Universiteit Utrecht

Real-World Complexity Of Central Trajectories

Roald Melssen

ICA-4298977

Game and Media Technology Master Thesis

supervised by:

Maarten Löffler

Mees van de Kerkhof

April 5, 2020

Abstract

When analyzing a set of trajectories, usually a very large amount of data is available. This dataset may be so large it makes full analysis impossible. A solution is to cluster the dataset into smaller subsets, and then to generate a representative trajectory for each subset, which represents the unique characteristics of the trajectories in the set. One way to do this is an algorithm called Central Trajectories, which generates a trajectory consisting of parts of the input trajectories and which is central relative to the trajectories in the set. This results in a trajectory that is relatively close to all other trajectories at all times. I analyze the complexity of the output of the algorithm, using multiple different methods and a large variety of real-world and synthetic datasets. The two main research questions are: "*What is the real-world complexity of Central Trajectories?*" and "*What is the effect of path-simplification algorithms on the complexity of Central Trajectories?*" The results suggest answer to the first question is that the complexity of the output is linear in the input, but the characteristics of the linear relationship between input and output vary depending on the characteristics of the dataset, in addition to the parameter epsilon (ϵ) of the Central Trajectory algorithm, which determines the maximum size of a discontinuity. Additionally, simplifying the output of Central Trajectories can greatly reduce the amount of vertices, since many redundant vertices are removed in the process.

Contents

1 Introduction

1.1 Motivation

My thesis focuses on the study of trajectories. A trajectory in this context means: the path an object follows when it moves. Nowadays, a large amount of trajectory data is available, generated using for example GPS trackers. By determining its position at regular moments in time, a GPS tracker can make a trajectory of an entity's movement over time. This means a large variety of interesting trajectories can be generated without much effort, by tracking, for example, the movement of vehicles, animals and humans.

1.1.1 Vehicles

By tracking cars, a large amount of interesting data becomes available, which is especially useful for branches of science such as traffic congestion analysis, urban planning, environmental research, etc.

Many countries use a Vessel Monitoring System (VMS) to track the activity (including position) of fishing vessels in real time. The purpose of this system is to ensure vessels abide by the rules and overfishing does not occur, which would harm the local marine ecosystem.

Finally, airplanes are tracked in real time as well, since it is vital to know the location of airplanes in real time to know whether or not a potential disaster is about to occur, to enable timely prevention (e.g. diverting two airplanes on a collision course to prevent a mid-air collision). But these trajectories are useful in non-real time scenarios as well, for example for environmental research (to give an example: the delay of ongoing expansion of the Lelystad Airport).

1.1.2 Animals

Tracking (wild) animals over time gives interesting insights into their behavior, since the trajectory of an animal reveals a lot about their routines. For example, a stationary trajectory indicates an animal is resting or sleeping, but a trajectory that traverses large distances may indicate the animal is scavenging for new territories, or maybe it is searching for food or water.

1.1.3 Humans

Humans can also be tracked with GPS trackers, and this is already happening with smartphones: for example, Android smartphones by default create a trajectory of everywhere you go. (I will leave the privacy implications of this aside, but it is worth a research on its own.) But there are also datasets of human movement that are publicly available, for example OpenPFLOW[1], which consists of (anonymized) trajectory data of pedestrians in Tokyo, Japan. This data can be very valuable, similarly to vehicle data, for branches of science such as urban planning, evacuation/riot handling, crowd simulation and sociology.

1.1.4 Hurricanes

It is also possible to track objects without using GPS trackers. For example, hurricanes are tracked in real time by satellites that analyze the movement of clouds, deducing wind information from this analysis, and then deducing the location of a hurricane from this wind information. This enables knowing not only the past and present location of a hurricane, but also predicting its future position. This information can be potentially life-saving, since it enables timely evacuation of citizens that are located near the hurricane's future predicted trajectory.

1.2 Mathematical context

In a formal, mathematical context, a trajectory is defined as a sequential list of points in space. I will focus on two-dimensional space here, but some algorithms extend naturally to three dimensions and beyond. Additionally, every point contains a timestamp. It may be tempting to model time by simply increasing the dimensionality of the points, and using time as the last dimension, but this does not make sense, since time is defined in seconds, yet location is defined in meters. Many spatial measures, such as Euclidean distance, only make sense when all dimensions are defined in the same units.

Trajectory clustering is the gathering of trajectories into groups, in which trajectories have similar characteristics. To cluster trajectories, various algorithms are available: see Section 2.1 and Section 3.4 for a sample of relevant algorithms.

After applying clustering, groups of trajectories that belong together are revealed: either because they follow similar paths, or because they share common sub-parts of the path. Then, for every cluster, it

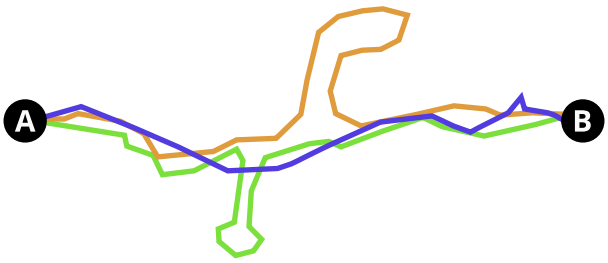


Figure 1: A cluster of three different trajectories going from A to B. When picking a representative trajectory, picking a random one will give the wrong result in this scenario, since none of the trajectories capture all three unique features at once (looping up, going straight, and looping down). A good representative trajectory would first follow the blue trajectory, then follow the green trajectory by looping down, then follow the orange trajectory by looping up, and then continue to the end. A summarization algorithm may find a trajectory like this.

is usually desirable to create a representative trajectory which captures the essence of all trajectories in the cluster. Generally speaking, a random trajectory is picked from the cluster for this purpose. However, this is not always a good solution, since the unique features of a cluster might, by chance, not be present in the selected trajectory. For a visual demonstration, see Figure 1.

A better algorithm would assemble a new representative trajectory, retaining the unique characteristics of the trajectories in the cluster. One of the ways to do this is to use an algorithm called Central Trajectories [2]. The key idea is to build a new trajectory that consists of sub-parts of trajectories in the cluster, allowing small discontinuous jumps between different trajectories if required. Note that the paper introduces only a theoretical algorithm, without any implementation. Thus, the paper only made claims about the theoretical complexity of the algorithm, not the real-world complexity.

The research of this paper consists of two research questions, aiming to discover the performance of the algorithm in practice. The first one is: "*What is the real-world complexity of Central Trajectories?*". The second one is: "*What is the effect of path-simplification algorithms on the complexity of Central Trajectories?*". More elaboration can be found in Section 1.3. To answer them, the Central Trajectories algorithm has been implemented, and its real-world performance has been measured using various experiments, described in Section 4.3. The results can be found in Section 5, and the discussion and conclusion based on these results can be found in Section 6 and 7, respectively.

1.3 Research questions

This section will explain all research questions for the research project.

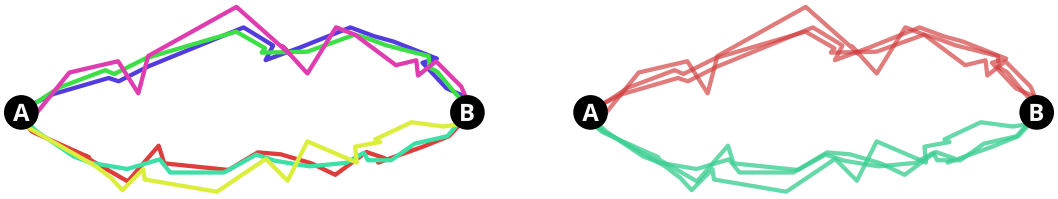
1.3.1 Question 1

The first research question is:

> *What is the real-world complexity of Central Trajectories?*

The theoretical worst case complexity is superquadratic: $O(nm^{5/2})$, where n is the amount of vertices per trajectory and m is the amount of trajectories. This upper bound is reached by constructing a pathological zigzagging pattern, as described by van Kreveld et al. [2]. But how often does this occur in real world data? Its complexity is largely defined by the size of the input and the amount of intersections between trajectories.

Hypothesis I expect the real-world complexity to be much lower than the theoretical bound of $O(nm^{5/2})$, since the zigzagging pattern mentioned in the paper, which triggers the upper bound, is very artificial.



(a) Before applying clustering.

(b) After applying clustering.

Figure 2: Six different trajectories going from A to B. After clustering, groups of trajectories that travel closely together are revealed.

1.3.2 Question 2

The second research question is:

> *What is the effect of path-simplification algorithms on the complexity of Central Trajectories?*

That means, if the complexity of the dataset happens to increase after running the Central Trajectories algorithm on it, how much of this increased complexity can be mitigated by running path-simplification algorithms before or after the Central Trajectories algorithm?

Hypothesis Path simplification will greatly reduce complexity of the generated trajectories. How effective this reduction is depends on the data: for example, flight data might reduce more in complexity than zoological data, since they are mostly straight lines.

2 Related work

Various algorithms have been introduced and implemented to analyze trajectories in various ways: this section will show a few examples. This section will be similar to the summary of trajectory algorithm types found in the thesis by Staals [3] and the paper by Konzack [4]. The most important subsections will be the ones dedicated to trajectory clustering and the generation of representative trajectories (*summarization*). Clustering is important to any research that implements summarization, since summarization usually assumes the input trajectories are already somewhat clustered. That means, a clustering algorithm must be used before running a summarization algorithm. However, trajectory visualization and classification will be considered as well – as we will see, they are tangentially related.

2.1 Clustering

Trajectory clustering is the gathering of trajectories into groups, in which trajectories are similar according to some kind of distance and/or time metric. For a visual example, see Figure 2.

Buchin et al. [5] introduce the notion of a cluster (or *group*) and present algorithms for computing all groups over a set of trajectories. The goal is to analyze the trajectories in such a way that we can find groups of trajectories, which can change over time. For example, when many trajectories converge, a new group is formed; when many trajectories diverge, a group is disbanded. To formalize the definition of a group, three metrics are used:

- A distance metric for the maximum mutual distance between the moving entities in a group, which is limited by a parameter ϵ.
- A time metric for the minimal duration of a group, called δ.

- A number for the minimum amount of members of a group, called m.

That means, when many entities move together in such a way their mutual distance is always at most ϵ, the group lasts for at least δ time units, and there are at least m entities moving together, then the moving entities form a group.

Van Kreveld et al. [6] build further upon this work, and introduce a more refined model that corresponds better to human intuition, especially in dense crowds. For example, in the old model, consider a large crowd of moving entities. Two entities that move together in the opposite direction of the entire crowd will be grouped together with the others in the crowd. But this does not make sense, since the two entities move in distinctly different directions compared to the rest of the crowd. So, although the two entities themselves should be together in a group, the rest of the crowd should not be part of that group.

Van Goethem et al. [7] also build upon the foundations laid by Buchin et al. [5] by introducing a specialized data structure that can be used to experiment with different values of ϵ, δ and m in real-time, to see which values most accurately capture the groups we are looking for.

A different approach, used by Lee et al. [8], is to partition all trajectories into line segments, and then apply clustering on these line segments, instead of on the trajectories themselves. The line segments are grouped based on certain distance metrics between them. The advantage of this approach is that it allows the algorithm to cluster based on small parts shared among different trajectories, even though the trajectories themselves might globally look totally different. The approach is density-based, which means clusters are formed where the density of points is the highest.

Andrienko et al. [9] focus on the domain of aviation trajectories. They introduce a clustering approach of trajectories using density-based methods such as DBSCAN, according to the similarity of the followed routes. This can be quantified using a specialized distance function, which, when given two trajectories, calculates the distance between them. An unique characteristic of their approach is the fact that they use a so-called *relevance mask*, which filters out irrelevant points such that they no longer have an effect on the distance function. An example would be to filter outliers, or in the domain of aviation, filtering out takeoff/landing parts of the trajectory. Trajectories that are too far away from every other trajectory are assigned to a "noise" cluster, which means they do not belong anywhere and are probably outliers. Note that the noise cluster is a characteristic of the DBSCAN algorithm, and DBSCAN was originally only made for clustering of points. Andrienko et al. [10] use a similar approach.

Gariel et al. [11] also focus on the domain of aviation trajectories. The goal is to identify common flight routes, and to detect whether or not airplanes nicely follow their predetermined flight routes. This is important, because if airplanes deviate too much from their routes, more attention is required from air traffic controllers to ensure a safe separation between aircraft. Their main contribution is the observation that airplane trajectories consist mostly of straight lines, with a few important waypoints near airports or other important locations. These points are called *turning points*, since they are points at which the airplane makes a turn, and usually the trajectory followed between two turning points is a straight line. This implies airplane trajectories can be simplified tremendously by removing all points from the trajectories that are not waypoints, without major loss of information. Then, after reduction, the trajectories can be clustered using the LCSS (Longest Common Subsequence) algorithm.

Precursory work by Buchin et al. [12] describes a clustering method with a clear purpose: to detect commuting patterns in traffic GPS data. They use the Fréchet distance as distance metric between two trajectories. The Fréchet distance has the useful property that it is invariant under differences in speed, so the two trajectories do not necessarily have to be aligned in the time domain. This enables commuting patterns to be discovered along groups of commuters, even when they are not traveling at the same speed. Inside the so called *free space diagram,* which is closely related to Fréchet distance, a distinct pattern can be seen whenever multiple trajectories (or multiple parts of a big trajectory) follow a roughly identical path in space; by detecting this pattern, clusters of trajectories can be discovered.

Gaffney and Smyth [13] use a previously unseen approach: they use a *finite mixture model*, which is a probability density function (PDF) consisting of a linear combination of smaller PDFs, also known as *regression model components*. These components represent the clusters, and they are learned using an expectation-maximization algorithm. Generally, the regression model components are modeled using

a simple Gaussian distribution – that means, the expectation-maximization algorithm finds the most likely mean μ and variance σ for every regression model component in an unsupervised manner. The authors went for a probabilistic clustering approach, since they assume the input data is noisy, so there is a certain amount of uncertainty involved. The algorithm does not require all trajectories to have an equal amount of vertices, unlike other clustering methods such as K-means. The drawback, however, is that the amount of clusters, K, is fixed and must be determined by hand. Additionally, since the algorithm is probabilistic, it might not always be clear to which cluster a certain trajectory belongs: for instance, the probability that a trajectory T belongs to cluster i might be equal to the probability that it belongs to cluster j.

The paper by Gaffney et al. [14] builds upon the previous approach, and has a clear real-world application. Given a historic dataset of air pressure levels across the globe, collected over the span of multiple decades, the algorithm works by first discovering the cyclones themselves, together with their (future) trajectories: this works by detecting small areas of very low pressure. Then, given these cyclone trajectories, probabilistic clustering is used to classify the cyclones into three possible clusters, based on their movement direction: S-N (south to north), SW-NE (south west to north east) and W-E (west to east). Note that there is no category for west-moving cyclones: because of the rotation of the earth, cyclones generally do not travel to the west. The algorithm uses a finite mixture model, as seen before. The authors found that quadratic polynomials provide the best fit among all possible regression models. Like the previous approach, the algorithm does not require all trajectories to have an equal amount of vertices, but does require the amount of clusters to determined manually beforehand.

Fu et al. [15] introduce a new *spectral clustering* approach, for a clear real-world purpose. They aim to cluster trajectories of cars, extracted from security camera footage, into frequently travelled paths. The algorithms works by constructing a similarity matrix between the trajectories, based on their mutual distances, and then finding an optimal partition of the graph induced by the similarity matrix. Additionally, their approach enables real-time anomaly detection, which reveals cars which are travelling too fast or in the wrong direction.

Lin et al. [16] aim to introduce a new method to approximate *time series*, which is a generalization of trajectories, although the paper focuses on one-dimensional trajectories. Their goal is to apply trajectory algorithms, such as clustering and anomaly detection, to very large datasets which do not fit in memory. To deal with such large datasets, they aim to approximate the trajectories using a method named *SAX (Symbolic Aggregate approXimation)*, and then running the trajectory algorithms on this approximated version of the original. The dataset is approximated symbolically, which means it is converted into a list of discrete *letters* (symbols) to form a *word*. It is possible to experiment with the alphabet size, which is the amount of possible letters. The obvious drawback of the approach is that it focuses on one-dimensional time series, which means it does not work on traditional trajectories gathered using e.g. GPS data – the authors acknowledge that an extension to higher-dimensional time series would be useful. Finally, notice the algorithm is also capable of classification and summarization.

