One-Pass Trajectory Simplification Using the Synchronous Euclidean Distance

Xuelian Lin, Jiahao Jiang, Shuai Ma, Yimeng Zuo, Chunming Hu

Abstract—Various mobile devices have been used to collect, store and transmit tremendous trajectory data, and it is known that raw trajectory data seriously wastes the storage, network band and computing resource. Line simplification algorithms are an effective approach to attacking this issue by compressing data points in a trajectory to a set of continuous line segments, and are commonly used in practice. However, although there exist one-pass line simplification algorithms appropriate for resource-constrained devices, none of them uses the synchronous Euclidean distance (SED), and cannot support spatio-temporal queries. In this study, we develop two one-pass error bounded trajectory simplification algorithms (CISED-S and CISED-W) using the synchronous Euclidean distance, based on a novel spatio-temporal cone intersection technique. Using four real-life trajectory datasets, we experimentally show that our approaches are both efficient and effective. In terms of running time, algorithms CISED-S and CISED-W are on average 3 times faster than SQUISH-E (the most efficient existing LS algorithm using SED). In terms of compression ratios, algorithms CISED-S and CISED-W are comparable with and 19.6% better than DPSED (the most effective existing LS algorithm using SED) on average, respectively, and are 21.1% and 42.4% better than SQUISH-E on average, respectively.

Index Terms—Trajectory Simplification, Spatiotemporal Compression, Synchronous Euclidean Distances, Cone Intersection

1 Introduction

Various mobile devices, such as smart-phones, on-board diagnostics, personal navigation devices, and wearable smart devices, have been using their sensors to collect massive trajectory data of moving objects at a certain sampling rate (e.g., a data point every 5 seconds), which is transmitted to cloud servers for various applications such as location based services and trajectory mining. Transmitting and storing raw trajectory data consumes too much network bandwidth and storage capacity [1], [3], [9], [11], [14], [15], [18]. It is known that these issues can be resolved or greatly alleviated by trajectory compression techniques via removing redundant data points of trajectories [1], [3], [4], [7], [9], [10], [11], [13], [14], [15], [18], [23], [25], among which the piece-wise line simplification technique is widely used [1], [3], [4], [8], [9], [11], [13], [14], [23], due to its distinct advantages: (a) simple and easy to implement, (b) no need of extra knowledge and suitable for freely moving objects, and (c) bounded errors with good compression ratios [8], [18].

Originally, line simplification algorithms adopt the *perpendicular Euclidean distance* (PED) as a metric to compute the errors, *e.g.*, $|\overrightarrow{P_4P_4^*}|$ is the PED of data point P_4 to the line $\overline{P_0P_{10}}$ in Figure 1 (left). Line simplification algorithms using PED have good compression ratios [1], [3], [4], [7], [9], [14], [23]. However, when using PED, the temporal information is lost. Thus, a spatio-temporal query, *e.g.*, "*the position of a moving object at time t*", on the compressed trajectories by line simplification algorithms using PED may return an approximate point P' whose distance to the actual position P of the moving object at time t is unbounded.

The *synchronous Euclidean distances* (SED) was then introduced for trajectory compression to support the above spatio-temporal queries [11]. SED is the Euclidean distance of a data point to its *approximate temporally synchronized data point [11]* on the corresponding line segment. For instance, P'_4 and P'_7 are the *synchronized data points* of points P_4 and P_7 *w.r.t.* line segments $\overrightarrow{P_0P_{10}}$ and $\overrightarrow{P_4P_{10}}$, respectively, in Figure 1 (right). Line simplification algorithms using SED may produce more line segments. However, the use of SED ensures that the Euclidean distance between a data point and its synchronized point *w.r.t.* the corresponding line segment is bounded. Hence, the above spatio-temporal query over the trajectories compressed by SED enabled approaches returns the synchronized point P' of a data point P within a bounded distance.

Line simplification methods using SED have been developed for batch algorithms (*e.g.*, Douglas-Peucker based algorithm DPSED [11]) and online algorithms (*e.g.*, SQUISH-E [14]). However, these methods still have a high time and/or space complexity, which hinders their utility in resource-constrained devices [8]. On the other hand, there are one-pass algorithms [5], [8], [24], [27], [28] that are more appropriate for resource-constrained devices. However, to our knowledge, all these existing one-pass algorithms are using PED, and cannot be directly applied for SED.

Contributions. To this end, we propose two one-pass error bounded line simplification algorithms using SED for compressing trajectories in an efficient and effective way.

(1) We first develop a novel local synchronous distance checking approach, *i.e.*, Cone Intersection using the Synchronous Euclidean Distance (CISED), by extending the sector intersection method [24], [27], [28] (Section 3). We

- *X. Lin, J. Jiang, S. Ma (correspondence), Y. Zuo and C. Hu are with SKLSDE lab, School of Computer Science and Engineering and Beijing Advanced Innovation Center for Big Data and Brain Computing, Beihang University, China.*
  E-mail: {linxl, jiangjh, mashuai, zuoym, hucm}@buaa.edu.cn

Manuscript received XXX, 2017; revised XXX, 2017.

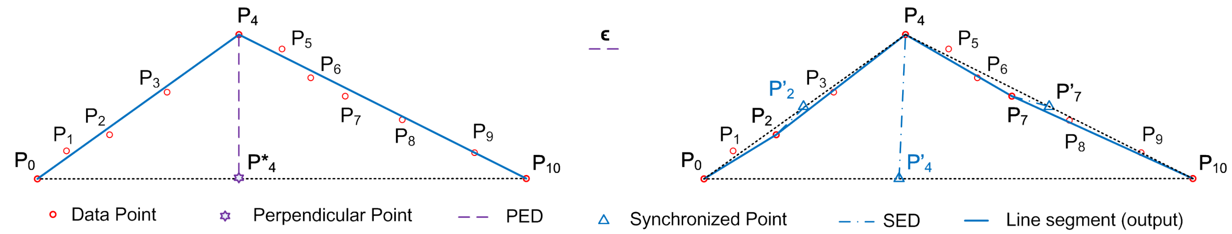


Figure 1. A trajectory $\dddot{\mathcal{T}}[P_0, \ldots, P_{10}]$ with eleven points is represented by two (left) and four (right) continuous line segments (solid blue), compressed by the Douglas–Peucker algorithm [4] using PED and SED, respectively. The Douglas–Peucker algorithm firstly creates line segment $\overrightarrow{P_0P_{10}}$, then it calculates the distance of each point in the trajectory to $\overrightarrow{P_0P_{10}}$. It finds that point P_4 has the maximum distance to $\overrightarrow{P_0P_{10}}$, and is greater than the user defined threshold ϵ. Then it goes to compress sub-trajectories $[P_0, \ldots, P_4]$ and $[P_4, \ldots, P_{10}]$, separately.

also approximate the intersection of spatio-temporal cones with the intersection of regular polygons, and develop a fast regular polygon intersection algorithm, such that each data point in a trajectory is checked in $O(1)$ time during the entire process of trajectory simplification.

(2) We then develop two one-pass error bounded trajectory simplification algorithms CISED-S and CISED-W using the synchronous Euclidean distance, based on our local synchronous distance checking technique (Section 4). CISED-S belongs to strong simplification that only has original points in its output, while CISED-W belongs to weak simplification that allows interpolated data points in its output.

(3) Using four real-life trajectory datasets (Truck, ServiceCar, GeoLife, PrivateCar), we finally conduct an extensive experimental study (Section 5), by comparing our methods CISED-S and CISED-W with DPSED [11] (the most effective existing LS algorithm using SED) and SQUISH-E [14] (the most efficient existing LS algorithm using SED).

Algorithms CISED-S and CISED-W are on average (20.7, 14.2, 18.2, 10.0) and (2.7, 2.8, 3.4, 2.9) times faster than DPSED and SQUISH-E on (Truck, ServiceCar, GeoLife, PrivateCar), respectively. For compression ratios, CISED-S is better than SQUISH-E and is comparable with DPSED. The sizes of the outputs of CISED-S are on average (91.8%, 79.3%, 71.9%, 72.7%) and (113.2%, 109.2%, 108.0%, 109.1%) of SQUISH-E and DPSED on (Truck, ServiceCar, GeoLife, PrivateCar), respectively. Moreover, CISED-W is better than SQUISH-E and DPSED that are on average (64.4%, 57.7%, 53.8%, 54.6%) and (79.2%, 79.5%, 80.9%, 82.0%) of SQUISH-E and DPSED on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

2 PRELIMINARIES

In this section, we introduce basic concepts and techniques for trajectory compression.

