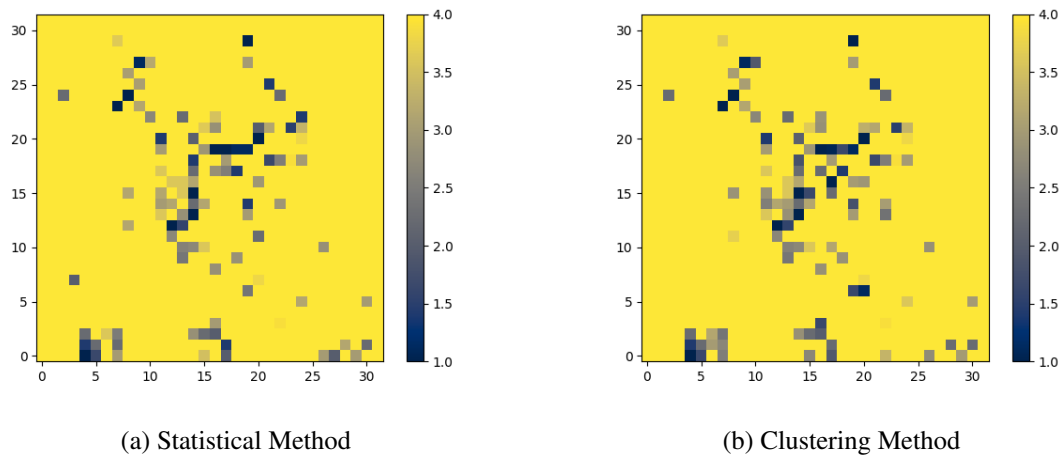(a) Statistical Method   (b) Clustering Method

Figure 4.1: Cross Correlation Heat Maps

Figure 4.1 shows the results of applying the statistical and clustering methods in concordance with the cross correlation algorithm. Immediately it is apparent that this algorithm performed extremely

poorly, only having any degree of success near the grid's center as indicated by the darker points. Cross correlation was not well suited to this problem whatsoever. It generated extremely faulty TDOAs and was very slow, running much longer than the other algorithms by several orders of magnitude. Notably, there is no clear difference between the outlier removal methods.

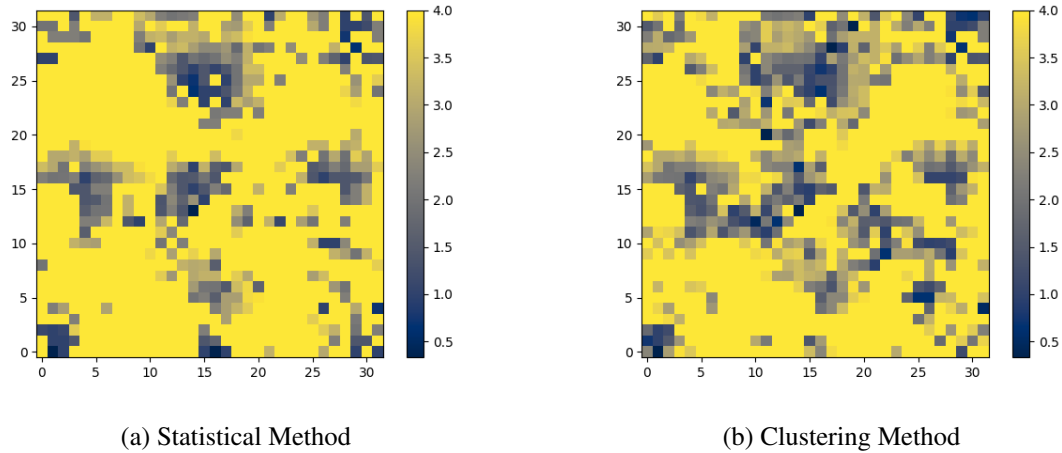(a) Statistical Method   (b) Clustering Method

Figure 4.2: First Hit Heat Maps

Figure 4.2 shows the results of applying the statistical and clustering methods in concordance with the first hit algorithm. This is immediately a big step up in accuracy as compared to the cross correlation algorithm. The first hit algorithm shows the highest accuracy close to the mics and in locations where multiple mics are nearly equidistant to each other from the point of impact such as the center of the grid or along the axes separating the mics into quadrants. Further, the clustering method shows larger regions of accuracy though both have their greatest accuracy in similar regions.