Kim and Mémoli [17] introduce the concept of a *formigram*, which is a slightly different version of a *dendrogram*. A dendrogram is a way to visualize an hierarchical clustering of entities. However, for clustering trajectories, it is not suitable, because it assumes once groups are formed, they are never disbanded again – an unrealistic assumption. To deal with this, the authors modify the dendrogram slightly to allow for groups that form and disband over time – they call this modified version a formigram. Additionally, they introduce a distance metric to determine the distance between two formigrams.

Zhang et al. [18] aim to compare different clustering algorithms using multiple experiments with real-world data, perturbed with random noise, while also randomly removing points from the trajectories. The trajectories are extracted from surveillance camera footage, as seen before in the paper by Fu et al. [15], except this time pedestrians are tracked instead of cars. The clustering methods tested include Euclidean, PCA+Euclidean (Principal Component Analysis), Hausdorff, HMM (Hidden Markov Model), LCSS (Longest Common Subsequence), and DTW (Dynamic Time Warping). A new metric called the CCR (Correct Clustering Rate) is introduced, which measures intra-cluster similarity and inter-cluster disparity. A high CCR means trajectories within every cluster are mutually alike, while clusters from different trajectories are sufficiently different: this is an indication that the clustering algorithm is performing well. The computation time is also measured. The results are as follows.

Euclidean, PCA+Euclidean, LCSS and DTW all produce roughly equal results in terms of CCR, although PCA+Euclidean is much less computationally expensive to compute compared to all other algorithms. When noise is introduced, the Euclidean algorithm quickly fails to cluster trajectories correctly. Generally, Hausdorff and HMM produce the worst results, even though they are relatively expensive to compute.

2.1.1 Trajectory distance metrics

As mentioned before, to cluster trajectories, some kind of distance and/or time metric is needed, which calculates how (dis)similar two trajectories are. The metric can be anything that results in a useful clustering of trajectories.

The simplest metric is the Euclidean distance. In the discrete case, given two trajectories A and B with their corresponding points $\boldsymbol{a_i}$ and $\boldsymbol{b_j}$ with $1 \leq i \leq n$ and $1 \leq j \leq n$, the Euclidean distance between two trajectories is defined as the average (or maximum) of the Euclidean distances between all corresponding points of the two trajectories [18], mathematically formulated as:

$$\frac{1}{n} \sum_{i=1}^{n} \sqrt{(\boldsymbol{a_i} - \boldsymbol{b_i})^2}$$

The obvious drawback is that both trajectories need to have exactly n points. In the continuous form, the distance becomes:

$$\frac{1}{l} \int_0^l \sqrt{(\boldsymbol{a}(t) - \boldsymbol{b}(t))^2} dt$$

And the condition becomes: they should both span exactly the same time interval (from 0 to l). Note that trajectories that follow identical paths in the opposite direction are considered to be highly dissimilar, according to the Euclidean distance – depending on the application, this might be undesired.

There are also other techniques like Dynamic Time Warping, which aim to match two trajectories by warping their respective time series – this allows distance to be measured between two sequences which vary in time and speed.

Finally, other examples include the Fréchet distance, the Hausdorff distance and LCSS (Longest Common Subsequence). The LCSS is a variation of the edit distance. The basic idea is to match two sequences by allowing them to stretch, without changing the order of the elements, yet allowing some elements to remain unmatched[19].

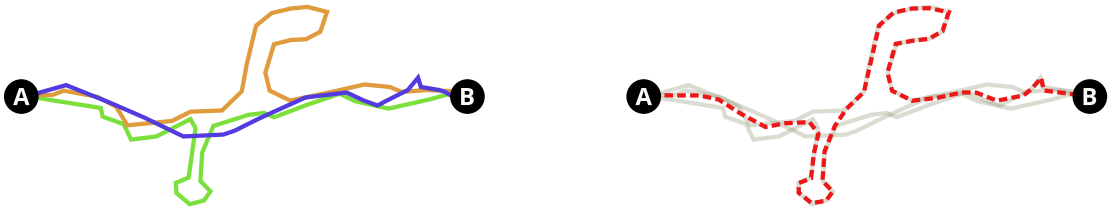
Wiratma et al. [20] introduce various metrics that can be used not just for comparing two trajectories, but also for comparing two groups of trajectories. A few examples include group similarity (average distance to another group), group closeness (the average distance to the nearest other group at each time), and group centrality (the average distance to the central position of the other groups).

2.2 Summarization

As mentioned before, after applying trajectory clustering, for every cluster we want to find a representative trajectory that captures the essence of all trajectories in the cluster. We do not want to pick a random trajectory from the cluster as representative, since the randomly picked trajectory might not contain all defining features of the group. The process of generating a representative trajectory is called *summarization*. For a visual example, see Figure 3.

van Kreveld et al. [2] introduce the concept of a *Central Trajectory*. A Central Trajectory is a *trajectoid,* a trajectory which consists of sub-parts of the input trajectories, which stays as central as possible according to some centrality measure. In the paper, the centrality measure used is the distance to the furthest point across all trajectories at a given time t.

van Kreveld et al. [2] also introduce a dichotomy between *time-dependence* and *time-independence*. Time-dependence means that time is relevant; time-independence means that it is not, and only the



(a) Before summarization.

(b) After summarization. The dotted red line is a possible representative trajectory.

Figure 3: Visual demonstration of the summarization of a cluster of three trajectories.

geometry or physical shape of the trajectory matters. The algorithm described in this paper is time-dependent, and most algorithms that deal with trajectories are.

A notable exception is the paper by Buchin et al. [21], which introduces the concept of a *median trajectory*. Like a Central Trajectory, a median trajectory is also a trajectoid, but there are three major differences. First off, the algorithm does not take time information into account, which makes it a time-independent algorithm. Secondly, the algorithm uses the median as centrality measure. Finally, the algorithm also supports the preservation of *homotopy*. This means it can identify *poles*, which are important locations in the paths the clusters take, and ensure the generated representative trajectory always curves along these poles in the same way as the input trajectories. This prevents the generated trajectory from skipping important turns, or missing essential landmarks.

The algorithm described by Lee et al. [8], seen before in Section 2.1, has an interesting side effect – it not only clusters trajectories, but it can also be used to find a representative trajectory for every cluster. So, it is a combination of both techniques. Additionally, the algorithm is time-independent, since it only uses the geometry of the trajectories. The included timestamps are left unused.

Another paper which combines clustering and representative trajectory generation is the one by Andrienko et al. [9], mentioned before in Section 2.1. It uses the aforementioned Central Trajectories algorithm to do summarization of the trajectories. It does not require the trajectories to have equal lengths.

Johard and Ruffaldi [22] focus on human movement analysis. The goal is to study trajectories followed by human body parts during a certain activity, such as sports. To measure the skill of the player, a good metric would be to analyze the variance of movement across multiple repetitions of the same action, such as kicking a football. The main idea is that a low variance is an indication that the player can reliably perform the same action repeatedly. On the other hand, a high variance means the player has not mastered this particular motion yet, resulting in clumsy movement that cannot be accurately reproduced. Thus, to quantify player skill, the paper introduces two statistical measures that can be applied to clusters of trajectories: the *mean trajectory* and the *trajectory variance*. The mean trajectory actually has a misleading name: it does not necessarily have to be the mean of the trajectories, it should just be representative of the set of trajectories in the input cluster. In the paper, the mean trajectory is calculated using an approach which is a generalization of the Dynamic Time Warping (DTW) Barycentric Averaging (DBA) algorithm, which combines the DTW algorithm with a self-organizing map. The algorithm can be considered time-independent: since Dynamic Time Warping is applied to the points, time information is not relevant, except for the order of the points. The trajectory variance is a measure of how much the trajectories in the cluster diverge from each other at a certain point in time. The variance can vary over time – ideally, it is minimized along the entire length of the mean trajectory. If all trajectories inside the cluster are identical, then the trajectory variance is zero. The variance depends on the mean: since the variance can be considered as a second-order moment, and the mean as a first-order moment, the two are closely related.

2.3 Visualization

Since this thesis focuses on trajectories, and trajectory data is hard to digest in its raw, textual form, we need some kind of tooling to visualize the trajectory data, preferably on top of a geographic

representation of the environment in which the trajectory data was captured. For instance, if we have gathered GPS data from taxis in Los Angeles, we would like to plot the data on top of a map of Los Angeles. There are some tools out there to do this, but data scientists have also thought of new and inventive ways to visualize the data to make it even easier to understand.

For example, the paper by Andrienko et al. [9] which was mentioned before, aims to visualize air traffic in a multitude of different ways. After applying trajectory clustering, the trajectories can be visualized either in 3D, in which more common trajectories are drawn as thickened cylinders, or in 2D, using a density map drawn on top of a topographical one. The latter approach reveals not only commonly used air plane trajectories, but also the holding lines they follow before landing.

In another paper by the same authors [10], trajectories of airplanes are plotted in polar coordinates onto a density plot. The result is essentially a circular heatmap: an intuitive visualization of the density of air traffic flight direction – in other words, a quick way to see in which direction air traffic tends to fly.

Gomes et al. [23] aim at detecting, in real-time, the formation (or dissipation) of *hot routes* from continuous trajectory data streams. Hot routes are parts of the road network that have a very high traffic density, indicating possible traffic jams and accidents. Using a GPU based approach, they can also visualize these hot roads on a map, giving a quick and real-time indication of possible traffic events happening in a certain area. The algorithm has multiple parameters, such as density threshold and continuous analysis time, which can be adjusted in real-time, giving rise to experiments to quickly find the optimal values.

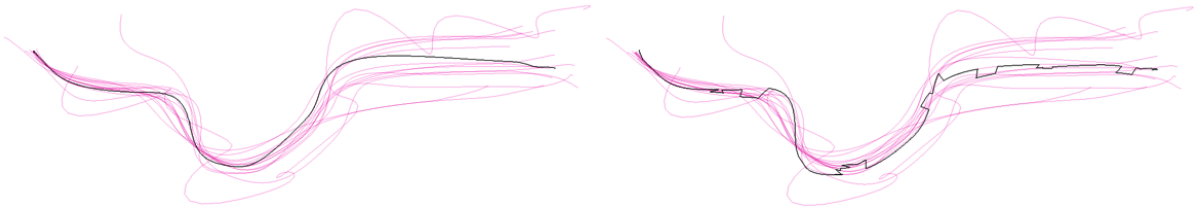
The thesis by Konzack [4] focuses on trajectory analysis and visualization. The problem domain is the visualization of seagull trajectories and their stopovers. The thesis introduces multiple methods to visualize their trajectories, and also ways to correlate different trajectories. For instance, the author introduces a method to quickly visualize the time difference between two seagulls that are travelling together. Both trajectories are drawn in lat-long space, but colored lines are drawn between them. Red lines indicate the first seagull is ahead of the second seagull (in time); blue lines indicate the opposite. This makes it easy to see which seagull is following the other seagull, and the moments at which their leadership changes, without needing to watch the entire trajectory like a movie.

2.4 Classification

A small amount of papers are dedicated to trajectory classification. This task is slightly different from trajectory clustering, since classification is applied to *individual* trajectories, *after* discovering clusters of trajectories. As stated by Vlachos et al. [19], the problem of classification is formally defined as follows: given a database D of trajectories and a query trajectory Q (not already in the database), we want to find the trajectory T that is closest to Q. Thus, classification algorithms usually aim to build some kind of indexing data structure, which generally requires some time to preprocess, after which new, unseen trajectories can be classified in real-time.

Vlachos et al. [19] discover similar multidimensional trajectories using similarity functions based on the Longest Common Subsequence (LCSS). According to the authors, this method is more efficient than the Euclidean and DTW distance metrics, especially when noise is present. Additionally, using a specialized data structure, nearest neighbor queries can be answered quickly – that means, formally, given a trajectory T, find the trajectory T' which is most similar to T. The approach scales to arbitrary dimensions, and can handle trajectories with irregular sampling rates or speeds, which makes it quite powerful. Finally, unlike the Euclidean distance, their approach can also compare two trajectories with a different amount of points, and also considers trajectories that are identical but translated copies of another trajectory to be similar.

Lee et al. [24] focus on the domain of nautical vessel identification. By tracking the GPS location of vessels on the sea, it can be classified as a multitude of possible vessels. For example, a vessel that visits a refinery will likely be a tanker, and a vessel that comes across a fishery will likely be a fishing boat. Their trajectories might be really similar, which means most classification systems will fail. Instead, they use *region* and *subtrajectory* based classification, which means classification happens based on which regions the vessel passes through, and which subparts of trajectories are near a certain



(a) Using epsilon $\epsilon = 2$.

(b) Using epsilon $\epsilon = 20$.

Figure 4: Some example Central Trajectories generated by the implementation. The pink lines indicate the original trajectories within a cluster, and the black line indicates the resulting Central Trajectory. The cluster was taken from the dataset generated by TrajGen.

waypoint. In the previous example, the subparts of the trajectory that lead to a refinery are part of the *subtrajectory*, and the fishery is the *region*. Thus, they first apply region-based clustering, and then trajectory-based clustering. The trajectory-based clustering algorithm is based on the partition-and-group framework proposed by Lee et al. [8]. By combining the two clustering approaches together into a hierarchical approach, the classification accuracy and efficiency are increased significantly.

3 Background

This section will explain the background knowledge needed to understand this research project.

3.1 Trajectory models

To represent a trajectory, the most common method used is to simply store a list of points, optionally with time stamps. The trajectories are assumed to follow the points, linearly interpolating between them. This means the trajectory points are stored discretely, but the trajectory itself is considered continuous, since there are no "jumps" or "gaps" in it. Deciding whether a trajectory can be considered discrete or continuous depends on the author's interpretation of it. In the discrete case, a trajectory A is modeled as a list of n points: $\boldsymbol{a_i}$ with $1 \leq i \leq n$ and $i \in \mathbb{N}$. In the continuous case, a trajectory A is modeled as a vector-valued function of time $\boldsymbol{a}(t)$ with $0 \leq t \leq l$, where l is the length of the timespan covered by the trajectory, and $t \in \mathbb{R}$. In this paper, I will assume trajectories are continuous.

Note that in this paper, the words "trajectory" and "entity" will be used interchangeably.

3.2 Central Trajectories

As mentioned before in Section 2.2, a Central Trajectory (CT) is a *trajectoid,* a trajectory which consists of sub-parts of the input trajectories, which stays as central as possible according to some centrality measure. The algorithm takes a cluster of trajectories as input, and outputs a single trajectory. In the paper, the centrality measure used is the distance to the furthest point across all trajectories at a given time t. The trajectory might jump to new trajectories at certain points, if this causes the trajectory to be more central at that moment in time, although its jump distance is limited by the parameter epsilon (ϵ). Since all coordinates in this paper are defined in meters, ϵ is defined in meters as well.

When it is possible to jump between two entities σ and ψ via a sequence of other entities, without jumping more than ϵ units at a time, the two entities are said to be *ϵ-connected* at a time t.

Some example Central Trajectories can be found in Figure 4 and 5. As you can see, increasing ϵ makes the trajectory more erratic and more likely to contain noticeable discontinuities. The higher epsilon, the more likely the algorithm is to prefer centrality over aesthetics. Generally speaking, if ϵ is equal to or smaller than the size of a pixel on the screen, then the jumps are not noticeable.

