# Final Report

Anirudh Indraganti

March 2025

## 1 Introduction

This is the final report of the research conducted with Professor Siddharth Vishwanath in the Winter quarter of 2025. The work explores the field of topological data analysis and writing algorithms for better results and efficiency. The work done first began with understanding the basics of the field of topology with a focus on gaining a general understanding of concepts related to persistent homology and generation of persistence diagrams. It then progressed to Julia basics and how to use it for persistent homology. From here, implementation and testing of a classical TDA method was done and tested against theoretically better ones done in the RobustTDA.jl package created by Professor Vishwanath. Once this testing was finished, parallel approaches were explored and implemented to improve the pure computational time of the RobustTDA.jl algorithms.

### 1.1 General Overview

The goal of topological data analysis to find major structures that exist within the shape of data. This information is computed by persistence diagrams. A persistence diagram is the $2D$ graphical representation of the persistent homology of a dataset. It tracks when topological features are born and when they die. Significant topological features have a long lifetime, which is calculated as $d_i - b_i$ where $d_i$ is the death of a topological feature and $b_i$ is its birth. Below, in Figure 1, we can see an example of creating a topological structure on a set of seven points. The example below uses a Cech complex, where the structure of the data is based on the intersection of radii around each data point, to create this topological structure.

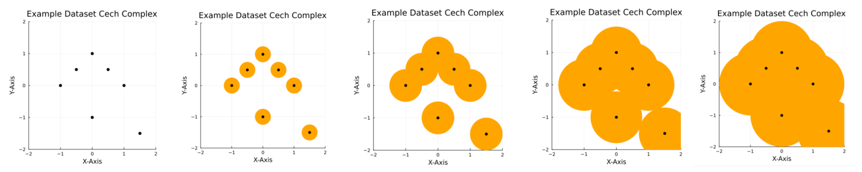

Figure 1: Example Dataset Persistent Homology

In the image above, the radius of the circles around each data point grow in the manner of $[0, 2, 0.5]$. Considering we have data in $R^2$, we can only have features in $H0$ and $H1$. These features correspond to connected components and holes. When looking at the image above, we can see that we begin with 7 connected components (one for each point), which then becomes 3 when $r = 1$. These connected components then merge a single one once $r > 1.5$. As for the holes, we see that the presence of holes does not show up until $r = 1.5$, where 2 holes are created, which then disappear once $r = 2$.

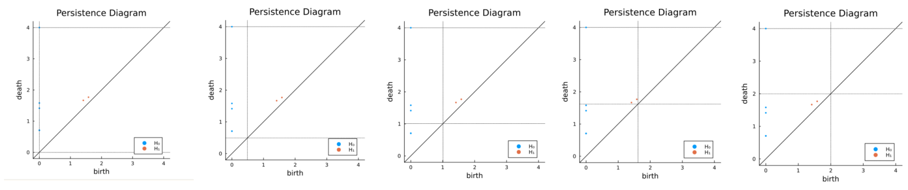

Figure 2: Example Dataset Persistece Diagram

In the image above, the persistence diagram of the dataset from Figure 1 is shown at the same radii. The information we gained at simply glancing at the data in Figure 1 is generated and displayed in a more concise and precise manner in Figure 2.

From the description above, it is easily noted that the statements were all based on viewing a visualization rather than any true computation. A pipeline to compute this information reliably is as follows:

1. Choose a distance function
2. Create a topological structure for the set of points
3. Create a persistence diagram on this structure

## 1.2 Distance Functions

Choosing a distance function is the first step in the pipeline. This step directly relates to how the topological structure is formed on the set of points. The reason for this is because the "growing circles" illustration from above are induced by the sub-level sets of the distance function.

There are two major factors that must be analyzed when choosing a distance function: robustness to noise and computational speed. Real world data tends to contain significant noise. Therefore, choosing a distance function that is able to be resistant to noise is crucial for examining important features and how they persist. However, data in the real world can also be very large. This means that we need a distance function that is able to perform calculations at high speeds. Finding a distance function that is able to achieve both of these is critical to the pipeline. Three main functions that will be focused on in this paper are the nearest neighbors (NN where k=1), distance-to-measure (DTM), and median-of-means (MoM).