2.1 Basic Notations

We first introduce basic notations.

Points (P). A data point is defined as a triple $P(x, y, t)$, which represents that a moving object is located at *longitude* x and *latitude* y at *time* t. Note that data points can be viewed as points in a three-dimension Euclidean space.

Trajectories ($\dddot{\mathcal{T}}$). A trajectory $\dddot{\mathcal{T}}[P_0, \ldots, P_n]$ is a sequence of data points in a monotonically increasing order of their associated time values (*i.e.*, $P_i.t < P_j.t$ for any $0 \le i < j \le n$). Intuitively, a trajectory is the path (or track) that a moving object follows through space as a function of time [12].

Directed line segments ($\mathcal{L}$). A directed line segment (or line segment for simplicity) $\mathcal{L}$ is defined as $\overrightarrow{P_sP_e}$, which represents the closed line segment that connects the start point P_s and the end point P_e. Note that here P_s or P_e may not be a point in a trajectory $\dddot{\mathcal{T}}$.

We also use $|\mathcal{L}|$ and $\mathcal{L}.\theta \in [0, 2\pi)$ to denote the length of a directed line segment $\mathcal{L}$, and its angle with the x-axis of the coordinate system (x, y), where x and y are the longitude and latitude, respectively. That is, a directed line segment $\mathcal{L}$ = $\overrightarrow{P_sP_e}$ can be treated as a triple $(P_s, |\mathcal{L}|, \mathcal{L}.\theta)$.

Piecewise line representation ($\overline{\mathcal{T}}$). A piece-wise line representation $\overline{\mathcal{T}}[\mathcal{L}_0, \ldots, \mathcal{L}_m]$ $(0 < m \le n)$ of a trajectory $\dddot{\mathcal{T}}[P_0, \ldots, P_n]$ is a sequence of continuous directed line segments $\mathcal{L}_i$ = $\overrightarrow{P_{s_i}P_{e_i}}$ $(i \in [0, m])$ of $\dddot{\mathcal{T}}$ such that $\mathcal{L}_0.P_{s_0} = P_0$, $\mathcal{L}_m.P_{e_m} = P_n$ and $\mathcal{L}_i.P_{e_i} = \mathcal{L}_{i+1}.P_{s_{i+1}}$ for all $i \in [0, m-1]$. Note that each directed line segment in $\overline{\mathcal{T}}$ essentially represents a continuous sequence of data points in $\dddot{\mathcal{T}}$.

Perpendicular Euclidean Distances (PED). Given a data point P and a directed line segment $\mathcal{L}$ = $\overrightarrow{P_sP_e}$, the perpendicular Euclidean distance (or simply perpendicular distance) $ped(P, \mathcal{L})$ of P to $\mathcal{L}$ is the Euclidean distance of P to line $\overline{P_sP_e}$, adopted by many trajectory simplification methods, *e.g.*, [4], [5], [7], [8], [9], [24], [27], [28].

Synchronized points. Given a sub-trajectory $\dddot{\mathcal{T}}_s[P_s, \ldots, P_e]$, the synchronized point P' of a data point $P \in \dddot{\mathcal{T}}_s$, *w.r.t.* line segment $\overrightarrow{P_sP_e}$ is defined as follows: (1) $P'.x = P_s.x + c \cdot (P_e.x - P_s.x)$, (2) $P'.y = P_s.y + c \cdot (P_e.y - P_s.y)$ and (3) $P'.t$ = $P.t$, where $c = \frac{P.t - P_s.t}{P_e.t - P_s.t}$.

Synchronous Euclidean Distances (SED). Given a data point P and a directed line segment $\mathcal{L}$ = $\overrightarrow{P_sP_e}$, the synchronous Euclidean distance (or simply synchronous distance) $sed(P, \mathcal{L})$ of P to $\mathcal{L}$ is $|\overrightarrow{PP'}|$ that is the Euclidean distance from P to its synchronized data point P' *w.r.t.* $\mathcal{L}$.

We illustrate these notations with examples.

Example 1: Consider Figure 1, in which

(1) $\dddot{\mathcal{T}}[P_0, \ldots, P_{10}]$ is a trajectory having eleven data points,

(2) the set of two continuous line segments $\{\overrightarrow{P_0P_4}, \overrightarrow{P_4P_{10}}\}$ (Left) and the set of four continuous line segments $\{\overrightarrow{P_0P_2}, \overrightarrow{P_2P_4}, \overrightarrow{P_4P_7}, \overrightarrow{P_7P_{10}}\}$ (Right) are two piecewise line representations of trajectory $\dddot{\mathcal{T}}$,

(3) $ped(P_4, \overrightarrow{P_0P_{10}}) = |\overrightarrow{P_4P_4^*}|$, where P_4^* is the perpendicular point of P_4 *w.r.t.* line segment $\overrightarrow{P_0P_{10}}$, and

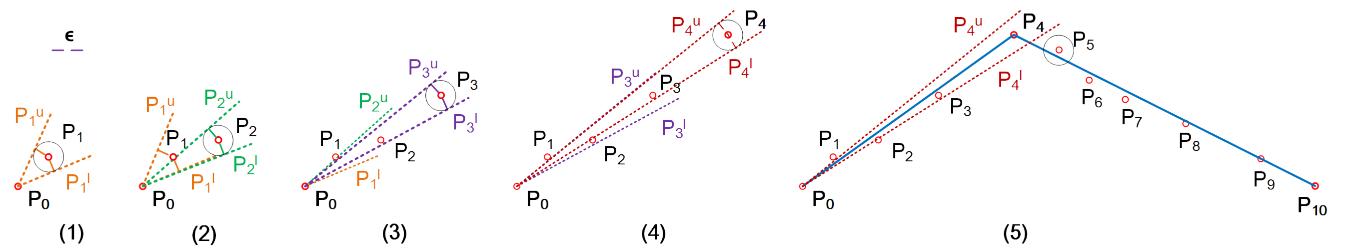


Figure 2. Trajectory $\dddot{\mathcal{T}}[P_0, \ldots, P_{10}]$ in Figure 1 is compressed into two line segments by the Sector Intersection algorithm [24], [27].

Table 1
Summary of notations

Notations	Semantics
P	a data point
$\dddot{\mathcal{T}}$	a trajectory $\dddot{\mathcal{T}}$ is a sequence of data points
$\mathcal{L}$	a directed line segment
$\overline{\mathcal{T}}$	a piece-wise line representation of a trajectory
$ped(P, \mathcal{L})$	the perpendicular Euclidean distance of P to $\mathcal{L}$
$sed(P, \mathcal{L})$	the synchronous Euclidean distance of P to $\mathcal{L}$
ϵ	the error bound
$\mathcal{S}$	a sector
$\vec{A} \times \vec{B}$	the cross product of (vectors) $\vec{A}$ and $\vec{B}$
$\mathcal{H}(\mathcal{L})$	The open half-plane to the left of $\mathcal{L}$
$\mathcal{G}$	a convex polygon
$\mathcal{G}^*$	the intersection of convex polygons
m	the max number of vertexes/edges of a polygon
E^j	a group of edges labeled with j
$g(e)$	the label of an edge e of polygons
$\mathcal{O}$	a synchronous circle
$\mathcal{C}$	a spatio-temporal cone
$\mathcal{O}^c$	a cone projection circle
$\prod$	intersection of geometries

(4) $sed(P_4, \overrightarrow{P_0P_{10}}) = |\overrightarrow{P_4P_4'}|$, $sed(P_2, \overrightarrow{P_0P_4}) = |\overrightarrow{P_2P_2'}|$ and $sed(P_7, \overrightarrow{P_4P_{10}}) = |\overrightarrow{P_7P_7'}|$, where points P_4', P_2' and P_7' are the synchronized points of points P_4, P_2 and P_7 *w.r.t.* line segments $\overrightarrow{P_0P_{10}}$, $\overrightarrow{P_0P_4}$ and $\overrightarrow{P_4P_{10}}$, respectively. □

Error bounded algorithms. Given a trajectory $\dddot{\mathcal{T}}$ and its compression algorithm $\mathcal{A}$ using SED (respectively PED) that produces another trajectory $\dddot{\mathcal{T}}'$, we say that algorithm $\mathcal{A}$ is error bounded by ϵ if for each point P in $\dddot{\mathcal{T}}$, there exist points P_j and P_{j+1} in $\dddot{\mathcal{T}}'$ such that $sed(P, \mathcal{L}(P_j, P_{j+1})) \le \epsilon$ (respectively $ped(P, \mathcal{L}(P_j, P_{j+1})) \le \epsilon$).