The algorithm consists of the following four steps:

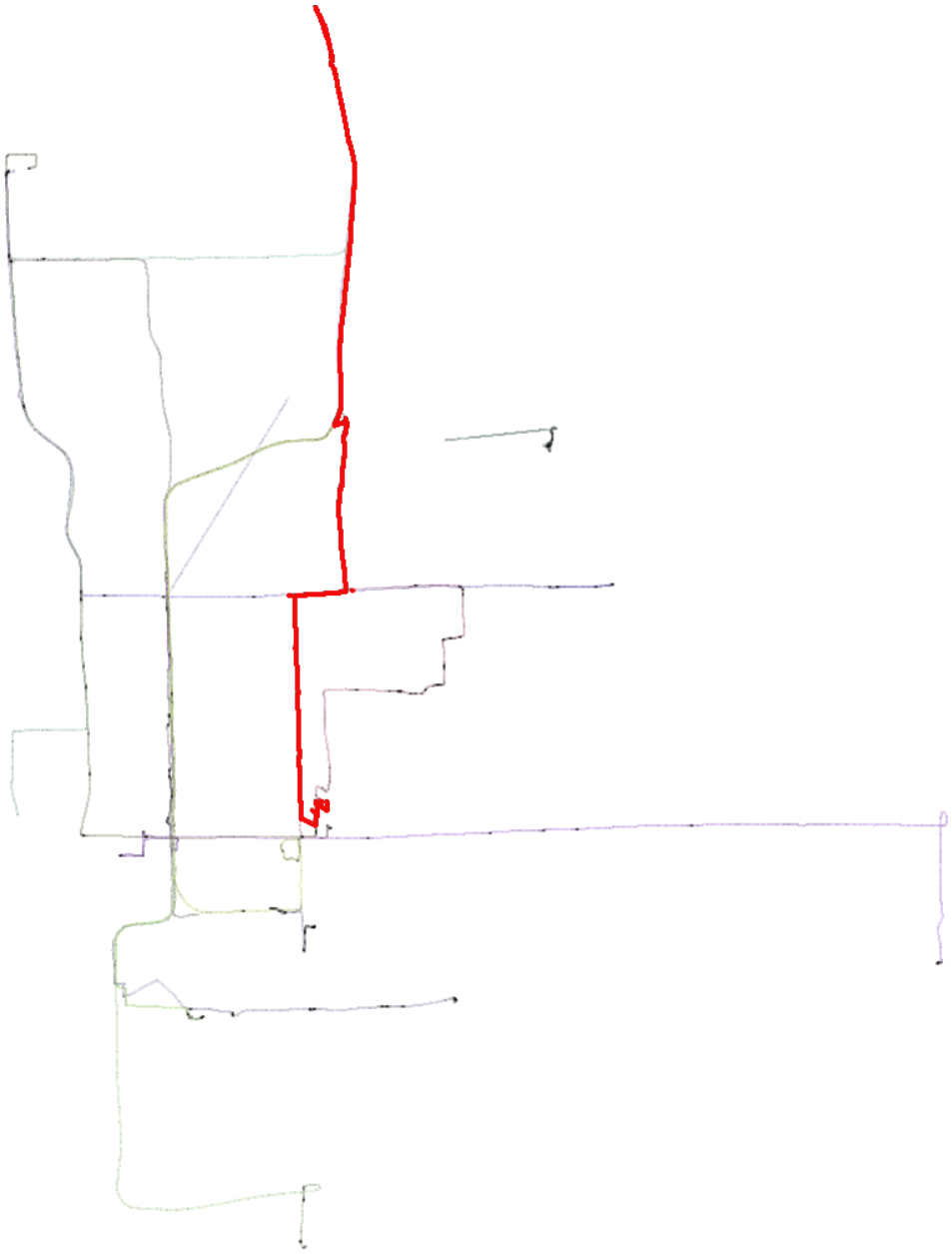


Figure 5: An example Central Trajectory generated on a cluster found in the real-life Geolife dataset. The thin colored lines represent individual trajectories, and the thick red line indicates the Central Trajectory. The trajectories all start at seemingly random points, but they all eventually come together, travelling on the same road. This is the same road that the Central Trajectory has chosen as its most central part, which means it is doing its job correctly.

1. Find Reeb Events
2. Create Reeb Graph
3. Calculate Edge Weights
4. Find Central Trajectory

They will be described in detail below.

3.2.1 Find Reeb Events

A Reeb event is a moment in time when any two trajectories lie exactly ϵ units away from each other. The first stage of Central Trajectories aims to find all Reeb events between all pairs of trajectories – this can be done by solving a simple quadratic equation per line segment pair, giving either 0, 1 or 2 solutions. Every Reeb event also stores a boolean, indicating two possibilities: either the trajectories were more than ϵ units apart beforehand and they just came together, or they were less than ϵ units apart beforehand and they just split up. Note that the pathological case where two trajectories are exactly ϵ units apart for their entire duration, resulting in infinite Reeb events, is assumed to be absent. This is because the input trajectories are assumed to be in *general position,* so no two line segments in any trajectory are parallel.

3.2.2 Create Reeb Graph

A Reeb graph is a directed acyclic graph that captures the structure of a two- or higher-dimensional scalar function, by considering the evolution of the connected components of the level sets[5]. In the case of Central Trajectories, it captures the change (forming and disbanding) of the groups of trajectories over time. A vertex in the Reeb graph represents a moment when two groups of trajectories split up or join together: it is called a *Reeb vertex*. An edge in the Reeb graph represents a group (or a *component*) of (one or many) trajectories: it is called a *Reeb edge*. The component of a Reeb edge e is referred to as C_e. Since the graph takes time into account, every Reeb vertex has a timestamp, and every Reeb edge has a timespan in which it is active: it is the time between its start vertex and its end vertex. This means it is possible to take a cross-section of the Reeb graph by fixing a certain value of t: this cross-section reveals which groups of trajectories are active at time t.

Given the Reeb events from the previous step, the Reeb graph can be constructed in a fairly straightforward manner, as described in the paper by Buchin et al. [5].

There are four different kinds of Reeb vertices:

Reeb Start This vertex type appears only at the start of the Reeb graph. There is one Reeb start vertex for every separate component of ϵ-connected entities at $t = 0$ (the very start). It has no incoming edges, and it has 1 outgoing edge.

Reeb End This vertex type appears only at the end of the Reeb graph. It has no outgoing edges, and it has 1 incoming edge.

Reeb Split This vertex type appears when a group of trajectories splits up into two. This happens when two entities are no longer less than ϵ units apart, and no other trajectory remains to stay ϵ-connected with the rest of the group. It has 2 outgoing edges, and it has 1 incoming edge.

Reeb Merge This vertex type appears when two groups of trajectories merge into one. This happens when two entities, which were not ϵ-connected before, become less than ϵ units apart. It has 2 incoming edges, and it has 1 outgoing edge.

3.2.3 Calculate Edge Weights

All edges in the Reeb graph are assigned a weight. This weight can be seen as the *centrality* of the edge: the lower the weight, the higher the centrality of the edge. That means, if an edge has a low weight, the trajectories in the edge's associated group tend to lie in the middle of all trajectories, during the time span of this edge. On the contrary, if an edge has a high weight, its trajectories tend to lie on the outskirts.

The exact definition of centrality is flexible: in the paper, centrality is defined as the distance to the furthest point across all trajectories at a given time t. Given that a Reeb edge is defined on a time span, not a single timestamp, the integral of the centrality over the time span is used as edge weight.

Formally, given a Reeb edge e, its weight ω_e is defined as follows:

$$\omega_e = \int_{t_u}^{t_v} \mathcal{L}(\mathcal{F}_e)(t)\mathrm{d}t$$

Where $\mathcal{L}$ is the *lower envelope* of a set of functions (or the *minimum*), t_u is the start of the timespan of the edge, t_v is the end of the timespan of the edge, and $\mathcal{F}_e = \{f_\sigma \mid \sigma \in C_e\}$ is the set of weight functions of all trajectories contained in the edge's component C_e. The weight function f_σ of a trajectory σ is defined as the distance between σ and the trajectory furthest away from σ (of all trajectories, not just the ones in C_e) over time. Formally:

$$f_\sigma(t) = \max_{\psi \in \chi} \|\sigma(t)\psi(t)\|$$

Where $\sigma(t)$ indicates the (interpolated) location of the entity at time t, $\|\sigma(t)\psi(t)\|$ indicates the Euclidean distance between the entities σ and ψ at time t, and χ is the set of all trajectories.

To sum it up, at every moment in time on the timespan of the edge, the weight functions of all trajectories in its component are calculated. The minimum is taken over the output of these functions, and this value is used in the integral as part of the total weight of the edge.

The integral can be calculated in two ways: either analytically or numerically. Doing it analytically gives the most precise results, however it requires an integral solver and is difficult to implement, since it requires splitting up the Reeb edges at every moment in time the maximum changes in any of the f_σ functions relevant to that particular edge. (This moment is called a *breakpoint*, since it indicates moments in time in which the weight graph of the edge displays a discontinuity.) Doing it numerically means approximating the integral, using a Riemann sum, for example.

3.2.4 Find Central Trajectory

Once all Reeb edges have a weight, the final step is to find a minimum-weight path from a source to a sink in the Reeb graph. Since an edge with a low weight has a high centrality, the resulting path is a trajectoid with maximum centrality. In the case that a Reeb edge on the path holds a group of more than one trajectory, the trajectory with the highest centrality is used. The most central trajectory in the group might change during the span of a single Reeb edge: if this happens, the trajectoid jumps to another trajectory in the group. This is always allowed, since all trajectories in a group are all within distance ϵ of each other. This jump event is called a *mid-edge jump*: a jump occurring in the middle of a Reeb edge. There are also *on-vertex jumps*, which means the Central Trajectory jumps to another trajectory at a Reeb vertex, instead of a Reeb edge.

3.3 Elementary Resample

After clustering the trajectories, there is one more necessary step before the Central Trajectories algorithm can be applied. Since it requires that all input trajectories have the same amount of points, the trajectories have to be resampled in such a way that the following requirements hold:

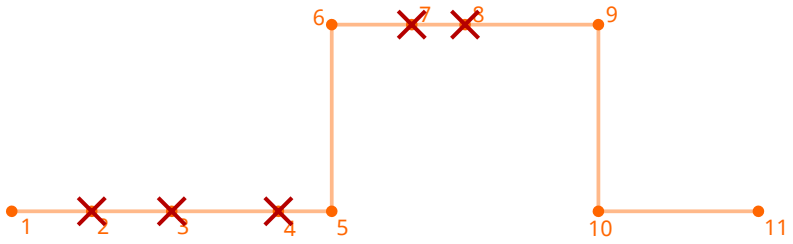


Figure 6: An example of simplification. In this trajectory, many redundant points can be removed, without changing its geometry. However, non-linear time information may be lost.

- All trajectories start/end at exactly the same moment in time.
- All trajectories have their points at exactly the same moments in time.

This operation is called an *elementary resample*. For example, define a cluster of two trajectories $\{A, B\}$, where A has 3 points on the times $t = 1$, $t = 3$ and $t = 6$, and B has 4 points on the times $t = 2$, $t = 4$, $t = 6$ and $t = 7$. After elementary resampling, both trajectories will have their points at $t = 2$, $t = 3$, $t = 4$ and $t = 6$. As you can see, this operation might cut off parts of a trajectory.

This might pose a problem in the case when two trajectories do not overlap in time. For example, if trajectory A has 2 points on the times $t = 1$ and $t = 6$, and B has 2 points on the times $t = 7$ and $t = 10$, then this cluster cannot be elementary resampled.

3.4 Clustering

The Central Trajectories algorithm assumes all input trajectories are somewhat clustered. If this is not the case, a clustering algorithm must be run first. Two clustering algorithms were used, and they will be described below:

3.4.1 K-means

The idea of K-means is to cluster a bunch of points into a predetermined amount of clusters k, alternating two steps for a certain amount of iterations: *Assignment* and *Update*. The algorithm was adapted to work on trajectories as well. This works as follows: every point of every trajectory is used as input for K-means. After clustering, every trajectory counts how many of its points lie in a certain cluster: this is done for every cluster. Then, the trajectory is assigned to the cluster which has the most of the trajectory's points inside of it.

3.4.2 QuickBundles

This algorithm, introduced by Garyfallidis et al. [25], is developed to be used in neuroanatomy. When an MRI scan is made of the brain during a tractography, a very large amount of nerves is revealed in the form of streamlines. Such a large amount of streamlines is hard to visualize, interact with, and interpret. QuickBundles aims to quickly gather usable clusters from such large datasets. A streamline is a list of three-dimensional points: it is almost a trajectory, except for the fact that the time dimension is missing. Additionally, a streamline is three-dimensional, while a trajectory is two-dimensional. To use the algorithm for trajectories, the time dimension is thrown away and the Z-coordinate is set to zero for all points. The algorithm takes a threshold parameter, which determines the maximum distance between two trajectories before they are no longer considered to be part of the same cluster. Like all spatial parameters in this thesis, it is defined in meters.

3.5 Simplification

To reduce the complexity of a trajectory, simplification can be applied, which makes a new trajectory which is similar to the original, but the amount of vertices is greatly reduced. See Figure 6 for a visual demonstration. An example of a widely used simplification algorithm is the Douglas-Peucker algorithm[26]. This algorithm has a parameter delta (δ), which determines the maximum distance (in meters) between the original curve and the simplified curve (using the Hausdorff distance metric).

In a nutshell, the algorithm works recursively, by repeatedly marking a first and last point and finding the most distant point between them (that means: the point furthest away from the line segment between the first and last point). If this point has a distance of $\leq \delta$ to the line segment, then all unmarked points are discarded. If not, the point is marked (so it is kept), and the algorithm recurses on two halves: one half from the first to the distant point, and another half from the distant to the last point.

Note that this algorithm does not take time into account, which may cause non-linear time information to be lost during simplification. As mentioned before, it may be tempting to use the three-dimensional variant of the Douglas-Peucker algorithm, in addition to using time as the Z-axis. However, this will give very non-intuitive results, since the X- and Y-dimensions are defined in meters, yet the Z-dimension is defined in time: as a result, no distance metric will give sensible results in this space.

4 Methodology

To answer the research questions, I created a program satisfying various requirements, and I gathered various datasets. Using this program, I conducted various experiments. They will all be described in this section.

4.1 Implementation

This section describes the implementation details of the various implemented algorithms.

The implementation was written in C++ for two reasons. First off, C++ is a highly performant language, which means it can quickly deal with potentially very large amounts of trajectory data. Secondly, there are many useful libraries such as CGAL[27], Boost, and MoveTK, which provide implementations of a variety of geometry, graph, and trajectory algorithms respectively, which are all made to be used via C++.

4.1.1 Central Trajectories

The Central Trajectories algorithm is implemented by following the four steps described before in 3.2. There are some implementation details to note:

- In the "Create Reeb Graph" step, the Boost.Graph library is used to create the graph. Additionally, there are some additional "dummy" vertices and edges in the graph, at the very start and at the very end. The dummy vertices are connected to the Reeb start and end vertices via dummy edges. This means we can find a shortest path from a source to a sink in the graph by using the dummy start vertex as the source, and the dummy end vertex as the destination. This avoids the need of trying to find a path from every possible Reeb start vertex to every possible Reeb end vertex: a possible combinatorial explosion.
- In the "Calculate Edge Weights" step, the edge weight integral is numerically approximated using a Riemann sum, using a maximum of 1000 samples over the entire time span of all trajectories. This value appears to be sufficient for all used datasets.

- In the "Find Central Trajectory" step, the decision to perform a mid-edge jump is considered at every point in time the previously described Riemann sum is evaluated. That means, since the Riemann sum in total is evaluated 1000 times, there can be no more than 1000 mid-edge jumps in a Central Trajectory. However, this does not appear to have a great effect on the end result, since the great majority of the jumps are on-vertex jumps, not mid-edge jumps.
- In the "Find Central Trajectory" step, the Dijkstra algorithm is used to find a shortest path from a source to a sink.

4.1.2 Clustering

The two clustering methods, described before, were either implemented manually or imported from a library. Implementation details will be described below.

K-means This algorithm was manually implemented. Since this is a very basic algorithm, it has some issues: for instance, trajectories with greatly varying point density bias the results.

QuickBundles An implementation of the algorithm can be found in the DIPY library[28], a Python library made specifically for computational neuroanatomy. To be able to use this library from C++, the Boost.Python library was used to be able to run Python code from C++.