Different distance functions provide different benefits. Starting with the nearest neighbors distance function, construcction of a KD-Tree must take place on the set of points. This takes $O(n\log(n))$ time to construct the tree. Then, finding the nearest neighbor of a single point takes $O(\log(n))$ time. Doing this process for $n$ points, the time complexity overall for the nearest neighbors algorithm comes to $O(n\log(n)) + O(n\log(n)) = O(n\log(n))$. However, the nearest-neighbors function is not robust to noise. This makes identifying the major topological features in a dataset difficult, making it less useful in practice.

$$NN_1(x) = \text{argmin}_{y \in X \setminus \{x\}} d(x,y) \text{ where } d(x,y) = \sqrt{\sum_{i=1}^{d}(x_i - y_i)^2}$$

The DTM function is a distance function much more robust to noise than the regular nearest-neighbors algorithm. However, a time complexity trade-off occurs here. First, a KD-Tree must be built, which takes $O(n\log(n))$ time. Then, finding $m$ nearest neighbors for a single evaluation point takes $m\log(n)$ time where $m$ is the number of nearest neighbors to consider. Doing this for $n$ data points leads to a time complexity of $O(n\log(n))+O(mn\log(n)) = O(mn\log(n))$. Though there is an increase in time complexity from the first, the robustness to noise of the DTM function makes it more preferable in practice.

$$DTM_m(x) = \left(\frac{1}{m}\sum_{i=1}^{m} d(x, x_{(i)})^r\right)^{\frac{1}{r}}$$

The median-of-means function is again a distance function robust to noise. The time complexity analysis begins with partitioning the dataset into $Q$ folds. This is a $O(n)$ time operation. Then, construct KD-Trees for each partition, which comes out be a $O(n\log(\frac{n}{Q}))$ time operation. Searching the KD-Tree for $m$ nearest-neighbors for each partition comes out to be a $O(Qnm\log(\frac{n}{Q}))$ time operation. Finding the mean of each partition's results comes to be a

$O(Qm)$ operation. Taking the $Q$ means and finding their median comes to be a $O(Qlog(Q))$ operation. Together, the time complexity is $O(n) + O(n \log(\frac{n}{Q})) + O(Qnm \log(\frac{n}{Q})) + O(Qm) + O(Qlog(Q)) = O(nQm \log(\frac{n}{Q}))$. Introducing parallelization, the steps of constructing the KD-Trees and querying nearest-neighbors can be decreased. In a truly parallel environment, the construction of the KD-Tree becomes $O(\frac{n}{Q} \log(\frac{n}{Q}))$ and the querying becomes $O(nm \log(\frac{n}{Q}))$. We save on computational time by a factor of $Q$. The implementation and details of this is the main purpose of this paper.

$$\mu_i = \frac{Q}{n} \sum_{x \in S_i} x \text{ for } i = 1, \ldots, \text{Q where } S_i \text{ is a partition of the dataset}$$

$$\text{MoM} = \text{median}(\mu_1, \mu_2, \ \ldots, \ \mu_Q)$$

The image below shows these functions in action with an example set of data of 500 points in $R^2$. The major topological feature that is desired to be clearly seen is the circle. The dataset samples a random 350 points from a circle of $r = 1$ and has a center at the origin. The rest of the points are uniform noise within the range of the circle's points. For the DTM function, $k = 4$ and $r = 2$ were considered while for MoM, $Q = 10$. Keep in mind these images do change significantly with different parameters. Below is merely an example of these distance functions in action.

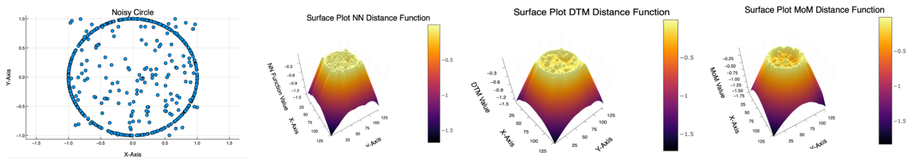

Figure 3: Distance Functions With Noisy Circle Dataset

## 1.3 Creating Structure On Data

Now that distance functions have been introduced, the actual creation of the topological structure is the next step in the pipeline. One of the major ways that this is done is through the creation of a grid and computing the distance function values on the set of grid points. These values are then used to create a cubical complex, a type of simplicial complex that creates its simplicies based on cubes rather than triangles such as in other complexes like the Rips or Cech. From here, the persistence diagram is computed, giving us the information we require about the homological features of the data.