We summarize notations used in Table 1.

2.2 Sector Intersection based Algorithms using PED

The sector intersection (SI) algorithm [24], [27] was developed for graphic and pattern recognition in the late 1970s, for the approximation of arbitrary planar curves by linear segments or finding a polygonal approximation of a set of input data points in a 2D Cartesian coordinate system. The Sleeve algorithm [28] in the cartographic discipline essentially applies the same idea as the SI algorithm. Further, [5] optimized algorithm SI by considering the distance between a potential end point and the initial point of a line segment. It is worth pointing out that all these SI based algorithms use the perpendicular Euclidean distances.

Given a sequence of data points $[P_s, P_{s+1}, \ldots, P_{s+k}]$ and an error bound ϵ, the SI based algorithms process the input points one by one in order, and produce a simplified polyline. Instead of using the distance threshold ϵ directly, the SI based algorithms convert the distance tolerance into a variable angle tolerance for testing the points.

For the start point P_s, any point P_{s+i} and $|\overrightarrow{P_sP_{s+i}}| > \epsilon$ ($i \in [1, k]$), there are two different lines $\overrightarrow{P_sP^u_{s+i}}$ and $\overrightarrow{P_sP^l_{s+i}}$ such that $ped(P_i, \overrightarrow{P_sP^u_{s+i}}) = ped(P_i, \overrightarrow{P_sP^l_{s+i}}) = \epsilon$ and either ($\overrightarrow{P_sP^l_{s+i}}.\theta < \overrightarrow{P_sP^u_{s+i}}.\theta$ *and* $\overrightarrow{P_sP^u_{s+i}}.\theta - \overrightarrow{P_sP^l_{s+i}}.\theta < \pi$) or ($\overrightarrow{P_sP^l_{s+i}}.\theta > \overrightarrow{P_sP^u_{s+i}}.\theta$ *and* $\overrightarrow{P_sP^u_{s+i}}.\theta - \overrightarrow{P_sP^l_{s+i}}.\theta < -\pi$). Indeed, they forms a *sector* $\mathcal{S}(P_s, P_{s+i}, \epsilon)$ that takes P_s as the center point and $\overrightarrow{P_sP^u_{s+i}}$ and $\overrightarrow{P_sP^l_{s+i}}$ as the border lines. Then there exists a data point Q such that for any data point P_{s+i} ($i \in [1, ...k]$), its PED to line $\overrightarrow{P_sQ}$ is no greater than the error bound ϵ if and only if the k sectors $\mathcal{S}(P_s, P_{s+i}, \epsilon)$ ($i \in [1, k]$) share common data points other than P_s, *i.e.*, $\prod_{i=1}^{k} \mathcal{S}(P_s, P_{s+i}, \epsilon) \neq \{P_s\}$ [24], [27], [28].

The point Q may not belong to $\{P_s, P_{s+1}, \ldots, P_{s+k}\}$. However, if P_{s+i} ($1 \le i \le k$) is chosen as Q, then for any data point P_{s+j} ($j \in [1, ...i]$), its PED to line $\overrightarrow{P_sP_{s+i}}$ is no greater than the error bound ϵ if and only if $\prod_{j=1}^{i} \mathcal{S}(P_s, P_{s+j}, \epsilon/2) \neq \{P_s\}$, as pointed out in [28].

That is, *these* SI *based algorithms can be easily adopted for trajectory compression using* SED *although they have been overlooked by existing trajectory simplification studies*. It is also worth pointing out that the SI based algorithms run in $O(n)$ time and $O(1)$ space, and are one-pass algorithms.

We next illustrate how the SI based algorithms can be used for trajectory compression.

Example 2: Consider Figure 2. A SI based algorithm takes as input a trajectory $\dddot{\mathcal{T}}[P_0, \ldots, P_{10}]$, and returns a simplified ployline consisting of two line segments $\overrightarrow{P_0P_4}$ and $\overrightarrow{P_4P_{10}}$.

(1) Initially, P_0 is the start point. Point P_1 is firstly read, and the sector $\mathcal{S}(P_0, P_1, \epsilon/2)$ of P_1 is created as shown in Figure 2.(1). Then P_2 is read, and the sector $\mathcal{S}(P_0, P_2, \epsilon/2)$ is created for P_2. The intersection of sectors $\mathcal{S}(P_0, P_1, \epsilon/2)$ and $\mathcal{S}(P_0, P_2, \epsilon/2)$ contains data points other than P_0 which has an up border line $P_0P^u_2$ and a low border line $P_0P^l_1$, as shown in Figure 2.(2). Similarly, points P_3 and P_4 are processed, as shown in Figures 2.(3) and 2.(4), respectively.

(2) When point P_5 is read, line segment $\overrightarrow{P_0P_4}$ is produced, and point P_4 becomes the start point, as $\prod_{i=1}^{4} \mathcal{S}(P_0, P_{s+i}, \epsilon/2) \neq \{P_0\}$ and $\prod_{i=1}^{5} \mathcal{S}(P_0, P_{s+i}, \epsilon/2) = \{P_0\}$ as shown in Figure 2.(5).

(3) Points $P_5, \ldots, P_{10}$ are processed similarly one by one in order, and finally the algorithm outputs another line segment $\overrightarrow{P_4P_{10}}$ as shown in Figure 2.(5). □

2.3 Intersection Computation of Convex Polygons

We also employ a convex polygon intersection algorithm in [16], whose basic idea is straightforward. Assume *w.l.o.g.* that the edges of polygons $\mathcal{G}_1$ and $\mathcal{G}_2$ are oriented counter-

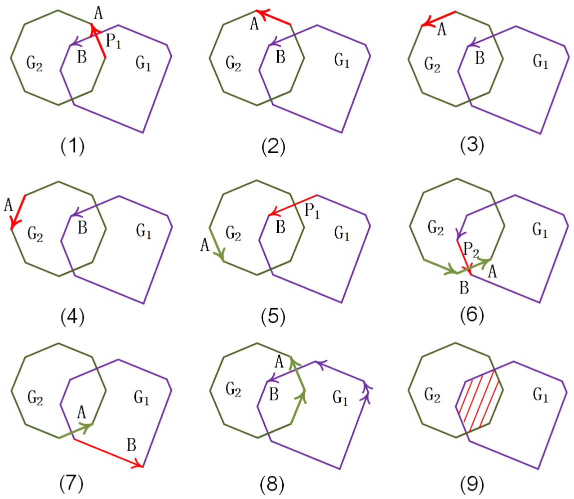


Figure 3. A running example of convex polygons intersection.

clockwise, and $\vec{A} = (P_{s_A}, P_{e_A})$ and $\vec{B} = (P_{s_B}, P_{e_B})$ are two (directed) edges on $\mathcal{G}_2$ and $\mathcal{G}_1$, repectively (see Figure 3).

The algorithm has $\vec{A}$ and $\vec{B}$ "chasing" one another, *i.e.*, moves $\vec{A}$ on $\mathcal{G}_2$ and $\vec{B}$ on $\mathcal{G}_1$ counter-clockwise step by step under certain rules, so that they meet at every crossing of $\mathcal{G}_1$ and $\mathcal{G}_2$. The rules, called *advance rules*, are carefully designed depending on geometric conditions of $\vec{A}$ and $\vec{B}$. Let $\vec{A} \times \vec{B}$ be the cross product of (vectors) $\vec{A}$ and $\vec{B}$, and $\mathcal{H}(\vec{A})$ be the open half-plane to the left of $\vec{A}$, the rules are as follows:

Rule (1): If $\vec{A} \times \vec{B} < 0$ and $P_{e_A} \notin \mathcal{H}(\vec{B})$, or $\vec{A} \times \vec{B} > 0$ and $P_{e_B} \in \mathcal{H}(\vec{A})$, then $\vec{A}$ is advanced a step.

For example, in Figure 3.(1) and 3.(2), $\vec{A}$ moves forward a step as $\vec{A} \times \vec{B} > 0$ and $P_{e_B} \in \mathcal{H}(\vec{A})$.

Rule (2): If $\vec{A} \times \vec{B} > 0$ and $P_{e_B} \notin \mathcal{H}(\vec{A})$, or $\vec{A} \times \vec{B} < 0$ and $P_{e_A} \in \mathcal{H}(\vec{B})$, then $\vec{B}$ is advanced a step.