In practice, the algorithm appears to generate very uneven clusters: some clusters have hundreds of trajectories, while a lot of clusters contain only a single trajectory. To mitigate this problem, a minimum cluster size filter was implemented, such that clusters that have less than x trajectories are discarded, where x depends on the dataset used. This also helps tremendously in reducing the amount of outliers: since outlier trajectories become part of their own cluster, containing only 1 or 2 trajectories, they are discarded automatically.

Another problem with QuickBundles is the fact that it does not differentiate between trajectories that have the same shape and geometry, but are travelling in opposite directions. To fix this, a "cluster splitter" was implemented, which takes the clusters from QuickBundles and splits them up into smaller clusters, depending on the angle of the trajectories. A quick summary of how it works:

- For every trajectory σ, calculate its angle $\alpha(\sigma)$. (The angle of a trajectory is simply the angle between its first and its last point.)
- Convert the point (1, $\alpha(\sigma)$) from polar coordinates to Cartesian coordinates. This results in a point $(x(\sigma), y(\sigma))$ on the unit circle.
- For every cluster C of trajectories:
  - Gather a list of all the points $(x(\sigma), y(\sigma))$ for every trajectory $\sigma \in C$ and use the list as input for a point clustering algorithm. In this case, the DBSCAN algorithm was chosen, with a minimum sample size of 1 and an epsilon of 0.5. (Note: epsilon is the name of a parameter of DBSCAN, not to be confused with Central Trajectory epsilon.)
  - Create a new unique cluster for every cluster the clustering algorithm returns, and assign the trajectories in C to it accordingly.

Intuitively, every trajectory is turned into a point on the unit circle, and these points are clustered. Since the clustering algorithm is constrained to work within the trajectories of one "original" cluster (that means, a cluster QuickBundles returned originally), no two trajectories from different "original" clusters can be merged back together – which is a good thing, because if they were anywhere near each other, they would be part of the same "original" cluster anyway.

Finally, clusters returned by the cluster splitter which have less than 3 trajectories are removed, to prevent running CT on clusters that are too small to return useful results.

Dataset	Trajectory count	Spatial range	Start date	End date
TrajGen	16,016	1km by 2km	-	-
SmallDutch	1300	47km by 46km	January 9, 2019	January 10, 2019
Starkey	253	8km by 14km	May 7, 1993	August 15, 1996
T-Drive	2500	250km by 332km	February 2, 2008	February 8, 2008
Geolife	14,300	100km by 100km	April 12, 2007	May 14, 2012
OpenPFlow	17,000	218km by 148km	January 12, 2017	January 13, 2017
MarineCadastre	5000	407km by 9000km	April 1, 2007	April 30, 2007

Table 1: A summary of all dataset characteristics. Note that some preprocessing steps were applied before determining the trajectory count and spatial range, to ensure outliers are not counted. A dash (-) indicates the given field is not relevant (e.g. date is not relevant for synthetic data).

Note that not all datasets use the cluster splitter: since it reduces the total amount of clusters, it also reduces the statistical validity of the results, and it causes outliers to have a much greater effect on the average trends. Since some datasets have a small amount of clusters to begin with, it would not be advantageous to use the cluster splitter on them: for example, Starkey contains only 253 trajectories. Additionally, some datasets contain trajectories that are too complex to classify based on their angle: two examples of this are Starkey and MarineCadastre. These datasets contain trajectories that are more complicated than just a path from A to B: they usually go back and forth multiple times, taking various detours and visiting many destinations in a row. On all other datasets, however, the cluster splitter works quite well.

4.2 Data

To perform the research experiments, which are described in Section 4.3, I have gathered many datasets, ranging from vehicle location data to marine location data, and even some animal trajectories. The reasons I picked such varied datasets are as follows:

- It enables the analysis of the general trends that occur when changing various parameters of the CT algorithm, to ensure the results are useful for a wide variety of research branches, which generally use very different datasets.
- It prevents a dataset from being an outlier itself. That means, if one dataset happens to give totally different results compared to most other trajectory datasets, then this will not bias the results.
- It allows analysis of the effect of various dataset characteristics on the results. For example, some datasets are spatially larger than others, while others may have trajectories that have a much higher sampling rate, others may have very angular trajectories, et cetera. By analyzing the datasets themselves alongside with the results, a correlation can be drawn between specific patterns in the results and the unique characteristics of the dataset.

4.2.1 Datasets

This section will describe all datasets that I have used. A summary of all dataset characteristics can be found in table1.

SmallDutch This is a dataset gathered by the company Here consisting of GPS data of cars travelling in the Netherlands, near The Hague and Rotterdam. The data was gathered in a period of 24 hours, ranging from 9 January 2019 23:00 to 10 January 2019 23:00. The spatial range of the data is roughly 47km by 46km. The data is actually a subset of a much larger dataset, reducing it to only 1300 trajectories. The sampling frequency differs per trajectory, ranging from 1 to 10 seconds between

points – it does not vary over time. The amount of points per trajectory varies from 20 to 1500, and the average duration of a trajectory is about 15 minutes.

A visualization of the dataset can be found in Figure 7a.

Starkey The Starkey dataset was gathered in a long-term study of elk, deer and cattle behavior, examining the effects of ungulates on ecosystems[29]. It is one of the most comprehensive field research projects ever attempted. Using GPS trackers, the location of various animals was tracked during a time span ranging from 7 May 1993 to 15 August 1996, although the research was only conducted during spring, summer and fall. In the winter, the animals were fed manually. The experiment was conducted in a very large animal enclosure near La Grande, Oregon, USA, spanning a region of 8km by 14km. Since the enclosure is larger than the summer home range of most deer and elk, the animals inside the area are living under conditions similar to wild, free-ranging herds. This ensures the resulting trajectories form a realistic representation of animal behavior in the wild.

The dataset consists of 253 trajectories, most of them spanning the entire duration of the study. The amount of points per trajectory varies from 161 for short trajectories, up to 4000 for long trajectories.

Since data was not collected during the winter months, the dataset was split up on a per-year basis. That means, for instance, if a trajectory ranges from 1993 to 1996, it was split up into four trajectories, one for every year it spans. This prevents the trajectories from displaying erratic behavior, since there are no samples during the winter months. Additionally, it makes clustering easier.

A visualization of the dataset can be found in Figure 7b.

T-Drive This is a dataset containing GPS trajectories of 10,357 taxis, gathered in a period of one week, from 2 February 2008 to 8 February 2008, near the city of Beijing[30]. There are about 17 million points in the dataset, occupying around 750MB of disk space, and the total distance travelled is about 9 million kilometers. The average sampling interval is about 177 seconds, with an average distance between two points of about 623 meters.

This dataset contains a lot of outliers: there are lots of random points that are way off, and lots of zeroes as well. These outliers were filtered by removing all points outside a rectangle spanning the majority of the city Beijing and the surrounding areas. Most trajectories span the entire range of one week, with some outliers starting a few days after 2 February and some ending a few days before 8 February.

Since the trajectories are very long and complex, the dataset was split up by day. Because the dataset spans a week of data, on average, every trajectory was split up into seven smaller trajectories. After splitting, trajectories with a bounding box area smaller than 2,800,000,000 m^2 were filtered out. After filtering, about 2500 trajectories remain, with an amount of points varying from 5 up to 9000, all of which are contained within a bounding box of 250km by 332km.

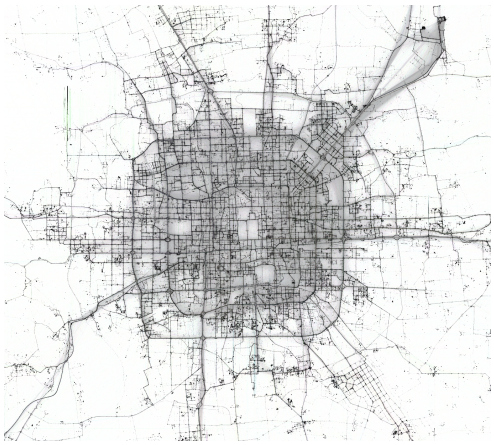
A visualization of the dataset can be found in Figure 7c.

Geolife This is a dataset gathered in the Microsoft Research Asia Geolife project over a time span of three years, from April 2007 to May 2012[31]. It is a fairly large dataset, both in storage and in dimensions: it consists of 1.6GB of raw data, and spans a region of 30,000km by 40,000km, although the dataset contains some outliers – most of the trajectories are contained in the 100km by 100km region surrounding Beijing. This makes the dataset similar to T-Drive, except it actually contains a greater variety of transportation modes: not only just taxis, but also pedestrians, cyclists, regular cars and even some airplanes. There are 14,300 trajectories in the dataset, with a sampling frequency ranging between 1 and 10 seconds per point, although there are some outliers which can go up to 200 seconds between points. This means the amount of points per trajectory varies greatly, ranging from 18 to 11,000. Additionally, a single trajectory can last anywhere from 1 minute to 30 hours. This makes sense, since the datasets contains a multitude of different transportation modes.

A visualization of the dataset can be found in Figure 7d.

(a) SmallDutch.

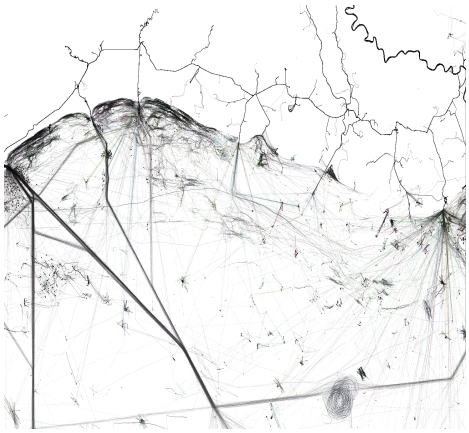
(b) Starkey.



(c) T-Drive.

(d) Geolife.

(e) OpenPFLOW.



(f) MarineCadastre.

Figure 7: Visualizations of the datasets used.

OpenPFlow This is a dataset based on human movement in urban areas, gathered in and around the city of Tokyo[1]. For privacy reasons, the data was anonymized such that no entity in the dataset matches the actual movement of any real person. A simulator was written which uses the real data to generate realistic but synthetic movement data – the output of which can be freely shared online. This means the dataset is technically synthetic, although since it was based on classified real data, the resulting synthetic dataset is very realistic.

Another result of the artificiality of the data is the volume of the data – since the simulator can generate an unlimited amount of data, the entire dataset can be made very large without much effort: 43.8 GB. Due to its sheer size, only a small subset of the data (99 million points) was analyzed in my thesis. This results in about 17,000 trajectories. The spatial range is roughly 218km by 148km.

An interesting quirk of this dataset is that every single trajectory contains about 1200 vertices, with a very small deviation (the majority of the trajectories have between 1197 and 1201 vertices, with some very exceptional outliers). This may sound like a good thing, however, remember that Central Trajectories not only requires all trajectories to have an equal amount of points, but also that all trajectories are sampled at exactly the same moment in time, which is not always the case. Thus, elementary resampling is still required.

A visualization of the dataset can be found in Figure 7e.

MarineCadastre This dataset, gathered by the U.S. Coast Guard Navigation Center from 2009 to 2017, consists of vessel traffic data containing the location and characteristics of large vessels in U.S. and international waters. It includes information such as location, time, ship type, speed, length, beam, draft, et cetera. Like OpenPFlow, due to its sheer size (84GB just for the 2017 data), only a small subset of the data was actually used. The dataset is split up by year, month, and UTM zone. There are 60 UTM zones in total, spanning the Earth entirely, although only 20 of the zones cover the U.S., which means not every zone appears in the dataset. Every UTM zone covers a vertical slab of the earth, each 6° of longitude in width. The 15th zone was used, using only data gathered in April 2007, resulting in about 8.7GB of data.

The used part of the dataset has about 5000 trajectories. Most trajectories are located in the northern half of the Gulf of Mexico. Since this zone contains a large part of the Mississippi river, and many ships follow a common route when entering or leaving a harbor, the dataset can be easily clustered. The region spanned is about 407km horizontally by 9000km vertically, although most of the data is near the center of the region. There are a lot of vertical outliers.

The dataset contains a lot of trajectories that are almost completely stationary – these are likely to be fishing ships, remaining near the same location for the entire duration of the trajectory. Since it is impossible to generate a sensible Central Trajectory for these entities, the dataset was filtered: trajectories that have a total bounding box area below a certain threshold are removed.

A visualization of the dataset can be found in Figure 7f.

4.2.2 Data generation

To increase the amount of useful pre-clustered data, I have written my own data generation program called TrajGen using the Godot game engine. It generates semi-realistic trajectories in clusters of entities, which tend to move together, although with a slight randomization applied. The entities are simulated using simple Euler integration, taking position, velocity and acceleration into account. They are instructed to roughly follow an AI agent that finds a path from a starting point to a destination, situated on a grid of random blocks. The path can deviate on a per-agent basis, since the starting point and destination are slightly randomized. This results in trajectories that are mostly similar within a single cluster, although some agents might take a different route compared to the other agents in the cluster. The result of this is that the trajectories have some context to them – a world of obstacles that need to be avoided. This is one of the situations in which the Central Trajectories algorithm shines: since a Central Trajectory consists solely of parts of the input trajectories, it can never go through obstacles. Some example generated trajectories can be found in Figure 8.

Figure 8: A selection of trajectories generated by TrajGen. The colors indicate clusters. Every cluster contains 16 trajectories.

The dataset generated with TrajGen consists of 16,016 trajectories, which are already clustered and elementary resampled by default, which means there is no need to apply any processing to the data before running experiments on it. The amount of points per trajectories varies from 70 to 150, since the length and sampling rate varies per trajectory, giving rise to a total amount of 1,524,960 points.

4.2.3 Data preprocessing

Most of the datasets required some form of preprocessing before they could be used in the results. A summary of preprocessing steps applied on the various data can be found in Table 2. The values in the columns were found by experimentation, aiming to extract about 2000 useful trajectories per dataset, clustered into roughly 50 clusters. This prevents running into computational and memory limits, while still getting a statistically useful sample size. Additionally, all trajectories in all datasets were shifted to the start (after the "Split by" step), to ensure the trajectories within a cluster overlap at least partially in time.

A notable exception is of course the TrajGen dataset, consisting of synthetic data: this dataset was not preprocessed, since it was generated in such a way to (a) have no outliers (b) be aligned in time, mitigating the need for elementary resampling and time shifting, and (c) be already clustered. Additionally, it has way more clusters than the others: up to 1000.

In Table 2, the column "Split by" indicates whether or not the dataset was split by a certain time measure. That means, for example, if the dataset was split by year, then every trajectory that spans more than a year is split up into multiple trajectories, one trajectory per year.

The preprocessing pipeline is applied in the following order: "*(Split by x $\rightarrow$) Shift to start $\rightarrow$ Remove trajectories below bounding box area $\rightarrow$ QuickBundles $\rightarrow$ Elementary resample*". Generally speaking, this pipeline works very well on very large datasets such as T-Drive, Geolife, OpenPFlow and MarineCadastre. This is because, for clustering to work well, there are four desiderata:

- The trajectories are mostly aligned in time: that means, all trajectories cover mostly the same period of time. If this is not the case, then the resulting clusters cannot be elementary resampled, which is a necessary step before Central Trajectories can be applied. (Remember, a cluster can only be elementary resampled if all trajectories in the cluster overlap in at least one moment in time.)
- The trajectories are not too long and complex. In the case that the trajectories are long, then ideally they do not go across the same area more than once, follow a non-self-intersecting curve, and have regular sampling rates.
- The trajectories have a clear direction, and do not remain near the same point for the entirety of its duration. If this criterion is not met, then the resulting clusters are not very useful for the Central Trajectories algorithm, due to their shortness.

Dataset	Cluster method	Elementary resample	Rectangle outlier removal	Split by	Remove trajectories below bounding box area (m^2)
TrajGen	-	-	-	-	-
SmallDutch	QuickBundles, threshold 2000 (min clus 7) + cluster splitter	Yes	-	-	-
Starkey	QuickBundles, threshold 1500	Yes	-	Year	-
T-Drive	QuickBundles, threshold 9000 (min clus 7) + cluster splitter	Yes	Yes	Day	2,800,000,000
Geolife	QuickBundles, threshold 3000 (min clus 7) + cluster splitter	Yes	-	-	100,000,000
OpenPFlow	QuickBundles, threshold 3500 (min clus 7) + cluster splitter	Yes	-	-	230,000,000
MarineCadastre	QuickBundles, threshold 15000	Yes	-	-	18,500,000,000

Table 2: A summary of preprocessing steps applied to the datasets. A dash (-) indicates the preprocessing step was not applied.