Figure 4.3 shows the results of applying the statistical and clustering methods in concordance with the first sustained hit algorithm. The first sustained hit algorithm shows the greatest accuracy around the center of the grid. Further, it shows a greater degree of accuracy within its region of accuracy than the first hit algorithm. The clustering method shows a clearly larger, denser region of accuracy than the statistical method.
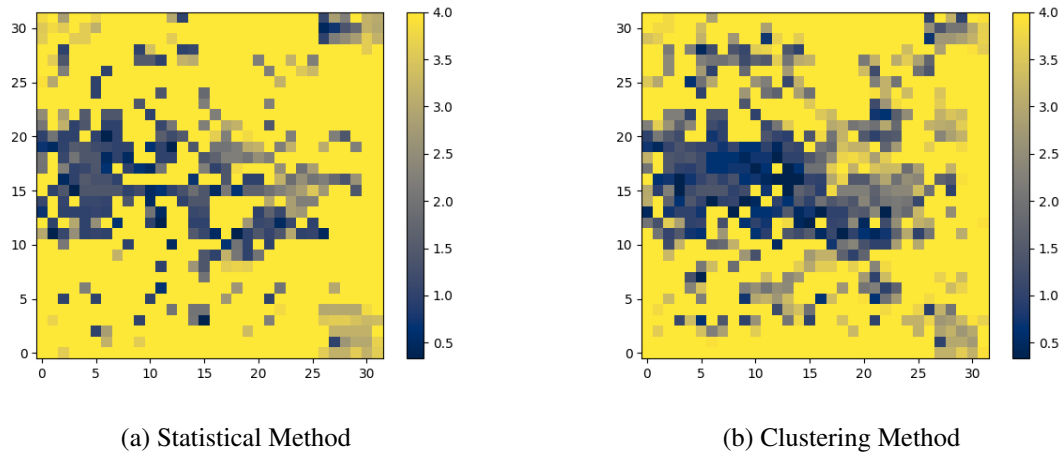
(a) Statistical Method   (b) Clustering Method

Figure 4.3: First Sustained Hit Heat Maps

4.1   Discussion of TDOA Determination Algorithms

Between the three TDOA determination algorithms, first hit and first sustained hit performed far better than cross correlation. First hit had the greatest spread of accuracy, having some degree of accuracy across much of the grid. On the other hand, first sustained hit had the largest contiguous region of high accuracy, concentrated near the grid's center. This makes the first sustained hit algorithm more usable than the first hit algorithm because the mic array can potentially be arranged in such a way that the desired region of accuracy is fully covered.

4.2   Discussion of Outlier Removal Methods

The clustering method outperformed the statistical method across the board, resulting in both higher accuracy and larger regions of accuracy. The clustering method commonly succeeded in finding enough points to construct a cluster in areas that the statistical method determined all points were outliers. Given a list of possible intersection points, the clustering algorithm was more capable of selecting a final, accurate, intersection point.

Chapter 5

Conclusion

This thesis has shown that impacts can be detected to a reasonable degree of accuracy (particularly near the center of the mic setup) using a very simple microphone array. Notably, this was possible using purely binary microphones, despite their limitations. Utilizing the code base built for this thesis, the best configuration would be to use the first sustained hit algorithm paired with the clustering algorithm. This provides the highest level of accuracy of all the tested methods with the highest accuracy near the center of the microphone array. As a result of this higher accuracy near the center of the array.

This thesis offers significant room for improvement and further study. While results were overall reasonable given the restrictions of the equipment used, better estimations can no doubt be obtained by refining the methodology. Notably:

- The statistical method for removing outliers commonly failed to find any viable points of impact. A better statistical method could be used instead such as the 1.5xIQR rule.

- The clustering method always looks for clusters of a static, pre-defined density. It may be possible to determine this density dynamically in order to prevent failure to return a viable impact point.

The greatest limitation of this thesis is that it relies on a specific type of microphone hardware for the various algorithms to function. However, the general methodology is sound and can be extended to microphones that output a more typical smooth waveform. In this case, replacement methods for determining the TOAs and TDOAs would be required. This may produce more accurate results due to the more refined initial data.

References

[1] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu, *A density-based algorithm for discovering clusters a density-based algorithm for discovering clusters in large spatial databases with noise*, KDD'96 Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (1996), 226-231.

[2] Howard W. Hodgkins, *Flash and Sound Ranging*, Journal of the United States Artillery **52** (1920), 51-53.

[3] Caleb Rascon and Ivan Meza, *Localization of sound sources in robotics: A review*, Robotics and Autonomous Systems **96** (2017), 184-210.

[4] Matthew Rhudy, Brian Bucci, Jeffrey Vipperman, Jeffrey Allanach, and Bruce Abraham, *Microphone Array Analysis Methods Using Cross-Correlations*, ASME 2009 International Mechanical Engineering Congress and Exposition **15** (2009), 281-288.

[5] José A. Somolinos, Amable López, Rafael Morales, and Carlos Morón, *A New Self-Calibrated Procedure for Impact Detection and Location on Flat Surfaces*, Sensors **13** (2013), 7104-7120.

[6] John L. Spiesberger, *Geometry of locating sounds from differences in travel time: Isodiachrons*, the Journal of the Acoustical Society of America **116** (2004), 3168-3177.

Appendix A

Primary Code Listing

For a complete code listing, please check out the GitHub link: https://github.com/andresenwc/Senior-Honors-Thesis.