For example, in Figure 3.(6) and 3.(7), $\vec{B}$ moves forward a step as $\vec{A} \times \vec{B} < 0$ and $P_{e_A} \in \mathcal{H}(\vec{B})$.

Algorithm CPolyInter. The complete algorithm is shown in Figure 4. Given polygons $\mathcal{G}_1$ and $\mathcal{G}_2$, algorithm CPolyInter first arbitrarily sets $\vec{A}$ on $\mathcal{G}_2$ and $\vec{B}$ on $\mathcal{G}_1$, respectively (line 1). It then checks the intersection of $\vec{A}$ and $\vec{B}$. If $\vec{A}$ intersects $\vec{B}$ (line 3), then the algorithm checks for some special termination conditions (*e.g.*, if $\vec{A}$ and $\vec{B}$ are overlapped and, at the same time, $\mathcal{G}_1$ and $\mathcal{G}_2$ are on the opposite sides of the overlapped edges, then the process is terminated) (line 4), and records the inner edge, which is a boundary segment of the intersection polygon (line 5). After that, the algorithm moves on $\vec{A}$ or $\vec{B}$ one step under the advance rules (lines 6–11). The above processes repeated, until both $\vec{A}$ and $\vec{B}$ cycle their polygons (line 12). Next, the algorithm handles three special cases of the two polygons, *i.e.*, $\mathcal{G}_1$ is inside of $\mathcal{G}_2$, $\mathcal{G}_2$ is inside of $\mathcal{G}_1$ and $\mathcal{G}_1 \sqcap \mathcal{G}_2 = \emptyset$ cases (line 13). At last, it returns the intersection polygon (line 14).

The algorithm has a time complexity of $O(|\mathcal{G}_1| + |\mathcal{G}_2|)$, where $|\mathcal{G}|$ is the number of edges of polygon $\mathcal{G}$. It is worth pointing out that $|\mathcal{G}_1 \sqcap \mathcal{G}_2| \le (|\mathcal{G}_1| + |\mathcal{G}_2|)$.

Example 3: Figure 3 shows a running example of the convex polygon intersection algorithm CPolyInter.

(1) Initially, directed edges $\vec{A}$ and $\vec{B}$ are on polygons $\mathcal{G}_2$ and $\mathcal{G}_1$, respectively, such that $\vec{A} \sqcap \vec{B} = \{P_1\}$, *i.e.*, $\vec{A}$ and $\vec{B}$ intersect on point P_1, as shown in Figure 3.(1).

(2) Then, by the advance rules, edge $\vec{A}$ moves on a step and makes $\vec{A} \sqcap \vec{B} = \emptyset$ as shown in Figure 3.(2). After 7 steps of moving of $\vec{A}$ or $\vec{B}$, each by an advance rule, edges $\vec{A}$ and $\vec{B}$ intersect on point P_2, as shown in Figure 3.(6).

(3) Next, edge $\vec{B}$ moves on a step, and makes $\vec{A} \sqcap \vec{B} = \emptyset$, as shown in Figure 3.(7). After 6 steps of moving of edge $\vec{B}$ or $\vec{A}$ one by one, both edges $\vec{A}$ and $\vec{B}$ have finished their cycles as shown in Figure 3.(8).

(4) The algorithm finally returns the intersection polygon as shown in Figure 3.(9). □

Algorithm CPolyInter ($\mathcal{G}_1$, $\mathcal{G}_2$)
1. **set** $\vec{A}$ and $\vec{B}$ arbitrarily on $\mathcal{G}_1$ and $\mathcal{G}_2$
2. **repeat**
3. **if** $\vec{A} \sqcap \vec{B} \neq \emptyset$ **then**
4. Check for termination.
5. Update an inside flag.
6. **if** ($\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \notin \mathcal{H}(\vec{B})$) **or**
7. ($\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \in \mathcal{H}(\vec{A})$) **then**
8. advance $\vec{A}$ one step
9. **elseif** ($\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \notin \mathcal{H}(\vec{A})$) **or**
10. ($\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \in \mathcal{H}(\vec{B})$) **then**
11. advance $\vec{B}$ one step
12. **until** both $\vec{A}$ and $\vec{B}$ cycle their polygons
13. **handle** $\mathcal{G}_1 \subset \mathcal{G}_2$ and $\mathcal{G}_2 \subset \mathcal{G}_1$ and $\mathcal{G}_1 \sqcap \mathcal{G}_2 = \emptyset$ cases
14. **return** $\mathcal{G}_1 \sqcap \mathcal{G}_2$

Figure 4. Algorithm for convex polygons intersection [16].

3 Local Synchronous Distance Checking

In this section, we develop a local synchronous distance checking approach such that each data point in a trajectory is checked only once in $O(1)$ time during the entire process of trajectory simplification, by extending the sector intersection method in Section 2.2, which lays down the key for the one-pass trajectory simplification algorithms in Section 4.

We consider a sub-trajectory $\ddot{\mathcal{T}}_s[P_s, \ldots, P_{s+k}]$, an error bound ϵ, and a 3D Cartesian coordinate system whose origin, x-axis, y-axis and t-axis are P_s, longitude, latitude and time, respectively.

3.1 Spatio-Temporal Cone Intersection

We first present the spatio-temporal cone intersection method in a 3D Cartesian coordinate system, which extends the sector intersection method [24], [27], [28].

Synchronous Circles ($\mathcal{O}$). The synchronous circle of a data point P_{s+i} ($1 \le i \le k$) in $\ddot{\mathcal{T}}_s$, denoted as $\mathcal{O}(P_{s+i}, \epsilon)$, or $\mathcal{O}_{s+i}$ in short, is a circle on the plane $P.t - P_{s+i}.t = 0$ such that P_{s+i} is its center and ϵ is its radius.

It is easy to know that for any point in the area of a circle $\mathcal{O}(P_{s+i}, \epsilon)$, its distance to P_{s+i} is no greater than ϵ. Figure 5 shows two synchronous circles $\mathcal{O}(P_{s+i}, \epsilon)$ of point P_{s+i} and $\mathcal{O}(P_{s+i+1}, \epsilon)$ of point P_{s+i+1}.

Spatio-temporal cones ($\mathcal{C}$). The spatio-temporal cone (or simply cone) of point P_{s+i} *w.r.t.* point P_s, denoted as $\mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon))$, or $\mathcal{C}_{s+i}$ in short, is an oblique circular cone such that point P_s is its apex and the synchronous circle $\mathcal{O}(P_{s+i}, \epsilon)$ is its base.

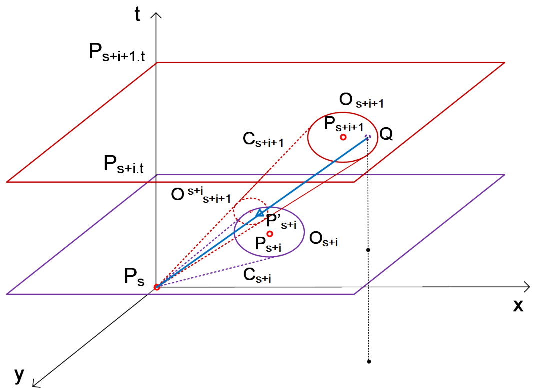


Figure 5. Examples of spatio-temporal cones.

Two example spatio-temporal cones $\mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon))$ and $\mathcal{C}(P_s, \mathcal{O}(P_{s+i+1}, \epsilon))$ are illustrated in Figure 5.

Cone projection circles. The projection of a cone $\mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon))$ on a plane $P.t - t_c = 0$ ($t_c > P_s.t$) is a circle $\mathcal{O}^c(P^c_{s+i}, r^c_{s+i})$, or $\mathcal{O}^c{}_{s+i}$ in short, such that (1) $P^c_{s+i}.x = P_s.x + c \cdot (P_{s+i}.x - P_s.x)$, (2) $P^c_{s+i}.y = P_s.y + c \cdot (P_{s+i}.y - P_s.y)$, (3) $P^c_{s+i}.t = t_c$ and (4) $r^c_{s+i} = c \cdot \epsilon$, where $c = \frac{t_c - P_s.t}{P_{s+i}.t - P_s.t}$.

The red dashed circle $\mathcal{O}^c(P^c_{s+i+1}, r^c_{s+i+1})$ on plane "$P.t - P_{s+i}.t = 0$" in Figure 5 is the projection circle of cone $\mathcal{C}(P_s, \mathcal{O}(P_{s+i+1}, \epsilon))$ on the plane.