- There are not too many trajectories in total – the manageable limit seems to be about 5000. Going beyond this limit causes the QuickBundles algorithm to perform extremely poorly, potentially taking hours to cluster a dataset. However, a dataset of about 2500 trajectories can be clustered in a few minutes. (Although the algorithm is called QuickBundles since it can supposedly quickly cluster a large dataset, the implementation is written in Python, which causes a large performance penalty.)

The "*Shift to start*" step provides the first desideratum, the "*Split by x*" step provides the second, the "*Remove trajectories below bounding box area*" step provides the third and fourth, and the "*QuickBundles*" step also provides the fourth (remember: clusters that have less than three trajectories are removed, so outliers are nicely discarded). The datasets SmallDutch, Geolife and OpenPFlow already have relatively short and simple trajectories, which makes the "*Split by x*" step unnecessary.

The column "Cluster Method" indicates the cluster method used. By default, a minimum cluster size of 3 is used: if this is not the case, then it is indicated within parentheses: "(min clus x)". Additionally, datasets that use the cluster splitter are indicated by "+ cluster splitter".

4.3 Experiments

To answer the research questions, experiments have been be conducted using the implemented algorithms. The experiments are as follows:

4.3.1 Question 1

As stated before, the first research question is:

What is the real-world complexity of Central Trajectories?

To find the answer, I have conducted two experiments:

1. A dataset has been clustered, elementary resampled, and then Central Trajectories was applied to every cluster individually, while measuring the amount of vertices in the resulting CT. Then, comparing this to the amount of vertices in the input trajectories, this pair of input/output vertex count is plotted in a scatterplot. Then, after fitting a curve to the data, a conclusion can be drawn about the output complexity of the algorithm. This experiment has been conducted using three different values of epsilon: $\epsilon = 1$, $\epsilon = 10$ and $\epsilon = 100$. These values were chosen such that the complexity could be analyzed at three different orders of magnitude, revealing potentially interesting trends without using too many values of ϵ, which would clutter the resulting graph.
2. The same experiment has been conducted, however this time ϵ was used as the X-axis, instead of the amount of input vertices. Twenty different values of epsilon, ranging linearly from 0 to 100 (or 1000 in the case of Starkey, to capture the moment of convergence), are plotted against three possible Y-values. This experiment reveals the value of epsilon which gives the highest complexity of the end result. The Y-values that have been tried are:
   (a) The amount of Central Trajectory vertices.
   (b) The amount of jumps in the Central Trajectory.
   (c) The amount of Central Trajectory vertices divided by the amount of input vertices (basically, the output:input ratio).

Both experiments were conducted on both synthetic data (generated by TrajGen) and real life data.

Note that the used values of ϵ are very high, likely much higher than values that would be used in practice. (Remember, ϵ is defined in meters, which means, for example, using a value of $\epsilon = 100$ on a vehicle traffic dataset means vehicles are allowed to jump up to 100 meters, which is quite a discontinuity.) The reason for this is that it causes the complexity to converge long before the maximum value of ϵ is reached. This gives an interesting graph, with a clear indication of the "point of diminishing returns", where increasing ϵ does not increase the amount of points/jumps much.

Also note that the experiments count the amount of input and output vertices after applying a very small amount of simplification ($\delta = 0.01$, or 1cm), such that the redundant points added by the elementary resample step and Central Trajectories are removed. This means that all points in the resulting scatterplot that belong to the same cluster have the same Y-value, since they all produce the same Central Trajectory. In the case of experiment 2c (output:input ratio), the input vertex count of a trajectory is set to be equal to the average (simplified) vertex count over all trajectories within its cluster.

4.3.2 Question 2

As stated before, the second research question is:

What is the effect of path-simplification algorithms on the complexity of Central Trajectories?

To find the answer, the following experiment has been conducted. The value of ϵ was fixed to a very high value of $\epsilon = 100$, for the following reasons:

- It allows CT to create a lot of extra edges that were not already present in the input dataset.
- It ensures most of the simplification is applied to the extra edges created by Central Trajectories, not on the edges that were already present in the input dataset. This ensures the resulting graph does not just become a plot of the vertex count of the input dataset, which is not very useful, and it also ensures there is a significant difference between the different simplification modes.

After running CT, one out of four different simplification methods is applied:

1. No simplification.
2. Simplification of the input trajectories *before* applying Central Trajectories.
3. Simplification of the output, *after* applying Central Trajectories.
4. A combination of 2 and 3.

As in question 1, the amount of output vertices has been compared to the amount of input vertices, which is then plotted in a graph. Then, fitting a curve to this graph reveals the complexity of the output.

Again, a very small amount of simplification ($\delta = 0.01$, or 1cm) is applied on all trajectories (both input and output) before determining the vertex count, regardless of the actual simplification mode. This removes the redundant points added by the elementary resample step and Central Trajectories.

The experiment was conducted on both synthetic data (generated by TrajGen) and real life data. On top of that, every experiment is repeated three times, each with a different value of δ: 1, 10 and 100. (Remember, δ indicates simplification strength.)

As a sidenote: it may seem tempting to scale ϵ with the spatial range of the dataset. As mentioned in Table 1, the spatial range varies wildly between datasets, ranging from only 1km by 2km to up to 407km by 9000km. Using bigger values of ϵ for the bigger datasets seems tempting, but there are some problems with this:

- There is no clear way to make ϵ dependent on both the width and height of the dataset. Should the total area be used? Or maybe the length of the diagonal? Or maybe the minimum/maximum of the width and height? Not a single choice will work across all datasets.
- ϵ is defined in meters, and since some datasets are spatially huge, ϵ could take on unrealistically high values in these cases. As mentioned before, a dataset consisting of vehicle trajectories which contain discontinuities that jump beyond 1000 meters is highly undesirable. An important thing to realize is that ϵ should be considered an absolute measure, not a relative one. That means, regardless of whether the dataset is 1km by 1km or 10,000km by 10,000km, entities should *never* be able to jump these huge gaps.

5 Results

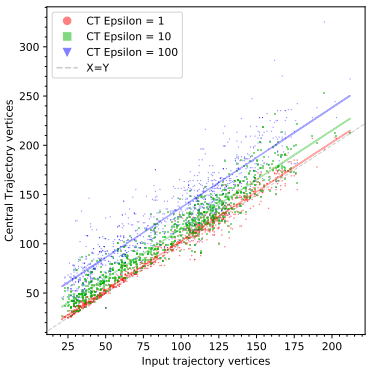
This section will describe the results of the conducted experiments. The experiments were run on a computer with an Intel i5 4670k CPU @ 3.4GHz with 32GB RAM.

5.1 Question 1

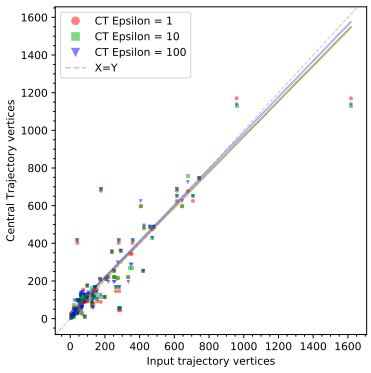
As mentioned before, to answer the first research question, two experiments were conducted. They will be described in the following subsections.

5.1.1 Experiment 1

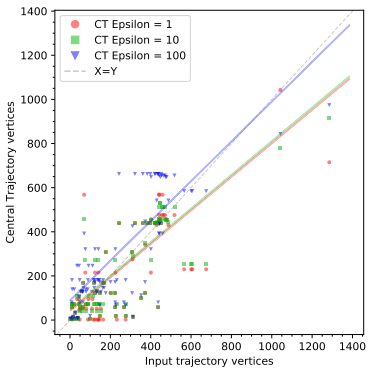
The results of this experiment can be found in Figure 9 on the following page. As you can see, there is a scatterplot of input vertices against Central Trajectory vertices for every dataset. Every point in the scatterplot represents a trajectory, with their color indicating the value of ϵ: either red for 1, green for 10 or blue for 100. Additionally, since the data appeared to be linear, a line was fitted through the points for every value of ϵ, using a least squares fit.



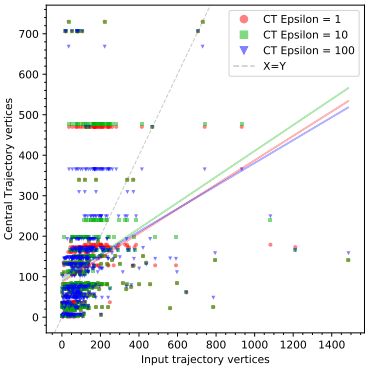
(a) Using synthetic data with 1000 clusters.



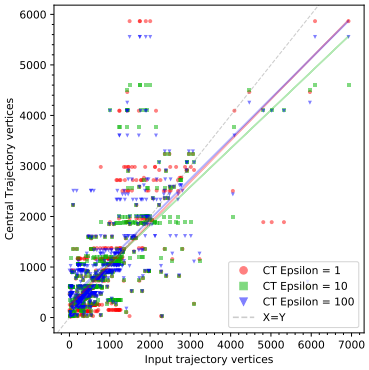
(b) Using SmallDutch with 65 clusters.



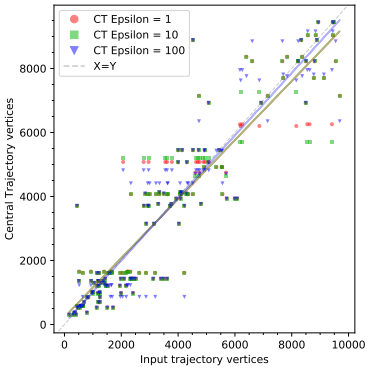
(c) Using Starkey with 30 clusters.



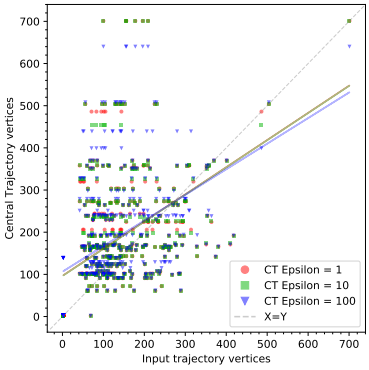
(d) Using T-Drive with 42 clusters.



(e) Using Geolife with 71 clusters.



(f) Using MarineCadastre with 39 clusters.



(g) Using OpenPFLOW with 47 clusters.

Figure 9: A scatterplot of input vertices against Central Trajectory vertices, on seven different datasets. The lines are fitted through the data using a least squares fit. The gray dotted line represents the line $X = Y$, for easy comparison.

5.1.2 Experiment 2

The results of this experiment can be found in Figure 10 on the next page (containing the smaller datasets) and Figure 12 on page 38 (containing the larger datasets). As you can see, the graphs plot ϵ against Central Trajectory vertices, jumps and input:output ratio on all datasets. This means there are three graphs per dataset. Every transparent orange line in the plot represents a cluster. The average value of all the transparent lines is shown as a thicker orange line.

5.2 Question 2

The results of this experiment can be found in Figure 11 on page 29 (containing the smaller datasets) and Figure 13 on page 39 (containing the larger datasets). The graphs plot the amount of input trajectory vertices compared to the amount of Central Trajectory output vertices, while trying four different simplification modes, indicated by different colors. Additionally, three different values of δ are tried, on all datasets. Every point in the scatterplot represents a trajectory, and lines are fitted through the data for every simplification mode using a least squares fit.

6 Discussion

This section will discuss the results.

6.1 Question 1

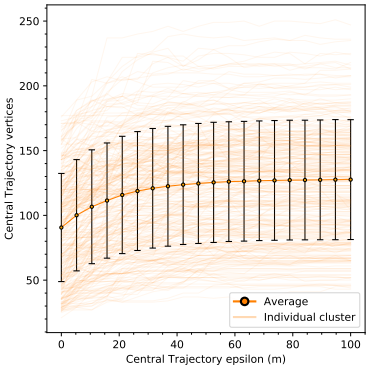
6.1.1 Experiment 1

Looking at the fitted line in Figure 9a on the previous page (the synthetic case), the gathered data appears to be linear in the amount of input vertices, with a slight vertical shift occurring when the value of ϵ is increased. That means in general, the higher ϵ, the higher the complexity of the output. This makes sense, since a higher ϵ allows the Central Trajectory to jump more often between the trajectories in the cluster, and every jump incurs at least two extra vertices: one when leaving the first trajectory, and one when entering the second trajectory. (However, as we will see, in some pathological cases, CT might actually *decrease* the complexity of the input trajectories.)

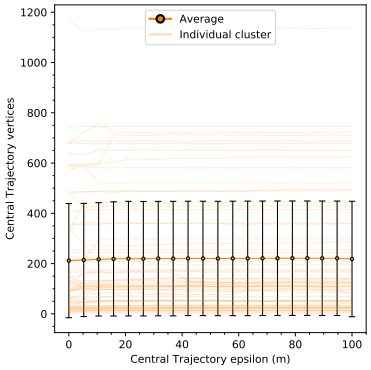
An interesting question raises when comparing the three fitted lines: why are they parallel, and why do they not go through the origin? In other words, why do the lines differ additively, not multiplicatively? The most likely reason for this effect is the variation of the sampling rate of the synthetic dataset. Remember, the sampling rate is randomly varied per trajectory. That means, some trajectories have more points than others, even though their geometry is mostly the same, since the trajectories follow mostly similar paths. If you feed a trajectory into Central Trajectories, increase the sampling rate of the trajectory and feed it into Central Trajectories again, the amount of jumps in the CT does not increase the second time, even though the amount of vertices did. This is because a CT is allowed to jump multiple times on a single edge, so it does not require a lot of vertices to perform a lot of jumps. However, as stated before, increasing ϵ *does* increase the amount of jumps, thus the amount of output vertices. That means, for this particular dataset, the amount of jumps is not correlated to the amount of input vertices, so the amount of extra vertices added by CT remains constant for a given value of ϵ. This is reflected in the shifted lines: the higher ϵ, the higher the shift on the Y-axis.

When looking at the real-world case in Figure 9b (SmallDutch), the data appears to be closer to the line $X = Y$, with small but noticeable differences between the three fitted lines. Changing ϵ in this case has a small effect. Upon closer inspection, the three lines appear to be less parallel than the synthetic dataset, which means the sampling rate is more consistent. This implies that if a point has a higher amount of vertices, it must be a longer, more complicated trajectory, which tends to jump more often.

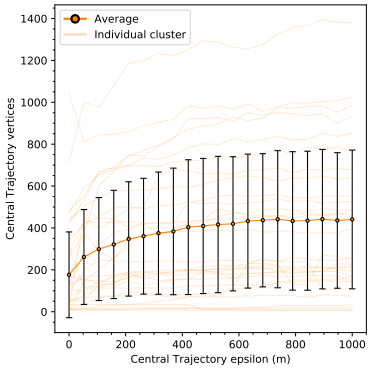
In Figure 9c (Starkey), it is apparent that applying Central Trajectories to the Starkey dataset has a more pronounced effect on the complexity compared to SmallDutch. As you can see, increasing ϵ



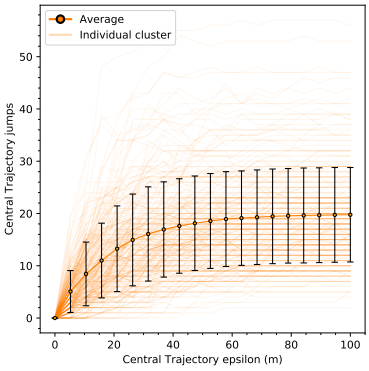
(a) Plotting Central Trajectory vertices on synthetic data, with 1000 clusters.