The grid used is predefined and usually a square matrix with a designated resolution size, similar to an image on a computer. As a result, the method above assumes uniform importance for all points in the predefined grid when this is usually not the case. Due to the nature of the grid, it also leads to an exponential increase in the number of cells when finer details are necessary.

The solution to this major issue that is provided in [1] is a weighted CW (Cellular Weak) complex. This method places weights on points that contain more important homological information. Therefore, it allows for resolutions to be adaptive rather than predefined, reducing pre-processing time. With no grid refinement, this also means the construction of a filtration is less expensive, leading to faster computations. With a weighted complex, it still takes in a distance function as input. However, instead of the circles, as seen in Figure 1, being uniform in radius, are different depending on their importance of information.

Weights can be assigned in different ways where one single way is not standard. These weights can represent local density estimation, a predefined function, or some other application-based criteria.

## 1.4 Persistence Diagrams

The construction of a persistence diagram is based on a filtration and how this filtration changes under different conditions. As mentioned above, this condition can be based on whatever type of filtration is being constructed. For the example above, a Cech complex was created. The persistence diagrams above tracked the birth and death of important homological features as the condition of the radius of each circle increased. The ability to track these features reliably is where the choice of the distance function comes into play. Below are the persistence diagrams of the distance functions in Figure 3.

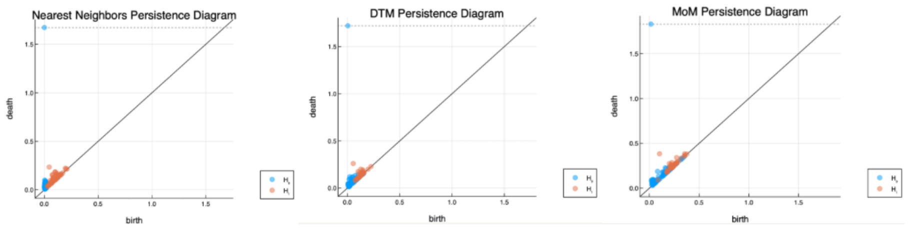

Figure 4: Persistence Diagrams on Distance Functions

The goal was to detect the circle that was present in the dataset. This is a circle, which is an $H1$ feature, meaning the interest is in the red points on the graph. In incremental improvement, it is shown that this circle persisted the least with NN, next with DTM, and most with MoM. The difference is slight, but noticeable. This is mererly an example, but the general principle is to allow major features in the data to persist regardless of noise. As mentioned, DTM and MoM did a better job of that in comparison to NN.

The specific goal of the work done in this paper resides in the analysis of computational runtimes of different distance functions and how to potentially improve upon them. However, the work done prior to this was listed above, creating a general understanding for the purpose of the work being done in the Methodology and subsequent Results sections.

## 2 Methodology

The first experiments done were to understand the computational cost of increasing the resolution size of a grid in the classical TDA approach. The anticipated result was some form of exponential increase as the resolution increases. In this experiment, the distance function used was NN. This function, however, can be replaced by any other functions and yield similar results since it is a constant in the experiment framework. The data size used was 100,000 data points. The grid sizes ($n \times n$) increased as follows: [16, 32, 64].

Theoretical time complexities were listed in the Introduction for the three main distance functions. To put this to the test, experiments were run on these distance functions. Testing was done on the three distance functions listed above: NN, DTM, and MoM. The MoM function provided in the RobustTDA.jl package already implements a multi-threaded solution. To improve upon this implementation, a truly parallel approach was implemented and tested here as well. This parallel approach used the Distributed.jl package, a widely used package for developing parallel code in Julia.