Proposition 1: *Given a sub-trajectory $[P_s, \ldots, P_{s+k}]$ and a point Q in the area of synchronous circle $\mathcal{O}(P_{s+k}, \epsilon)$, the intersection point P'_{s+i} of the directed line segment $\overrightarrow{P_sQ}$ and the plane $P.t - P_{s+i}.t = 0$ is the synchronized point of P_{s+i} ($1 \le i \le k$)* w.r.t. *$\overrightarrow{P_sQ}$, and the distance $|\overrightarrow{P_{s+i}P'_{s+i}}|$ from P_{s+i} to P'_{s+i} is the synchronous distance of P_{s+i} to $\overrightarrow{P_sQ}$.* □

Proof: It suffices to show that P'_{s+i} is indeed a synchronized data point P_{s+i} *w.r.t.* $\overrightarrow{P_sQ}$. The intersection point P'_{s+i} satisfies that $P'_{s+i}.t = P_{s+i}.t$ and $\frac{P'_{s+i}.t - P_s.t}{Q.t - P_s.t} = \frac{P_{s+i}.t - P_s.t}{Q.t - P_s.t} = \frac{|\overrightarrow{P_sP'_{s+i}}|}{|\overrightarrow{P_sQ}|} = \frac{P'_{s+i}.x - P_s.x}{Q.x - P_s.x} = \frac{P'_{s+i}.y - P_s.y}{Q.y - P_s.y}$. Hence, by the definition of synchronized points, we have the conclusion. □

Proposition 2: *Given a sub-trajectory $[P_s, ..., P_{s+k}]$ and an error bound ϵ, there exists a point Q such that $Q.t = P_{s+k}.t$ and $sed(P_{s+i}, \overrightarrow{P_sQ}) \le \epsilon$ for each $i \in [1, k]$ if and only if $\bigcap_{i=1}^{k} \mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon)) \ne \{P_s\}$.* □

Proof: Let P'_{s+i} ($i \in [1, k]$) be the intersection point of the line segment $\overrightarrow{P_sQ}$ and the plane $P.t - P_{s+i}.t = 0$. By Proposition 1, P'_{s+i} is the synchronized point of P_{s+i} *w.r.t.* $\overrightarrow{P_sQ}$.

Assume first that $\bigcap_{i=1}^{k} \mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon)) \ne \{P_s\}$. Then there must exit a point Q in the area of the synchronous circle $\mathcal{O}(P_{s+k}, \epsilon)$ such that $\overrightarrow{P_sQ}$ passes through all the cones $\mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon))$ $i \in [1, k]$. Hence, $Q.t = P_{s+k}.t$. We also have $sed(P_{s+i}, \overrightarrow{P_sQ}) = |\overrightarrow{P'_{s+i}P_{s+i}}| \le \epsilon$ for each $i \in [1, k]$ since P'_{s+i} is in the area of circle $\mathcal{O}(P_{s+i}, \epsilon)$.

Conversely, assume that there exists a point Q such that $Q.t = P_{s+k}.t$ and $sed(P_{s+i}, \overrightarrow{P_sQ}) \le \epsilon$ for all P_{s+i} ($i \in [1, k]$). Then $|\overrightarrow{P'_{s+i}P_{s+i}}| \le \epsilon$ for all $i \in [1, k]$. Hence, we have $\bigcap_{i=1}^{k} \mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon)) \ne \{P_s\}$. □

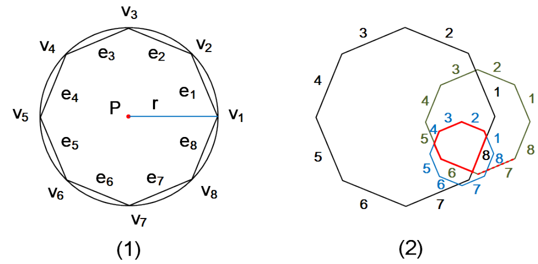


Figure 6. Regular octagons and their intersections ($m = 8$).

By Proposition 2, we now have a spatio-temporal cone intersection method in a 3D Cartesian coordinate system, which extends the sector intersection method [24], [27], [28].

Proposition 3: *Given a sub-trajectory $[P_s, \ldots, P_{s+k}]$, an error bound ϵ, and any $t_c > P_s.t$, there exists a point Q such that $Q.t = P_{s+k}.t$ and $sed(P_i, \overrightarrow{P_sQ}) \le \epsilon$ for all points P_{s+i} ($i \in [1, k]$) if and only if $\bigcap_{i=1}^{k} \mathcal{O}^c(P^c_{s+i}, r^c_{s+i}) \ne \emptyset$.* □

Proof: By Proposition 2, it suffices to show that $\bigcap_{i=1}^{k} \mathcal{O}^c(P^c_{s+i}, r^c_{s+i}) \ne \emptyset$ if and only if $\bigcap_{i=1}^{k} \mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon)) \ne \{P_s\}$, which is obvious. Hence, we have the conclusion. □

Proposition 3 tells us that the intersection checking of spatio-temporal cones can be reduced to simply check the intersection of cone projection circles on a plane.

3.2 Inscribed Regular Polygon Intersection

Finding the common intersection of N circles has a time complexity of $O(N \log N)$ [22], which cannot be used for designing one-pass trajectory simplification algorithms using the synchronous distance. However, we can approximate a circle with its m-edge inscribed regular polygon, whose intersection can be computed more efficiently.

Inscribed regular polygons ($\mathcal{G}$). Given a cone projection circle $\mathcal{O}^c(P, r)$, its inscribed m-edge regular polygon is denoted as $\mathcal{G}(V, E)$, where (1) $V = \{v_1, \ldots, v_m\}$ is the set of vertexes that are defined by a polar coordinate system, whose origin is the center P of $\mathcal{O}^c$, as follows:

$$v_j = (r, \frac{(j-1)}{m} 2\pi),\ j \in [1, m],$$

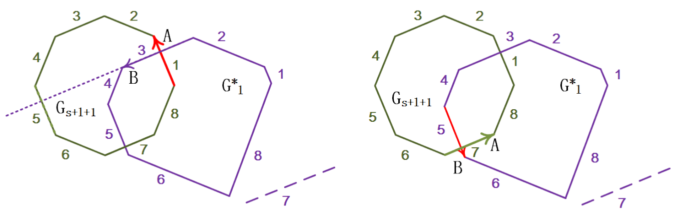
and (2) $E = \{\overrightarrow{v_m v_1}\} \bigcup \{\overrightarrow{v_j v_{j+1}} \mid j \in [1, m-1]\}$ is the set of edges that are labeled with the subscript of their start points.

Figure 6-(1) illustrates the inscribed regular octagon ($m = 8$) of a cone projection circle $\mathcal{O}^c(P, r)$.

Let $\mathcal{G}_{s+i}$ ($1 \le i \le k$) be the inscribed regular polygon of the cone projection circle $\mathcal{O}^c(P^c_{s+i}, r^c_{s+i})$, $\mathcal{G}^*_l$ ($1 \le l \le k$) be the intersection $\bigcap_{i=1}^{l} \mathcal{G}_{s+i}$, and E^j ($1 \le j \le m$) be the group of k edges labeled with j in all $\mathcal{G}_{s+i}$ ($i \in [1, k]$). It is easy to verify that all edges in the same edge groups E^j ($1 \le j \le m$) are in parallel with each other by the above definition of inscribed regular polygons, as illustrated in Figure 6.(2).

Proposition 4: *The intersection $\mathcal{G}^*_l \bigcap \mathcal{G}_{s+l+1}$ ($1 \le l < k$) has at most m edges,* i.e., *at most one from each edge group.* □

Proof: We shall prove this by contradiction. Assume that $\mathcal{G}^*_l \bigcap \mathcal{G}_{s+l+1}$ has two distinct edges $\overrightarrow{A_i}$ and $\overrightarrow{A_{i'}}$ with the same label j ($1 \le j \le m$), originally from $\mathcal{G}_{s+i}$ and $\mathcal{G}_{s+i'}$ ($1 \le i < i' \le l + 1$). Note that here $\mathcal{G}_{s+i} \bigcap \mathcal{G}_{s+i'} \ne \emptyset$



(1) (2)

Figure 7. Examples of fast advancing rules.

since $\mathcal{G}_l^* \sqcap \mathcal{G}_{s+l+1} \neq \emptyset$. However, when $\mathcal{G}_{s+i} \sqcap \mathcal{G}_{s+i'} \neq \emptyset$, the intersection $\mathcal{G}_{s+i} \sqcap \mathcal{G}_{s+i'}$ cannot have both edges $\vec{A_i}$ and $\vec{A_{i'}}$, which contradicts the assumption. □

Figure 6.(2) shows the intersection polygon (red lines) of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ with 7 edges, and here edges labeled with 7 have no contributions to the resulting intersection polygon.