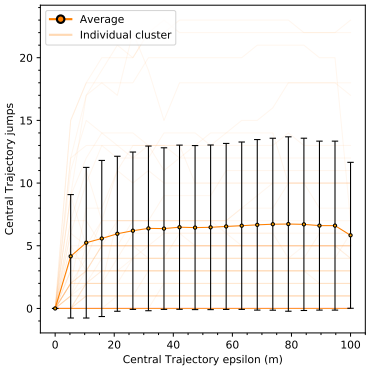
(b) Plotting Central Trajectory vertices on SmallDutch, with 65 clusters.



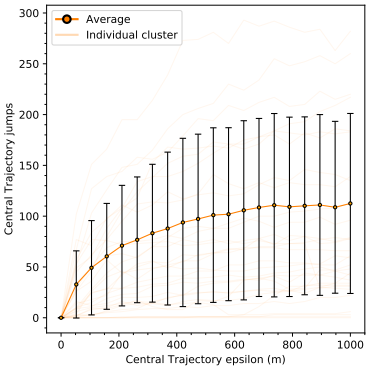
(c) Plotting Central Trajectory vertices on Starkey, with 30 clusters.



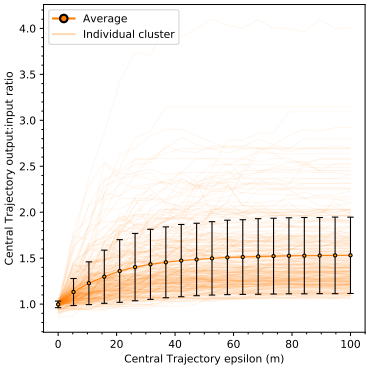
(d) Plotting Central Trajectory jumps on synthetic data, with 1000 clusters.



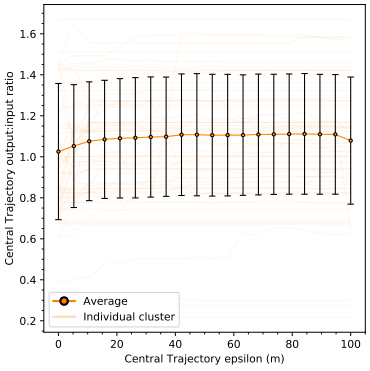
(e) Plotting Central Trajectory jumps on SmallDutch, with 65 clusters.



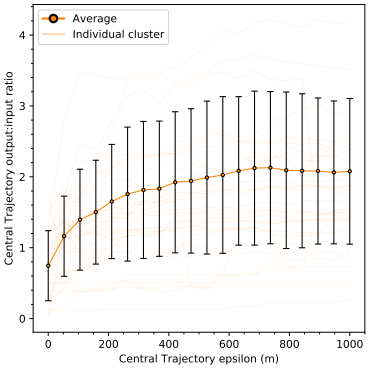
(f) Plotting Central Trajectory jumps on Starkey, with 30 clusters.



(g) Plotting output:input ratio on synthetic data, with 1000 clusters.

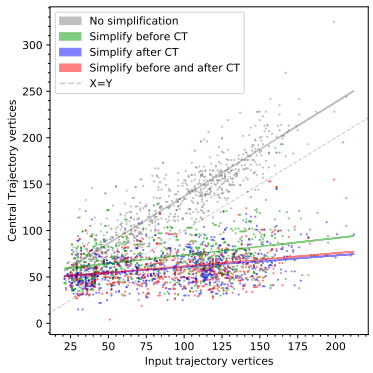


(h) Plotting output:input ratio on SmallDutch, with 65 clusters.

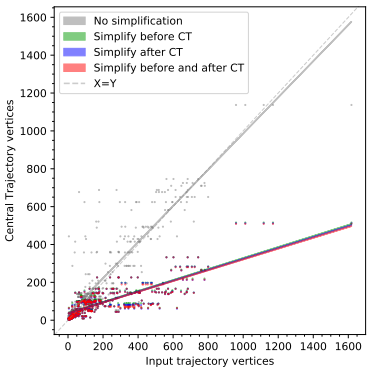


(i) Plotting output:input ratio on Starkey, with 30 clusters.

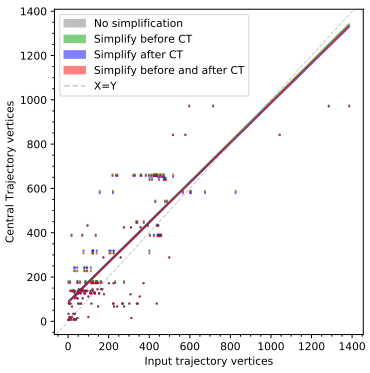
Figure 10: A plot of many clusters, plotting ϵ versus various statistics, on three datasets. The average of all clusters is shown as a thicker orange line. The black bars represent error bars, indicating one standard deviation. Note that the Starkey experiment uses a different range of ϵ: it ranges from 0...1000 instead of 0...100, since it does not converge within the 0...100 range.



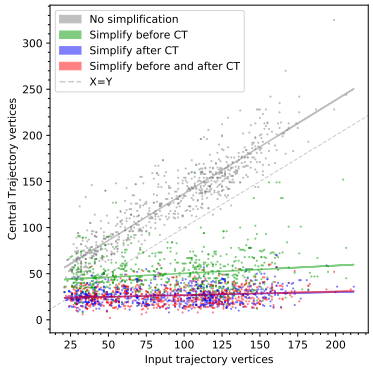
(a) Using synthetic data with 1000 clusters and $\delta = 1$.



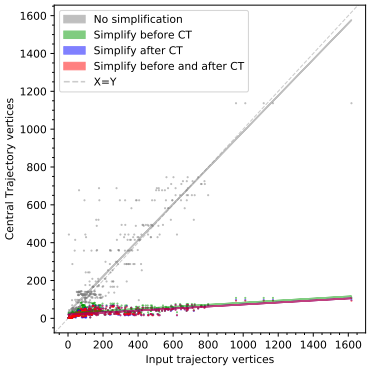
(b) Using SmallDutch with 65 clusters and $\delta = 1$.



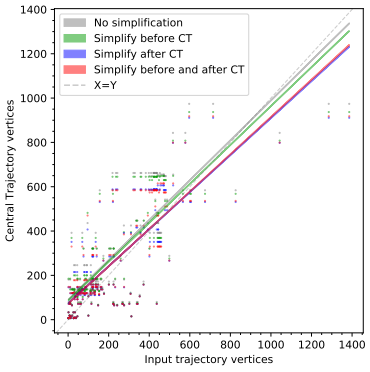
(c) Using Starkey with 30 clusters and $\delta = 1$.



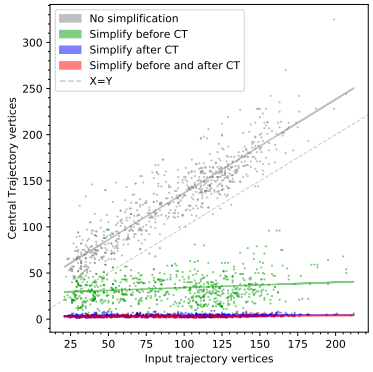
(d) Using synthetic data with 1000 clusters and $\delta = 10$.



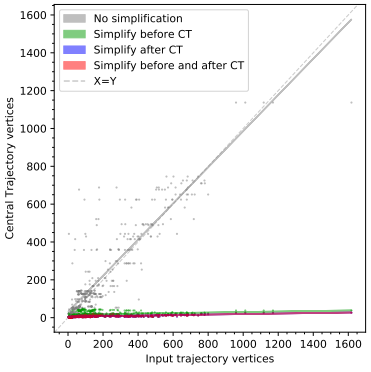
(e) Using SmallDutch with 65 clusters and $\delta = 10$.



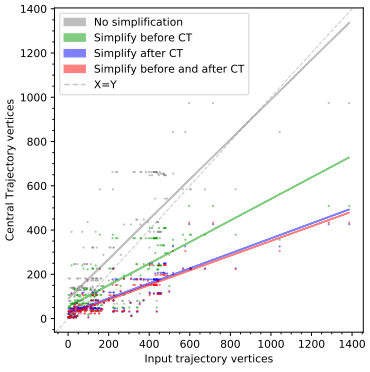
(f) Using Starkey with 30 clusters and $\delta = 10$.



(g) Using synthetic data with 1000 clusters and $\delta = 100$.



(h) Using SmallDutch with 65 clusters and $\delta = 100$.



(i) Using Starkey with 30 clusters and $\delta = 100$.

Figure 11: A scatterplot of input vertices against Central Trajectory vertices, trying four different simplification modes, three different values of δ, and three datasets. The lines are fitted through the data using least squares. The gray dotted line represents the line $X = Y$, for easy comparison.

increases the amount of vertices quite significantly, especially compared to the other two datasets. For example, looking at the highest cluster at the top-right, increasing ϵ brings the amount of vertices from 1050 up to 1300. Additionally, it appears the lines are not parallel. This means, again, that (nearly) all trajectories in the dataset have the same sampling rate.

Looking at Figure 9d (T-Drive), an interesting anomaly appears that only occurs in one other dataset (OpenPFLOW): the blue line ($\epsilon = 100$) actually lies *below* the other lines. A possible explanation is a follows. The dataset may contain some simpler trajectories that are outliers, but which are still somewhat central in a bigger cluster: this is a possibility, since the cluster threshold is large (9000), which implies trajectories within a cluster may be quite far apart. Since they are outliers, a large value of $\epsilon = 100$ is required to be able to make the jump. However, as soon as the jump is made, the amount of vertices drops significantly, since these outliers may have much less vertices than the other trajectories in the cluster.

Looking at Figure 9e (Geolife), another interesting anomaly appears: the green line ($\epsilon = 10$) lies *below* the other two lines. Again, this could be caused by the large clustering threshold, or by statistical error due to wildly varying point counts.

Looking at Figure 9f (MarineCadastre), it appears the general trends are similar to the other graphs. Regardless of the distribution of points, the fitted lines appear to correspond nicely with the expected results.

Looking at Figure 9g (OpenPFLOW), the blue line appears below the others, similarly to T-Drive. Again, this could be caused by the large clustering threshold, or simply due to statistical error: for example, looking beyond the $x = 450$ range, there are only 3 clusters: their point distribution has a large effect on the fitted line, since they are the only ones that contain such a large amount of vertices. Additionally, another interesting anomaly is the fact that the vast majority of the trajectories have more than 40 input points, but there is a single cluster that has only a handful of points (the blue points all the way on the left).

Comparing Figure 9a to the other graphs reveals that the graph of the synthetic dataset is very different compared to all other datasets, since it is the only one in which the three lines appear to be exactly parallel. This indicates the synthetic dataset might actually be biased in some way, making it less realistic. This will be discussed further in Section 8.3.

6.1.2 Experiment 2

Looking at Figure 10 (the top row of graphs), it is apparent that increasing ϵ increases the amount of vertices, as mentioned before. However, it is interesting to see the different convergence behavior across datasets. For instance, the synthetic dataset (Figure 10a) initially has a strong response to the increasing of ϵ, however, around the $\epsilon = 50$ mark, the amount of vertices appears to converge – increasing ϵ beyond this value has very little effect. Looking at the graph below it (Figure 10d), which plots the amount of jumps instead of the amount of vertices, explains why this happens: the amount of jumps converges at roughly the same moment. And remember, every jump adds two new vertices, so the amount of jumps largely determines the amount of vertices in the output. The reason the amount of jumps converges is likely because the distance between two trajectories in a cluster is limited, and if ϵ goes beyond this limit, no new jumps will occur, since the Central Trajectory can already jump to any trajectory it wants to within the cluster. Essentially, there are no new opportunities for jumps when ϵ takes on a higher value.

When looking at Figure 10b (SmallDutch), the effect of increasing ϵ is much less pronounced. The average amount of vertices stays within the 200...220 range. This phenomenon can be easily explained by looking at the graph below it (Figure 10e), which plots the amount of jumps instead of the amount of vertices. It is apparent that the amount of jumps is very low, converging at $\epsilon = 40$ to an average value of about 6. This explains why the amount of vertices does not increase much: since every jump adds two new vertices, performing only 6 jumps on average means there are 12 new vertices on average per cluster: a small increase, compared to the average vertex count of 200. (Note that the average decrease in the amount of jumps at $\epsilon = 100$ is caused by an outlier: looking closely at the transparent lines reveals that one of them drops significantly around this point, causing the average to drop as well. This outlier can be seen in the graph below it (Figure 10h) as well.)

When looking at Figure 10c (Starkey), the graph is completely different again. (But note that ϵ now goes up to 1000 instead of 100, to capture the moment of convergence.) The amount of vertices steadily increases up to about $\epsilon = 700$, where it converges to an average value of about 410. Again, looking at the graph below it (Figure 10f) reveals the cause: the amount of jumps converges at around the same value of ϵ. Interestingly, it seems the amount of vertices rises much more strongly compared to the previous two datasets (synthetic and SmallDutch): it ranges from 180 all the way up to 400, more than double the amount of vertices. Again, this is explained by the amount of jumps, which lies much higher than the other two datasets: on average, it can go up to 100, compared to only 19 and 6 for the synthetic dataset and SmallDutch, respectively. And some outliers in the Starkey dataset jump up to 290 times, which is the highest jump count among all analyzed clusters in all datasets.

Since Starkey has a smaller spatial range than SmallDutch (only 8km by 14km, compared to 47km by 46km for SmallDutch), it makes sense that it would produce more jumps than SmallDutch. However, even when using an identical value of $\epsilon = 100$ on both datasets, Starkey gives 50 jumps on average, compared to 6 for SmallDutch. This probably means there are much more trajectories per cluster, which makes sense, since the trajectories lie closer together, so more of them will end up in the same cluster.

Analyzing the larger datasets, contained within Figure 12 on page 38, reveals some interesting results. Among all datasets, the amount of jumps increases when increasing ϵ, as expected. But in some cases, the amount of vertices actually *decreases*. (This is also visible in the output:input ratio graphs in the bottom row, where the ratio drops below 1.) As mentioned before, the amount of vertices in a CT can decrease while increasing ϵ if the CT happens to jump to outlier trajectories that have a lower vertex count than the others, within the same cluster. Since they are outliers, it is not possible to jump to them with a low ϵ, because the required jump distance is simply too big. Another possible explanation is that the CT jumps more often to trajectories that have simple geometry (such as straight lines or slight curves), which means they contain points that are easily simplified away (remember, a tiny amount of simplification is applied before counting the vertices).

Even among the bigger datasets, the amount of jumps varies significantly. The average amount of jumps appears to range from 8 (MarineCadastre, Figure 12g) to 50 (T-Drive, Figure 12e). However, looking at the transparent lines, it is apparent that the amount of jumps varies largely even within a dataset: for example, in the case of Geolife (Figure 12f), one cluster jumps up to 110 times, even when epsilon takes on a relatively small value of $\epsilon = 50$. Another indication of this is the fact that the error bars, which indicate one standard deviation, are generally very large.

There is one interesting anomaly that appears only in Figure 12l (OpenPFLOW). Since many of the trajectories in this dataset have an (almost) equal amount of vertices, many of the transparent lines coincide with the line $y = 1$. Additionally, there are some extreme outliers visible in the graph. These outliers are caused by trajectories in the dataset that have a very large amount of redundant points. For example, after simplification, some trajectories only retain 2 or 3 points. Applying CT on a cluster that contains a trajectory like this will cause the output:input ratio to shoot up dramatically, since the average amount of CT vertices for this dataset lies around 200. This gives an output:input ratio of $\frac{200}{2} = 100$, which can be seen in the graph in the form of a huge outlier.

6.2 Question 2

Looking at the global trends in Figure 11 on page 29, it is apparent that applying Central Trajectories with a high value of ϵ, without any form of simplification, increases the complexity of the trajectories. (Note that the X- and Y-scales vary quite a lot: use the dashed line, indicating the line $x = y$, as reference.) This increase was already revealed in Figure 9 on page 26. However, some interesting patterns start to arise when multiple methods of simplification are applied. For instance, in general, applying simplification before CT (the green line) appears to give a smaller reduction in the complexity compared to applying simplification after CT (or both), indicated by the red and blue lines. However, the difference depends highly on the dataset. For example, in the synthetic case (leftmost column of plots in Figure 11), the difference between simplification before and after CT appears to range from 10 to 30 vertices on average. However, looking at SmallDutch (central column of plots), the difference appears to be much smaller, even when accounting for the different Y-scale on the graph. Finally,

when looking at Starkey (rightmost column of plots), the average difference in the amount of vertices suddenly becomes huge, ranging up to 300.