For the experiments, dataset sizes of 1,000, 10,000, 25,000, 50,000, and 100,000 were tested. Each distance function's runtimes were measured using the BenchmarkTools.jl package. For the DTM function, 10% of the total number of data points was used as the amount of nearest neighbors $m$. For the multi-threaded and parallel MoM functions, $Q = 4$, which was the number of available workers on the machine that was used, a Mac Mini while $m = 1$ for the nearest-neighbors to consider. This means the theoretical runtimes can be simplified to gain a better understanding of our expected outcome. The NN function stays the same, having a time complexity of $O(n\log(n))$. The DTM function becomes $O(0.1n^2\log(n)$ since we are using ten percent of total data points as the amount of nearest neighbors. The Multi-Threaded MoM Function becomes $O(4n\log(\frac{n}{4}))$ and the Parallel implementation becomes $O(n\log(\frac{n}{4}))$. Though this would make us assume the Parallel MoM function would perform the best, we forget there are many constants that were removed from this calculation, which outscale the nearest-neighbors function. Therefore, the results we expect to see from worst-to-best runtimes are DTM, Multi-Threaded MoM, Parallel MoM, then Nearest Neighbors.

## 3 Results

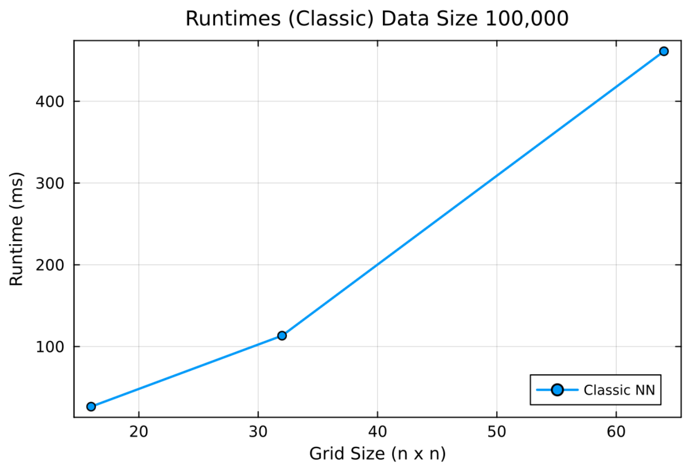

Figure 5: Grid Scaling Runtimes

In the image, there is a clear exponential increase as the grid size increases. This supports the expected results mentioned in the previous section.

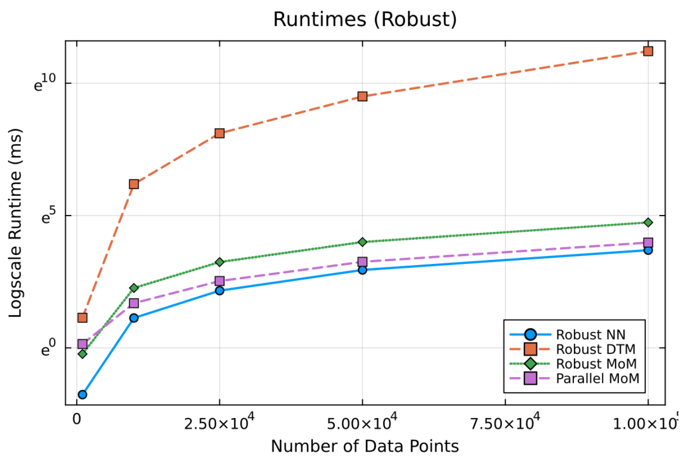

Figure 6: Distance Function Runtimes

The results came back just as expected. Though only a single graph, there is much to unpack here.

The implementation of the parallel MoM function was successful and has a better runtime than its multi-threading counterpart as the dataset size grows larger. However, we see at the beginning, where the dataset size was 1,000 points, that it did not outperform. This was a semi-anticipated result. In general, parallel execution reduces the pure computational time of any algorithm. However, the the input size must be so that it out-scales the overhead of doing parallel computing. The break-through limit seems to be somewhere between 1,000 and 10,000 data points. It also is seen that it slightly underperforms the NN function. This also was anticipated due to the multitude of constants that exist in the computation of MoM along with the parallel overhead. It also noted that DTM performed the worst by far. This is just as expected and we see that it is truly scaling by more than a linear or overhead factor, as calculated in the simplified time complexities.

## References

[1] Arxiv. (n.d.). `https://arxiv.org/pdf/2206.01795.pdf`

[2] Hal. (n.d.). `https://hal.science/hal-02093445/file/DTM-filtrations_SoCG.pdf`