Proposition 5: *The intersection $\mathcal{G}_l^* \sqcap \mathcal{G}_{s+l+1}$ $(1 \leq l < k)$ of $\mathcal{G}_l^*$ and $\mathcal{G}_{s+l+1}$ can be done in $O(1)$ time.* □

Proof: The inscribed regular polygon $\mathcal{G}_{s+l+1}$ has m edges, and $\mathcal{G}_l^*$ has at most m edges by Proposition 4. As the intersection of two m-edge convex polygons can be computed in $O(m)$ time [16], the intersection of $\mathcal{G}_l^*$ and $\mathcal{G}_{s+l+1}$ can be done in $O(1)$ time for a fixed m. □

3.3 Speedup Inscribed Regular Polygon Intersection

Observe that algorithm CPolyInter in Figure 4 is for general convex polygons, while the inscribed regular polygons $\mathcal{G}_{s+i}$ $(i \in [1,k])$ of the cone projection circles are constructed in a unified way, which allows us to develop a fast method to compute their intersection.

Let $\vec{A} = (P_{s_A}, P_{e_A})$ and $\vec{B} = (P_{s_B}, P_{e_B})$ be two directed edges on polygons $\mathcal{G}_{s+l+1}$ and $\mathcal{G}_l^*$, respectively. Again edges $\vec{A}$ and $\vec{B}$ are moved counter-clockwise. Note that $\vec{A}$ and $\vec{B}$ are advanced step by step each time by the two advancing rules of algorithm CPolyInter. However, it is possible to advance $\vec{A}$ or $\vec{B}$ multiple steps each time. For example, in Figure 3.(1)–(5), edge $\vec{A}$ successively moves four steps, each under the advance rule (1) "$(\vec{A} \times \vec{B} < 0$ and $P_{e_A} \notin \mathcal{H}(\vec{B}))$ or $(\vec{A} \times \vec{B} > 0$ and $P_{e_B} \in \mathcal{H}(\vec{A}))$" of algorithm CPolyInter. Alternatively, we can directly move A from Figure 3.(1) to Figure 3.(5), by reducing four steps to one step only.

Proposition 6: *If either* $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \notin \mathcal{H}(\vec{B}))$ *or* $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \in \mathcal{H}(\vec{A}))$ *holds, then $\vec{A}$ is moved forward s steps such that*

$$s = \begin{cases} 2 \times (g(\vec{B}) - g(\vec{A})) & if\ g(\vec{B}) > g(\vec{A}) \\ 1 & if\ g(\vec{A}) = g(\vec{B}) \\ 2 \times (m + g(\vec{B}) - g(\vec{A})) & if\ g(\vec{B}) < g(\vec{A}), \end{cases}$$

in which $g(e)$ denotes the label of edge e. □

Proof: We first explain how the edge $\vec{A}$ is moved forward. Indeed, $\vec{A}$ is moved from its original position to its symmetric edge on $\mathcal{G}_{s+l+1}$ *w.r.t.* the symmetric line that is perpendicular to $\vec{B}$ on $\mathcal{G}_l^*$. For example, in Figure 7.(1), there is $\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \in \mathcal{H}(\vec{A})$, hence $\vec{A}$ moves on. As $g(\vec{B}) = 3 > 1 = g(\vec{A})$, $\vec{A}$ moves forward $2 \times (g(\vec{B}) - g(\vec{A})) = 2 \times (3-1) = 4$ steps. Here, the label of edge $\vec{A}$ is changed to 5, its symmetric edge 1 on $\mathcal{G}_{s+l+1}$ *w.r.t.* the symmetric line that is perpendicular to $\vec{B}$ labeled with 3 on $\mathcal{G}_l^*$.

We then present the proof. If $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \notin \mathcal{H}(\vec{B}))$ or $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \in \mathcal{H}(\vec{A}))$, then as all edges in the same edge groups E^j $(1 \leq j \leq m)$ are in parallel with each other, it is easy to find that, for each position of $\vec{A}$ between its original to its opposite positions, we have (1) $\vec{A} \sqcap \vec{B} = \emptyset$, and (2) either $P_{e_A} \notin \mathcal{H}(\vec{B})$ or $P_{e_B} \in \mathcal{H}(\vec{A})$. Hence, by the advance rule (1) of algorithm CPolyInter in Section 2.3, edge $\vec{A}$ is always moved forward until it reaches the opposite position of its original one. From this, we have the conclusion. □

Proposition 7: *If either* $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \notin \mathcal{H}(\vec{A}))$ *or* $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \in \mathcal{H}(\vec{B}))$ *holds, then $\vec{B}$ is directly moved to the edge after the one having the same edge group as $\vec{A}$.* □

Proof: We first explain how the edge $\vec{B}$ is moved forward. For example, in Figure 7.(2), $\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \in \mathcal{H}(\vec{B})$, hence $\vec{B}$ is moved forward. As the edge $\vec{A}$ is labeled with 7, $\vec{B}$ moves to the edge labeled with 8 on $\mathcal{G}_l^*$, which is the next of the edge labeled with 7 on $\mathcal{G}_l^*$. Note that if the edge labeled with 8 were not actually existing in the intersection polygon $\mathcal{G}_l^*$, then $\vec{B}$ should repeatedly move on until it reaches the first "real" edge on $\mathcal{G}_l^*$.

We then present the proof. If $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} > 0$ **and** $P_{e_B} \notin \mathcal{H}(\vec{A}))$ or $(\vec{A} \sqcap \vec{B} \neq \emptyset$ **and** $\vec{A} \times \vec{B} < 0$ **and** $P_{e_A} \in \mathcal{H}(\vec{B}))$, then it is also easy to find that, for each position of $\vec{B}$ between its original to its target positions (*i.e.*, the edge after the one having the same edge group as $\vec{A}$), we have (1) $\vec{A} \sqcap \vec{B} = \emptyset$, and (2) either $P_{e_B} \notin \mathcal{H}(\vec{A})$ or $P_{e_A} \in \mathcal{H}(\vec{B})$. Hence, by the advance rule (2) of algorithm CPolyInter in Section 2.3, edge $\vec{B}$ is always moved forward until it reaches the target position. From this, we have the conclusion. □

Algorithm FastRPolyInter. The presented regular polygon intersection algorithm, *i.e.*, FastRPolyInter, is the optimized version of the convex polygon intersection algorithm CPolyInter, by Propositions 6 and 7. We also saves vertexes of a polygon in a fixed size array, which is different from CPolyInter that saves polygons in linked lists. Considering the regular polygons each having a fixed number of vertexes/edges, marked from 1 to m, this policy allow us to quickly address an edge or vertex by its label.

Given intersection polygon $\mathcal{G}_l^*$ of the preview l polygons and the next approximate polygon $\mathcal{G}_{s+l+1}$, the algorithm FastRPolyInter returns $\mathcal{G}_{l+1}^* = \mathcal{G}_l^* \sqcap \mathcal{G}_{s+l+1}$. It runs the similar routine as the CPolyInter algorithm, except that (1) it saves polygons in arrays, and (2) the advance strategies are partitioned into two parts, *i.e.*, $\vec{A} \sqcap \vec{B} \neq \emptyset$ and $\vec{A} \sqcap \vec{B} = \emptyset$, where the former applies Propositions 6 and 7, and the later remains the same as algorithm CPolyInter.

Correctness and complexity analyses. Observe that algorithm FastRPolyInter basically has the same routine as algorithm CPolyInter, except that it fastens the advancing speed of directed edges $\vec{A}$ and $\vec{B}$ under certain circumstances as shown by Propositions 6 and 7, which together ensure the correctness of FastRPolyInter. Moreover, algorithm

FastRPolyInter runs in $O(1)$ time by Proposition 5.

4 ONE-PASS TRAJECTORY SIMPLIFICATION

Following [8], [26], we consider two classes of trajectory simplification. The first one, referred to as *strong simplification*, that takes as input a trajectory $\ddot{\mathcal{T}}$, an error bound ϵ and the number m of edges for inscribed regular polygons, and produces a simplified trajectory $\ddot{\mathcal{T}}'$ such that all data points in $\ddot{\mathcal{T}}'$ belongs to $\ddot{\mathcal{T}}$. The second one, referred to as *weak simplification*, that takes input a trajectory $\ddot{\mathcal{T}}$, an error bound ϵ and the number m of edges for inscribed regular polygons, and produces a simplified trajectory $\ddot{\mathcal{T}}'$ such that some data points in $\ddot{\mathcal{T}}'$ may not belong to $\ddot{\mathcal{T}}$. That is, weak simplification allows data interpolation.