Additionally, increasing δ decreases the complexity, as expected, but the degree in which the slope of the complexity changes appears to vary between datasets. For instance, looking at the rightmost column of plots of Figure 11 (Starkey), the slope of the green "Simply before CT" line does not change much when increasing δ from 1 to 10. However, looking at the same trend in the central column of plots (SmallDutch), the slope of the green line goes down dramatically when increasing δ from 1 to 10. This is due to the fact that the input trajectories in Starkey are much longer and more angular than SmallDutch, even though the area covered by the trajectories in Starkey much smaller: only 8km by 14km, compared to 47km by 46km for SmallDutch. Additionally, both datasets appear to have roughly the same amount of vertices per trajectory (the X- and Y-scale of both graphs are comparable). Since SmallDutch has a much higher sampling rate, its trajectories must be much shorter and simpler, which means simplification has a much stronger effect on them.

The average amount of CT jumps for Starkey can be deduced by looking at Figure 10f on page 28 and reading the value at $\epsilon = 100$ (since this value of ϵ is used for all simplification experiments): in this case it appears to be 50. This means, on average, applying CT with $\epsilon = 100$ on Starkey increases the vertex count by about $2 \times 50 = 100$ (without taking simplification into account). This can be clearly seen in Figure 11c, since all lines appear to intersect the X-axis at about $y = 100$. However, when the amount of input vertices increases, the difference becomes smaller and smaller until about $x = 800$, after which the difference becomes negative, and applying CT actually *decreases* the amount of vertices slightly. I have previously mentioned some possible explanations for this, but in this case it may actually be a statistical error, since there are only three clusters that have more than 800 vertices. This could have been mitigated if the Starkey dataset was bigger: it contains only 250 trajectories.

Looking at the larger datasets, contained within Figure 13 on page 39, reveals some interesting patterns. The central two columns of plots (Geolife and MarineCadastre) follow the same pattern: simplification after CT is slightly better than simplification before CT. Applying simplification both before and after CT has roughly the same effect as only applying simplification afterwards. However, the differences between the simplification modes are very small.

Two datasets, however, appear to deviate from the norm. The first one is T-Drive (leftmost column of plots in Figure 13). In this case, the red line lies below the green line, which lies below the blue line. This indicates simplification before and after CT is better than the other simplification modes, which was already established, but the green and blue lines are now swapped. That means, simplification before CT is now *better* than simplification after CT. This may be due to the fact that T-Drive generally has a high amount of jumps, as seen in Figure 12e on page 38.

The second defiant dataset is OpenPFLOW (rightmost column of plots in Figure 13). In this case, the blue line now lies below the red line, which lies below the green line. In other words: applying simplification after CT is better than simplification both before and after CT, which seems counter-intuitive. Additionally, simplification before and after CT is better than only applying simplification before CT: this follows the trends of the other datasets. The former observation could be explained either by statistical error, or by geometric traits that differ significantly from the other datasets.

7 Conclusion

When looking at the results, it is clear that the complexity of Central Trajectories is linear when applied to real-world data, even though its theoretical upper bound is slightly above quadratic, as mentioned before: $O(nm^{5/2})$, where n is the amount of vertices per trajectory and m is the amount of trajectories. That means, the pathological case given in the paper by van Kreveld et al. [2], which triggers the quadratic behavior, does not seem to appear in real world scenarios.

Additionally, the effect of ϵ appears to be strongly dependent on the dataset used. As mentioned before, increasing ϵ on the Starkey dataset has a huge effect, with the amount of jumps and vertices increasing dramatically. However, doing the same thing on the SmallDutch dataset has only a very minor effect. The general trend appears to be that every single dataset produces an unique graph,

which could be called a *response curve*, for various values of ϵ. This makes sense, since there are lots of parameters of the dataset that have an effect on the Central Trajectory vertex count. To name a few: trajectory amount, length, sampling rate, spatial range, timespan, cluster amount, cluster size, et cetera.

Finally, the effect of simplification is quite significant, even when using small values of δ. This is an indication Central Trajectories adds a lot of redundant vertices, which are easily simplified away, since they are usually collinear with other points on the same trajectory. However, as with ϵ, the effect of increasing δ depends strongly on the dataset used. For example, with small values of δ, there are two datasets that do not respond strongly at all: Starkey and T-Drive, since they have a low sample rate, which means their trajectories are more angular and thus more difficult to simplify.

Generally speaking, applying simplification both before and after CT gives the best results, and applying simplification before CT gives the worst results (but not all datasets follow this trend). However, in the latter case, the reduction in complexity is still quite impressive. The results indicate it may be beneficial to store a database of simplified trajectories when doing trajectory analysis, instead of storing the originals. Then, since the simplified trajectories have such a low amount of vertices, they are much easier to work with, and running CT on them only takes a fraction of a second. In addition, the resulting CT will also have a low complexity, making it easier to work with as well.

8 Future work

This section will describe possible future work, and some things that were originally going to be part of the research project but were canceled due to lack of time or other resources. They provide a interesting opportunity for future endeavors.

8.1 Research question 3

Originally, there were three different research questions. One of the research questions, "*What is the performance of mean trajectories compared to central trajectories for different kinds of data?*", was removed. The reason is as follows.

The experiment needed to answer this question required inventing a certain performance measure (meaning: the quality of the generated output, not computational performance). This measure would be based not only on the amount of vertices, but also on the degree in which it can avoid obstacles and capture the unique defining features of the clustered trajectories. Checking whether a trajectory intersects an obstacle is fairly trivial (it can be done with basic collision detection), however, in my research, I have not found a single trajectory dataset that also provides contextual data, such as nearby obstacles. Using TrajGen in combination with its generated obstacles would work, however it would mean the experiment could only be run on synthetic data. Additionally, the degree in which unique defining features of a trajectory are captured is not exactly trivial to determine computationally. Humans are good at extracting these features, but for computers, this poses an algorithmic challenge.

8.2 Effect of trajectory count (m)

Another question that my research does not answer is the effect of trying various values of m – the amount of trajectories – on the complexity of Central Trajectories. In theory, we know the effect can be slightly above quadratic (remember, the theoretical bound is $O(nm^{5/2})$), but it would be interesting to see what would happen in practice. The research could be conducted by, for example, using random subsets of the clusters and making a scatterplot of cluster size on the X-axis, versus Central Trajectory vertex count on the Y-axis. Then, fitting a polynomial to this graph could reveal its real-world complexity.

8.3 Synthetic data bias

As mentioned before, some of the graphs based on the experiments conducted on the synthetic dataset are very different compared to all other graphs. This means there must be some kind of bias in this dataset, making it unlike the real datasets. A possible explanation is the fact that the generated trajectories are simulated using a Verlet physics engine, taking acceleration and momentum into account. The problem with this is that the entities constantly accelerate towards a "goal" trajectory, which makes them oscillate around it randomly. This oscillation causes a large change in velocity over time, going up and down like a rollercoaster, which means the trajectories have a very non-linear distribution of points.

Additionally, entities that are further away from the goal trajectory are usually travelling at higher speeds than those that are near the goal trajectory, since the acceleration towards it is proportional to the distance from it. This results in trajectories that have a high point density near the center of the cluster, but a low point density near the outskirts of the cluster.

Conclusively, the trajectories do not follow the behavior displayed by most entities in real life, even though the laws of physics are accurately simulated. A possible solution would involve some tweaks in the physical parameters to ensure the trajectories have a more constant speed, regardless of their distance to the target trajectory.

Another possible explanation is the fact that the synthetic dataset has a much higher amount of clusters. It may be the case that if the other datasets had more clusters to use, the results would be more consistent.

8.4 Accuracy of vertex counting

In the experiments, the amount of vertices of a trajectory is determined by using a slightly simplified version of the trajectory ($\delta = 0.01$), and then counting those vertices. This avoids counting redundant vertices in the output, which are added by various algorithms such as elementary resampling, Central Trajectories itself, etc. However, this method is not entirely accurate. For example, imagine a trajectory that has a sample rate that is constant not in time, but in space. That means, for example, its vertices are always 10 meters apart, but the time difference can vary. Now imagine this trajectory speeding up over time, while it traverses in a straight line. The simplification step would remove all vertices except the first and last one, even though there is lots of valuable time information on the line itself. This time information is thrown away, which might not be desirable, especially since it cannot be easily reconstructed in the case the time information was not linear (which, in the case of an accelerating trajectory, it is definitely not). In other words, the simplification step throws non-linear time information away.

A better approach would keep track of the manner in which a vertex is added. It could distinguish between original vertices, vertices added by resampling, vertices added by a Central Trajectory jump, vertices added by a Central Trajectory Reeb edge change, et cetera. This would allow more accurate vertex counts, without any kind of simplification. This was already partially implemented, but the results were not perfect due to the inherent difficulty in keeping track of the vertex origins while applying the various (preprocessing) steps in the Central Trajectory pipeline. Thus, it was decided to use the simplification method instead, which was much easier to implement, at the cost of a slightly less accurate vertex count.

8.5 Performance measures

Since some of the used datasets are quite large, some preprocessing steps and experiments take quite a while to complete. It would be interesting to measure the performance of some algorithms, such as Central Trajectories, on the various datasets, comparing the dataset size to the amount of time it takes to generate a CT. Comparing this with research question 2, it would also be interesting to see the effect of simplification before CT on the running time of CT. According to van Kreveld et al. [2], the theoretical bound on the running time is $O(nm^3)$, where n is the amount of points per trajectory

and m is the amount of trajectories. That means, theoretically, increasing the amount of trajectories should have a much stronger effect on the running time, compared to increasing the amount of points per trajectory.

References

[1] Takehiro Kashiyama, Yanbo Pang, and Yoshihide Sekimoto. Open pflow: Creation and evaluation of an open dataset for typical people mass movement in urban areas. *Transportation research part C: emerging technologies*, 85:249–267, 2017.

[2] Marc van Kreveld, Maarten Löffler, and Frank Staals. Central trajectories. *arXiv preprint arXiv:1501.01822*, 2015.

[3] Frank Staals. *Geometric Algorithms for Trajectory Analysis*. PhD thesis, Utrecht University, 2015.

[4] Maximilian Peter Konzack. Trajectory analysis: bridging algorithms and visualization. *IPA Dissertation Series*, 2018, 2018.

[5] Kevin Buchin, Maike Buchin, Marc van Kreveld, Bettina Speckmann, and Frank Staals. Trajectory grouping structure. *Journal of Computational Geometry*, 6(1):75–98, 2015. ISSN 1920-180X. doi: 10.20382/jocg.v6i1a3.

[6] Marc van Kreveld, Maarten Löffler, Frank Staals, and Lionov Wiratma. A refined definition for groups of moving entities and its computation. *International Journal of Computational Geometry & Applications*, 28(02):181–196, 2018. doi: 10.1142/S0218195918600051.

[7] Arthur Van Goethem, Marc Van Kreveld, Maarten Löffler, Bettina Speckmann, and Frank Staals. Grouping time-varying data for interactive exploration. *arXiv preprint arXiv:1603.06252*, 2016.

[8] Jae-Gil Lee, Jiawei Han, and Kyu-Young Whang. Trajectory clustering: a partition-and-group framework. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 593–604. ACM, 2007.

[9] G. Andrienko, N. Andrienko, G. Fuchs, and J. M. C. Garcia. Clustering trajectories by relevant parts for air traffic analysis. *IEEE Transactions on Visualization and Computer Graphics*, 24(1): 34–44, Jan 2018. ISSN 1077-2626. doi: 10.1109/TVCG.2017.2744322.

[10] N. Andrienko, G. Andrienko, J. M. C. Garcia, and D. Scarlatti. Analysis of flight variability: a systematic approach. *IEEE Transactions on Visualization and Computer Graphics*, 25(1):54–64, Jan 2019. ISSN 1077-2626. doi: 10.1109/TVCG.2018.2864811.

[11] Maxime Gariel, Ashok N Srivastava, and Eric Feron. Trajectory clustering and an application to airspace monitoring. *IEEE Transactions on Intelligent Transportation Systems*, 12(4):1511–1524, 2011.

[12] Kevin Buchin, Maike Buchin, Joachim Gudmundsson, Maarten Löffler, and Jun Luo. Detecting commuting patterns by clustering subtrajectories. In Seok-Hee Hong, Hiroshi Nagamochi, and Takuro Fukunaga, editors, *Algorithms and Computation*, pages 644–655, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-92182-0.

[13] Scott Gaffney and Padhraic Smyth. Trajectory clustering with mixtures of regression models. In *KDD*, volume 99, pages 63–72, 1999.

[14] Scott J Gaffney, Andrew W Robertson, Padhraic Smyth, Suzana J Camargo, and Michael Ghil. Probabilistic clustering of extratropical cyclones using regression mixture models. *Climate dynamics*, 29(4):423–440, 2007.

[15] Zhouyu Fu, Weiming Hu, and Tieniu Tan. Similarity based vehicle trajectory clustering and anomaly detection. In *IEEE International Conference on Image Processing 2005*, volume 2, pages II–602. IEEE, 2005.

[16] Jessica Lin, Eamonn Keogh, Stefano Lonardi, and Bill Chiu. A symbolic representation of time series, with implications for streaming algorithms. In *Proceedings of the 8th ACM SIGMOD workshop on Research issues in data mining and knowledge discovery*, pages 2–11. ACM, 2003.

[17] Woojin Kim and Facundo Mémoli. Formigrams: Clustering summaries of dynamic data. In *Conference on Computational Geometry (CCCG 2018)*, page 180, 2018.

[18] Zhang Zhang, Kaiqi Huang, Tieniu Tan, et al. Comparison of similarity measures for trajectory clustering in outdoor surveillance scenes. In *ICPR (3)*, pages 1135–1138. Citeseer, 2006.

[19] M. Vlachos, G. Kollios, and D. Gunopulos. Discovering similar multidimensional trajectories. In *Proceedings 18th International Conference on Data Engineering*, pages 673–684, Feb 2002. doi: 10.1109/ICDE.2002.994784.

[20] Lionov Wiratma, Marc van Kreveld, and Maarten Löffler. On measures for groups of trajectories. In Arnold Bregt, Tapani Sarjakoski, Ron van Lammeren, and Frans Rip, editors, *Societal Geo-innovation*, pages 311–330, Cham, 2017. Springer International Publishing. ISBN 978-3-319-56759-4.

[21] Kevin Buchin, Maike Buchin, Marc Van Kreveld, Maarten Löffler, Rodrigo I Silveira, Carola Wenk, and Lionov Wiratma. Median trajectories. *Algorithmica*, 66(3):595–614, 2013.

[22] Leonard Johard and Emanuele Ruffaldi. Self-organizing trajectories. *Pattern Recognition Letters*, 84:177–184, 2016.

[23] George Gomes, Emanuele Santos, Creto Vidal, Ticiana Coelho da Silva, and Jose Antonio F. Macedo. Real-time discovery of hot routes on trajectory data streams using interactive visualization based on gpu. *Computers & Graphics*, 76, 09 2018. doi: 10.1016/j.cag.2018.09.008.

[24] Jae-Gil Lee, Jiawei Han, Xiaolei Li, and Hector Gonzalez. Traclass: trajectory classification using hierarchical region-based and trajectory-based clustering. *Proceedings of the VLDB Endowment*, 1(1):1081–1094, 2008.

[25] Eleftherios Garyfallidis, Matthew Brett, Marta Morgado Correia, Guy B Williams, and Ian Nimmo-Smith. Quickbundles, a method for tractography simplification. *Frontiers in neuroscience*, 6:175, 2012.

[26] David H Douglas and Thomas K Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *Cartographica: the international journal for geographic information and geovisualization*, 10(2):112–122, 1973.

[27] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.14 edition, 2019. URL `https://doc.cgal.org/4.14/Manual/packages.html`.