Appendix B

Supplemental Code Listing

B.1 waveplot.py

Listing B.1: plot.py

```python
1  import sys
2  import matplotlib
3  matplotlib.use('Agg')
4  import matplotlib.pyplot as plt
5  import re
6
7  # Parse command line args.
8  xarg = -1
9  yarg = -1
10 try:
11     xarg = sys.argv[1]
12     yarg = sys.argv[2]
13
14     int(xarg)
15     int(yarg)
16 except:
17     print('Usage: python plot.py <x> <y>')
```

```
18    print('\t<x> and <y> are the x and y coordinates of the data to be
          plotted')

19

20  print(xarg)

21  print(yarg)

22

23  # Open and parse file.

24  with open('2D-01-3-samples-per-point.log', 'r') as f:

25

26      line = f.readline();

27      while line:

28          if 'Ready' in line:

29

30              xy = []

31

32              # Get coordinates.

33              line = re.sub('[^0-9]', ' ', line)

34              for i in line.split():

35                  xy.append(i)

36              x = xy[0]

37              y = xy[1]

38

39              if (x != xarg or y != yarg):

40                  continue

41

42              # Skip fluff line

43              line = f.readline()

44

45              # Line 3: Data

46              line = f.readline()

47              if 'init:' in line:

48

49                  # Remove non-digit characters.
```

```python
50            line = re.sub('[^0-9]', ' ', line)
51
52            # List to hold data for each mic.
53            m0data = []
54            m1data = []
55            m2data = []
56            m3data = []
57            # List to hold string data.
58            stringdata = line.split()
59
60            # Remove first (fluff) item from string list.
61            del stringdata[0]
62
63            # Generate data list for each mic
64            for i in range(0, len(stringdata), 2):
65                value = int(stringdata[i])
66                iterations = int(stringdata[i+1])
67
68                m0 = (value >> 0) & 1
69                m1 = (value >> 1) & 1
70                m2 = (value >> 2) & 1
71                m3 = (value >> 3) & 1
72
73                for i in range(0, iterations):
74                    m0data.append(m0)
75                    m1data.append(m1)
76                    m2data.append(m2)
77                    m3data.append(m3)
78
79            # List to hold x-values
80            xvals = []
81            for i in range(0, 1024):
82                xvals.append(i)
```

```
83
84              # Create plot
85              fig, axs = plt.subplots(4)
86              axs[0].plot(xvals, m0data)
87              axs[0].set_title('Mic 0')
88              axs[1].plot(xvals, m1data)
89              axs[1].set_title('Mic 1')
90              axs[2].plot(xvals, m2data)
91              axs[2].set_title('Mic 2')
92              axs[3].plot(xvals, m3data)
93              axs[3].set_title('Mic 3')
94              fig.suptitle('Microphone Data for Point (' + x + ',' + y + ')')
95              plt.tight_layout(rect=[0, 0.03, 1, 0.95])
96              plt.savefig('plots/(' + x + ',' + y + ').png')
97
98              break
99
100         line = f.readline()
```

B.2   heatmap.py

Listing B.2: heatmap.py

```
1  from __future__ import division
2
3  import numpy as np
4  import matplotlib
5  matplotlib.use('Agg')
6  import matplotlib.pyplot as plt
7  import numpy as np
8  import math
9  import re
10
11 # Open out.txt for reading
```

```python
with open('out.txt', 'r') as f:

    # 2d list to store xvals, yvals, and counts
    xvals = [[0 for x in range(32)] for y in range(32)]
    yvals = [[0 for x in range(32)] for y in range(32)]
    count = [[0 for x in range(32)] for y in range(32)]

    line = f.readline();
    while line:

        xy = map(int, re.findall(r'\d+', line))

        if (len(xy) < 4):
            line = f.readline();
            continue

        x = xy[0]
        y = xy[1]
        xval = xy[2]
        yval = xy[3]

        xvals[x][y] += xval
        yvals[x][y] += yval
        count[x][y] += 1

        line = f.readline();

    # Calculate average x and y values
    for i in range(32):
        for j in range(32):
            if (count[i][j] != 0):
                xvals[i][j] = xvals[i][j]/count[i][j]
                yvals[i][j] = yvals[i][j]/count[i][j]
```

```python
45
46    # 2d list to store average distances
47    distances = [[0 for x in range(32)] for y in range(32)]
48
49    # Calculate average distances
50    for i in range(32):
51        for j in range(32):
52            if (count[i][j] != 0):
53                x = i
54                y = j
55                xval = xvals[i][j]
56                yval = yvals[i][j]
57                distances[i][j] = math.sqrt((xval-x)**2 + (yval-y)**2)
58
59    # Find max distance
60    max = 0
61    for i in range(32):
62        for j in range(32):
63            if (count[i][j] != 0):
64                if (distances[i][j] > max):
65                    max = distances[i][j]
66
67    for i in range(32):
68        for j in range(32):
69            if (count[i][j] != 0):
70                #distances[i][j] = distances[i][j] / 8#(max)
71                if (distances[i][j] > 4):
72                    distances[i][j] = 4
73            else:
74                distances[i][j] = 1
75
76    arr = np.array(distances)
77    plt.imshow(arr, origin='lower', cmap='cividis')
```

```
78    plt.colorbar()

79    plt.savefig('plots/heatmap.png')
```