The main result here is stated as follows.

Theorem 8: *There exist one-pass error bounded trajectory simplification algorithms using the synchronous distance for both strong and weak trajectory simplification.* □

We shall prove this by providing such algorithms for both strong and weak trajectory simplifications, by employing the constant time synchronous distance checking technique developed in Section 3.

4.1 Strong Trajectory Simplification

Recall that in Propositions 2 and 3, the point Q may not be in the input sub-trajectory $[P_s, ..., P_{s+k}]$. If we restrict $Q = P_{s+k}$, the end point of the sub-trajectory, then the cones whose base circles with a radius of $\epsilon/2$ suffice.

Proposition 9: *Given a sub-trajectory* $[P_s, ..., P_{s+k}]$ *and an error bound* ϵ, $sed(P_{s+i}, \overrightarrow{P_sP_{s+k}}) \le \epsilon$ *for each* $i \in [1, k]$ *if and only if* $\prod_{i=1}^{k} \mathcal{C}(P_s, \mathcal{O}(P_{s+i}, \epsilon/2)) \neq \{P_s\}$. □

Proof: If $\prod_{i=s+1}^{e} \mathcal{C}(P_s, P_{s+i}, \epsilon/2) \neq \{P_s\}$, then by Proposition 2, there exists a point Q, $Q.t = P_{s+k}.t$, such that $sed(P_{s+i}, \overrightarrow{P_sQ}) \le \epsilon/2$ for all $i \in [1, k]$. By the triangle inequality essentially, $sed(P_{s+i}, \overrightarrow{P_sP_{s+k}}) \le sed(P_{s+i}, \overrightarrow{P_sQ}) + |\overrightarrow{QP_{s+k}}| \le \epsilon/2 + \epsilon/2 = \epsilon$. □

We first present the one-pass error bounded *strong trajectory simplification* algorithm using the synchronous distance, which is shown in Figure 8.

Procedure getRegularPolygon. We first present procedure getRegularPolygon that given a cone projection circle, generates its inscribed m-edge regular polygon, following the definition in Section 3.2.

The parameters P_s, P_i, r and t_c together form the projection circle $\mathcal{O}^c(P_i^c, r_i^c)$ of the spatio-temporal cone $\mathcal{C}(P_s, \mathcal{O}(P_i, r))$ of point P_i *w.r.t.* point P_s on the plane $P.t - t_c = 0$. Firstly, $P_i^c.x$ and $P_i^c.y$ are computed (lines 1–3), and $r_i^c = c{\cdot}r$. Then it builds and returns an m-edge inscribed regular polygon $\mathcal{G}$ of $\mathcal{O}^c(P_i^c, r_i^c)$ (lines 4–8), by transforming a polar coordinate system into a Cartesian one. Note that here θ, $r \cdot \sin\theta$ and $r \cdot \cos\theta$ only need to be computed once during the entire processing of a trajectory.

Algorithm CISED-S. We now present algorithm CISED-S. It takes as input a trajectory $\ddot{\mathcal{T}}[P_0, \ldots, P_n]$, an error bound ϵ and the number m of edges for inscribed regular polygons, and returns a simplified trajectory $\overline{\mathcal{T}}$ of $\ddot{\mathcal{T}}$.

Algorithm CISED-S ($\ddot{\mathcal{T}}[P_0, \ldots, P_n]$, ϵ, m)
1. $P_s := P_0$; $i := 1$; $\mathcal{G}^* := \emptyset$; $\overline{\mathcal{T}} := \emptyset$; $t_c := P_1.t$;
2. **while** $i \le n$ **do**
3. $\quad \mathcal{G} :=$ getRegularPolygon (P_s, P_i, $\epsilon/2$, m, t_c);
4. $\quad$ **if** $\mathcal{G}^* = \emptyset$ **then** /* $\mathcal{G}^*$ needs to be initialized */
5. $\quad\quad \mathcal{G}^* := \mathcal{G}$;
6. $\quad$ **else**
7. $\quad\quad \mathcal{G}^* :=$ FastRPolyInter($\mathcal{G}^*$, $\mathcal{G}$);
8. $\quad\quad$ **if** $\mathcal{G}^* = \emptyset$ **then** /* generate a new line segment */
9. $\quad\quad\quad i := i - 1$; $\overline{\mathcal{T}} := \overline{\mathcal{T}} \cup \{\overrightarrow{P_sP_i}\}$; $P_s := P_i$; $t_c := P_{i+1}.t$
10. $\quad i := i + 1$;
11. $\overline{\mathcal{T}} := \overline{\mathcal{T}} \cup \{\overrightarrow{P_sP_n}\}$;
12. **return** $\overline{\mathcal{T}}$.

Procedure getRegularPolygon (P_s, P_i, r, m, t_c)
1. $c := (t_c - t_s)/(P_i.t - P_s.t)$;
2. $x := P_s.x + c \cdot (P_i.x - P_s.x)$;
3. $y := P_s.y + c \cdot (P_i.y - P_s.y)$;
4. **for** $(j := 1; j \le m; j++)$ **do**
5. $\quad \theta := (2j + 1) * \pi/m$;
6. $\quad \mathcal{G}.v_j.x := x + c \cdot r \cdot \cos\theta$;
7. $\quad \mathcal{G}.v_j.y := y + c \cdot r \cdot \sin\theta$;
8. **return** $\mathcal{G}$.

Figure 8. One-pass strong trajectory simplification algorithm.

The algorithm first initializes the start point P_s to P_0, the index i of the current data point to 1, the intersection polygon $\mathcal{G}^*$ to $\emptyset$, the output $\overline{\mathcal{T}}$ to $\emptyset$, and t_c to $P_1.t$, respectively (line 1). The algorithm sequentially processes the data points of the trajectory one by one (lines 2–10). It gets the m-inscribed regular polygon *w.r.t.* the current point P_i (line 3) by calling procedure getRegularPolygon. When $\mathcal{G}^* = \emptyset$, $\mathcal{G}^*$ is simply initialized as $\mathcal{G}$ (lines 4, 5). Otherwise, $\mathcal{G}^*$ is the intersection of the current regular polygon $\mathcal{G}$ with $\mathcal{G}^*$ by calling procedure FastRPolyInter() introduced in Section 3.3 (line 7). If the resulting intersection $\mathcal{G}^*$ is empty, then a new line segment $\overrightarrow{P_sP_{i-1}}$ is generated (lines 8–10). After the final new line segment $\overrightarrow{P_sP_n}$ is generated (line 11), it returns the simplified piece-wise line representation $\overline{\mathcal{T}}$ (line 12).

Example 4: Figure 9 shows a running example of algorithm CISED-S for compressing the trajectory $\ddot{\mathcal{T}}$ in Figure 1.

(1) After initialization, the CISED-S algorithm reads point P_1 and builds a narrow *oblique circular cone* $\mathcal{C}(P_0, \mathcal{O}(P_1, \epsilon/2))$, taking P_0 as its apex and $\mathcal{O}(P_1, \epsilon/2)$ as its base (green dash). The *circular cone* is projected on the plane $P.t - P_1.t = 0$, and the inscribe regular polygon $\mathcal{G}_1$ of the projection circle is returned. As $\mathcal{G}^*$ is empty, $\mathcal{G}^*$ is set to $\mathcal{G}_1$.

(2) The algorithm reads P_2 and builds $\mathcal{C}(P_0, \mathcal{O}(P_2, \epsilon/2))$ (red dash). The *circular cone* is also projected on the plane $P.t - P_1.t = 0$ and the inscribe regular polygon $\mathcal{G}_2$ of the projection circle is returned. As $\mathcal{G}^* = \mathcal{G}_1$ is not empty, $\mathcal{G}^*$ is set to the intersection of $\mathcal{G}_2$ and $\mathcal{G}^*$, which is $\mathcal{G}_1 \prod \mathcal{G}_2 \neq \emptyset$.

(3) For point P_3, the algorithm runs the same routing as P_2 until the intersection of $\mathcal{G}_3$ and $\mathcal{G}^*$ is $\emptyset$. Thus, line segment $\overrightarrow{P_0P_2}$ is generated, and the process of a new line segment is started, taking P_2 as the new start point and $P.t - P_3.t = 0$ as the new projection plane.