[28] Eleftherios Garyfallidis, Matthew Brett, Bagrat Amirbekian, Ariel Rokem, Stefan Van Der Walt, Maxime Descoteaux, and Ian Nimmo-Smith. Dipy, a library for the analysis of diffusion mri data. *Frontiers in neuroinformatics*, 8:8, 2014.

[29] Mary M Rowland. *The Starkey Project: History Facilities, and Data Collection Methods for Ungulate Research*, volume 396. US Department of Agriculture, Forest Service, Pacific Northwest Research Station, 1997.

[30] Jing Yuan, Yu Zheng, Chengyang Zhang, Wenlei Xie, Xing Xie, Guangzhong Sun, and Yan Huang. T-drive: driving directions based on taxi trajectories. In *Proceedings of the 18th SIGSPATIAL International conference on advances in geographic information systems*, pages 99–108, 2010.

[31] Yu Zheng, Xing Xie, Wei-Ying Ma, et al. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

[32] Woojin Kim and Facundo Memoli. Stable persistent homology features of dynamic metric spaces. *arXiv preprint arXiv:1812.00949*, 2018.

[33] Mehmet Ercan Nergiz, Maurizio Atzori, and Yucel Saygin. Towards trajectory anonymization: a generalization-based approach. In *Proceedings of the SIGSPATIAL ACM GIS 2008 International Workshop on Security and Privacy in GIS and LBS*, pages 52–61. ACM, 2008.

[34] Stan Salvador and Philip Chan. Toward accurate dynamic time warping in linear time and space. *Intelligent Data Analysis*, 11(5):561–580, 2007.

Appendix

Extra graphs

To prevent cluttering the thesis with a large amount of graphs, part of the graphs were placed in the Results section, while others were placed in the appendix. See figures 12 and 13.

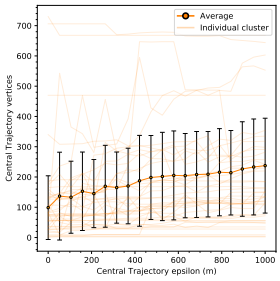
Program implementation details

This section will give some more details about the program that was written for the purpose of this research.

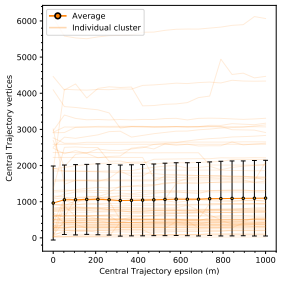
Visualization

Since trajectory data is hard to digest for humans in its pure, textual form, I have implemented a trajectory visualization program. It also aids in debugging the various trajectory algorithms, since it will allow real-time feedback on the generated trajectories. The program has the following features:

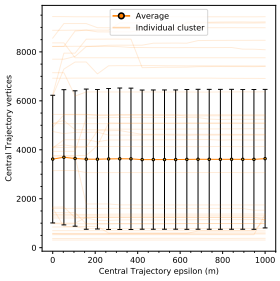
- Loading datasets from a SQLite database or CSV file, while converting latitude and longitude to meters via planar projection.
- Drawing trajectories in a window, allowing zoom and panning control to look around, optionally with a customizable background grid for scale reference.
- Playing back the trajectories like a movie, to analyze the behavior of the entities over time.
- Showing statistics for individual trajectories, such as start and end time, total duration, and average sampling density.
- Showing/hiding certain trajectories, either manually or by filtering on trajectory key, cluster id, total distance travelled or bounding box area.
- Shifting the trajectories in time.
- Applying various clustering algorithms to the visible trajectories, with user-defined parameters. (K-Means and QuickBundles)
- Applying various resampling algorithms to the visible trajectories, with user-defined parameters. (Regular resampling and elementary resampling)
- Applying the Douglas-Peucker simplifying algorithm to the visible trajectories. The simplification amount δ can be changed at will.
- Applying the Central Trajectory algorithm to the visible trajectories, and drawing the corresponding Reeb graph for debugging. The value of ϵ can be changed at will, updating the trajectory in real time.
- Drawing a histogram of the trajectory's points in time, allowing the analysis of the time distribution of the trajectories.



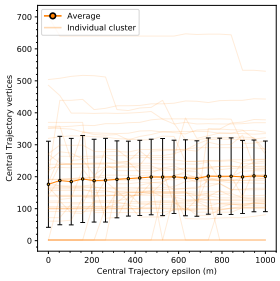
(a) Plotting Central Trajectory vertices on T-Drive, with 42 clusters.



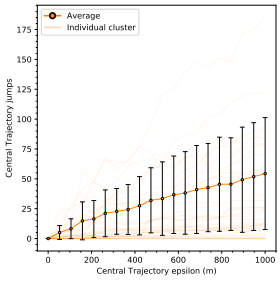
(b) Plotting Central Trajectory vertices on Geolife, with 71 clusters.



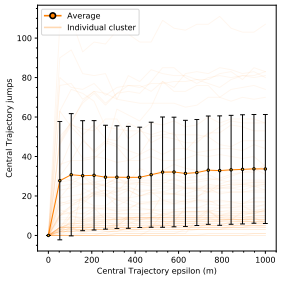
(c) Plotting Central Trajectory vertices on MarineCadastre, with 39 clusters.



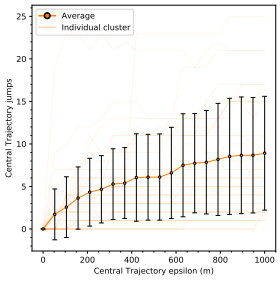
(d) Plotting Central Trajectory vertices on OpenPFLOW, with 47 clusters.



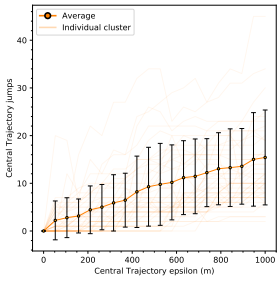
(e) Plotting Central Trajectory jumps on T-Drive, with 42 clusters.



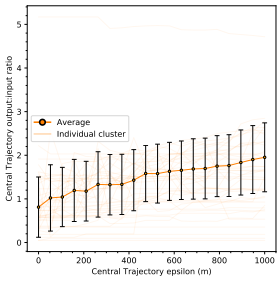
(f) Plotting Central Trajectory jumps on Geolife, with 71 clusters.



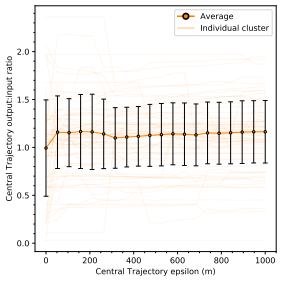
(g) Plotting Central Trajectory jumps on MarineCadastre, with 39 clusters.



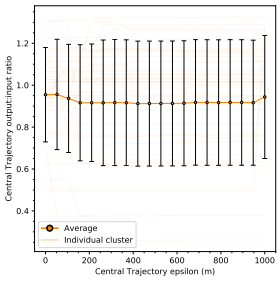
(h) Plotting Central Trajectory jumps on OpenPFLOW, with 47 clusters.



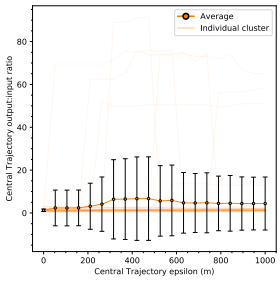
(i) Plotting output:input ratio on T-Drive, with 42 clusters.



(j) Plotting output:input ratio on Geolife, with 71 clusters.

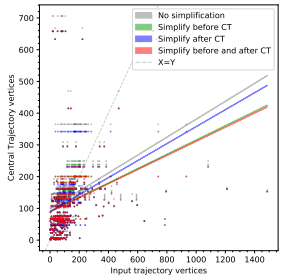


(k) Plotting output:input ratio on MarineCadastre, with 39 clusters.

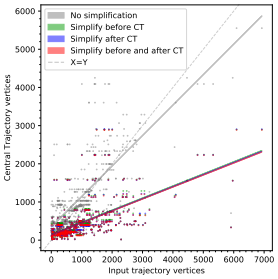


(l) Plotting output:input ratio on OpenPFLOW, with 47 clusters.

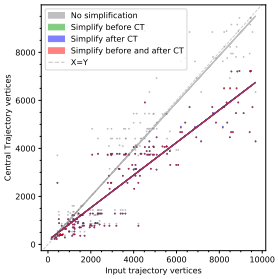
Figure 12: A plot of many clusters, plotting ϵ versus various statistics, similar to Figure 10, on the four largest datasets.



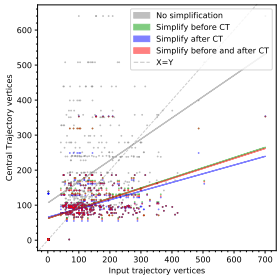
(a) Using T-Drive with 42 clusters and $\delta = 1$.



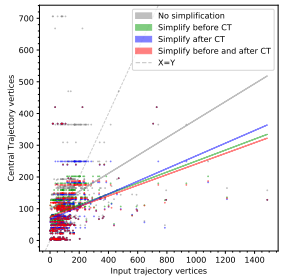
(b) Using Geolife with 71 clusters and $\delta = 1$.



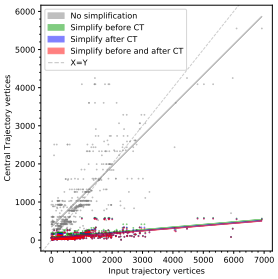
(c) Using MarineCadastre with 39 clusters and $\delta = 1$.



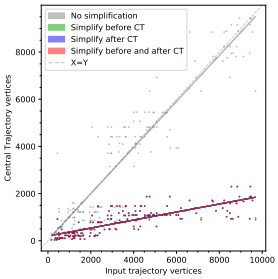
(d) Using OpenPFLOW with 47 clusters and $\delta = 1$.



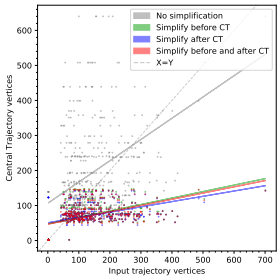
(e) Using T-Drive with 42 clusters and $\delta = 10$.



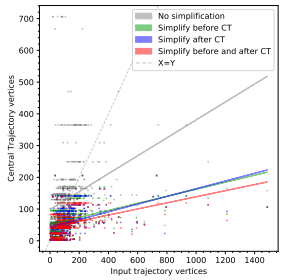
(f) Using Geolife with 71 clusters and $\delta = 10$.



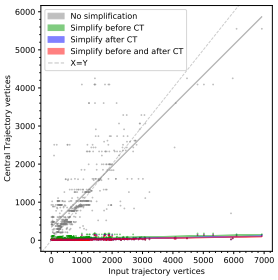
(g) Using MarineCadastre with 39 clusters and $\delta = 10$.



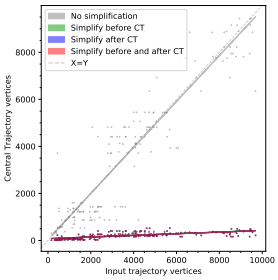
(h) Using OpenPFLOW with 47 clusters and $\delta = 10$.



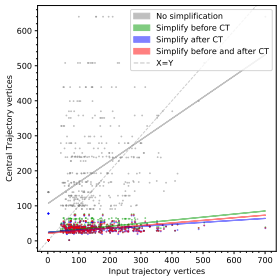
(i) Using T-Drive with 42 clusters and $\delta = 100$.



(j) Using Geolife with 71 clusters and $\delta = 100$.



(k) Using MarineCadastre with 39 clusters and $\delta = 100$.



(l) Using OpenPFLOW with 47 clusters and $\delta = 100$.

Figure 13: A scatterplot of input vertices against Central Trajectory vertices, trying four different simplification modes, three different values of δ, and four datasets, similar to Figure 11.

- Drawing a compass of the trajectory's angles, allowing the analysis of the directional distribution of the trajectories. (Remember, the angle of a trajectory is simply the angle between its first and its last point.)
- Running the experiments described in Section 4.3 on page 23 and writing the resulting data to a SQLite database.

The program uses the following libraries:

- Boost (Boost.Python for the Python interface needed for QuickBundles, and Boost.Graph for the Central Trajectory implementation)
- SFML (for drawing trajectories)
- ImGui (for the graphical user interface)
- LibGeographic, GeoPY, DIPY[28] (for the QuickBundles clustering algorithm)
- OpenMP (for parallelism)
- MoveTK (for trajectory operations such as simplification)
- SQLiteCpp (a C++ wrapper for SQlite, used for loading trajectory data and storing experiment data)
- Fast C++ CSV Parser (for loading CSV files)

Reeb graph visualization

The program can also draw a Reeb graph used in the generated Central Trajectory, showing the Reeb start, end, split and merge vertices, Reeb edges including their weights, and a per-edge graph of centrality over time. An example can be found in Figure 14.

The green vertices at the bottom indicate Reeb start vertices, the red vertices at the top indicate Reeb end vertices, the orange vertices are Reeb split vertices and the blue ones are Reeb merge vertices. The thin gray edges indicate Reeb edges, and the thick white edges indicate Reeb edges on the lowest-weight path from a Reeb start to a Reeb end vertex: they are part of the Central Trajectory.

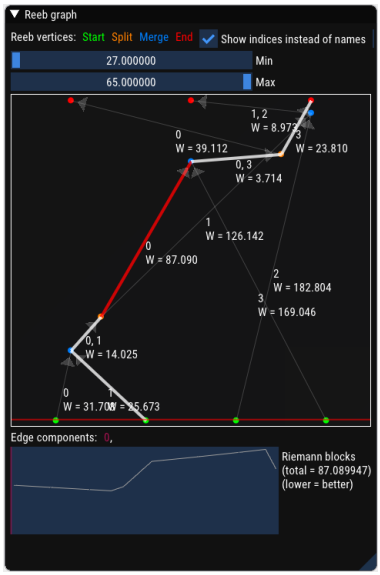
Since the Reeb graph is defined over a period of time, the graph can indicate the current time by a horizontal red line. The user can scrub through time, allowing to see the formation and disbanding of groups over time. As mentioned before, taking a horizontal cross-section of the graph (a single moment in time) reveals the groups currently active at that time.

The $W = ...$ numbers indicate the edge weights, also known as the centrality: the lower it is, the more central the trajectories on this edge are. The number(s) above it indicates the Reeb edge's components, also known as the group of trajectories this edge represents. For example, an edge with component "0, 3" contains the first and the fourth trajectory (since indexing starts from 0).

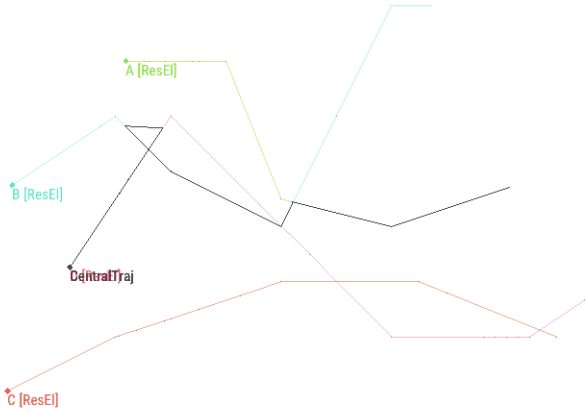
Finally, the user can select an edge by clicking on it, which makes it turn red. Then it will show the edge's centrality over time in the graph at the bottom, in addition to its component. The weight of the edge is equal to the integral of this graph.

Output data

The program contains an Experimentor class, which is responsible for conducting the various experiments described in Section 4.3 on page 23. The class tries a list of predefined combinations of ϵ, δ, simplification mode, et cetera. Then it gathers data, and stores the results in a SQLite database. This database can then be used for further analysis and plotting.



(a) The Reeb graph window.



(b) Its corresponding Central Trajectory. The Central Trajectory is generated over four input trajectories (A, B, C, D) and is marked in black.

Figure 14: The Reeb graph window and its corresponding Central Trajectory (using $\epsilon = 0.88$).

Plotting

To generate the plots for the results, the Python library matplotlib is used. A Python script was written for every experiment. The scripts read the database output by the program, analyze it and then produce a plot in PDF format.