(4) At last, the algorithm outputs four continuous line segments, *i.e.*, $\{\overrightarrow{P_0P_2}, \overrightarrow{P_2P_4}, \overrightarrow{P_4P_7}, \overrightarrow{P_7P_{10}}\}$. □

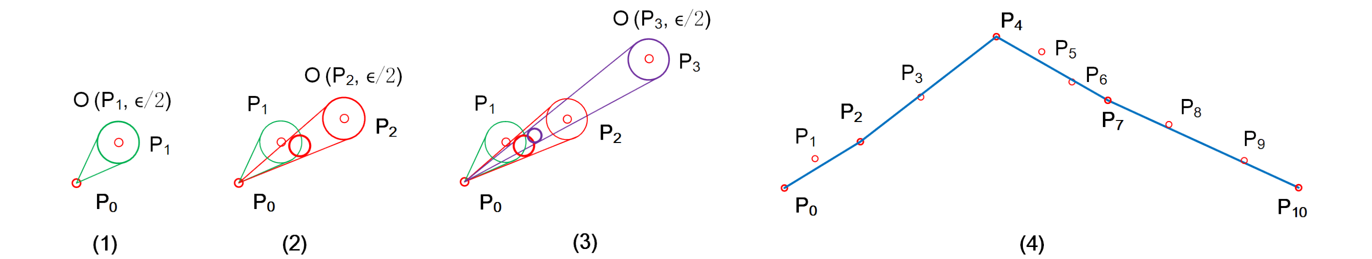


Figure 9. A running example of the CISED-S algorithm. The points and the oblique circular cones are projected on an x-y space. The trajectory $\ddot{\mathcal{T}}[P_0, \ldots, P_{10}]$ is compressed into four line segments.

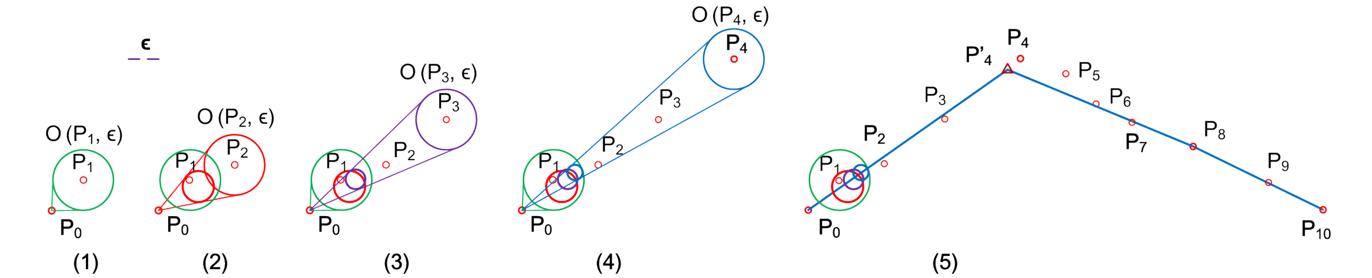


Figure 10. A running example of the CISED-W algorithm. The points and the oblique circular cones are projected on an x-y space. The trajectory $\ddot{\mathcal{T}}[P_0, \ldots, P_{10}]$ is compressed into three line segments.

4.2 Weak Trajectory Simplification

We then present the one-pass error bounded *weak simplification* algorithm using the synchronous distance.

Algorithm CISED-W. Given a trajectory $\ddot{\mathcal{T}}[P_0, \ldots, P_n]$, an error bound ϵ and the number m of edges for inscribed regular polygons, it returns a simplified trajectory $\overline{\mathcal{T}}$ which may contain interpolated points.

By Proposition 3, algorithm CISED-W generates spatio-temporal cones whose bases are circles with a radius of ϵ, and, hence, it replaces $\epsilon/2$ with ϵ (line 3 of CISED-S). It also generates new line segments with interpolated data points Q, and, hence, it replaces point P_i and line segment $\overrightarrow{P_sP_i}$ (line 9 of CISED-S) with Q and $\overrightarrow{P_sQ}$, respectively, such that Q is generated as follows.

Proposition 10: *Given a sub-trajectory $\ddot{\mathcal{T}}[P_s, \ldots, P_{s+k}]$ and an error bound ϵ, $t_c = P_{s+k}.t$ and G^*_k be the intersection polygons of all polygons G_{s+i} ($i \in [1, k]$) on the plane $P.t - t_c = 0$. If G^*_k is not empty, then any point in G^*_k is feasible for Q.* □

Proof: By Proposition 3 and the nature of inscribed regular polygon, it is easy to find that for any $Q \in G^*_k$ *w.r.t.* $t_c = P_{s+k}.t$, there is $sed(P_i, \overrightarrow{P_sQ}) \le \epsilon$ for all points P_{s+i} ($i \in [1, k]$). From this, we have the conclusion. □

The choice of Q from G^*_k may slightly affect the effectiveness (*e.g.*, average errors and compression ratios). However, the choice of an optimal Q is non-trivial. For the benefit of efficiency, we apply the following strategies.

(1) If $P_{s+k} \in G^*_k$ *w.r.t.* $t_c = P_{s+k}.t$, then Q is simply P_{s+k}.

(2) If $G^*_k \neq \emptyset$ and $P_{s+k} \notin G^*_k$ *w.r.t.* $t_c = P_{s+k}.t$, then the central point of G^*_k is chosen as Q.

(3) If $t_c \neq P_{s+k}.t$, which is the general cases, then we project the intersection polygon G^*_k *w.r.t.* $t_c \neq P_{s+k}.t$ on the plane $P.t - P_{s+k}.t = 0$, then we apply strategies (1) and (2) above. That is, the projection has no affects on the choice of Q.

Example 5: Figure 10 shows a running example of algorithm CISED-W for compressing the trajectory $\ddot{\mathcal{T}}$ in Figure 1 again.

(1) After initialization, the CISED-W algorithm reads point P_1 and builds an *oblique circular cone* $\mathcal{C}(P_0, \mathcal{O}(P_1, \epsilon))$, and projects it on the plane $P.t - P_1.t = 0$. The inscribed regular polygon $\mathcal{G}_1$ of the projection circle is returned. The intersection of polygon $\mathcal{G}_1$ and polygon $\mathcal{G}^*$ is $\mathcal{G}_1 \neq \emptyset$.

(2) P_2, P_3 and P_4 are processed in turn. The intersection polygons are not empty.

(3) For point P_5, the intersection of $\mathcal{G}_5$ and $\mathcal{G}^*$ is ϕ. Thus, line segment $\overrightarrow{P_0Q} = \overrightarrow{P_0P'_4}$ is output, and a new process is started such that $Q = P'_4$ is the new start point and plane $P.t - P_5.t = 0$ is the new projection plane.

(4) At last, the algorithm outputs 3 continuous line segments, *i.e.*, $\overrightarrow{P_0P'_4}$, $\overrightarrow{P'_4P_8}$ and $\overrightarrow{P_8P_{10}}$, in which P'_4 is an interpolated data points not in $\ddot{\mathcal{T}}$. □

Correctness and complexity analyses. The correctness of algorithms CISED-S and CISED-W follows from Propositions 3 and 9, and Propositions 3 and 10, respectively. It is easy to verify that each data point in a trajectory is only processed once, and each can be done in $O(1)$ time, as both procedures getRegularPolygon and FastRPolyInter can be done in $O(1)$ time. Hence, these algorithms are both one-pass error bounded trajectory simplification algorithms. It is also easy to see that these algorithms take $O(1)$ space.

These together also complete the proof of Theorem 8.

5 Experimental Study

In this section, we present an extensive experimental study of our strong and weak trajectory simplification algorithms (CISED-S and CISED-W) and existing algorithms of DPSED and SQUISH-E on trajectory datasets. Using four real-life trajectory datasets, we conducted three sets of experiments to evaluate: (1) the compression ratios of CISED-S and CISED-W vs. DPSED and SQUISH-E, (2) the average errors

Table 2
Real-life trajectory datasets

Data Sets	Number of Trajectories	Sampling Rates (s)	PointsPer Trajectory (K)	Total points
Truck	1,000	1-60	~ 132.7	132.7M
ServiceCar	1,000	3-5	~ 114.1	114M
GeoLife	182	1-5	~ 132.8	24.2M
PrivateCar	10	1	~ 11.8	118K

of CISED-S and CISED-W vs. DPSED and SQUISH-E, (3) the execution time of CISED-S and CISED-W vs. DPSED and SQUISH-E, and (4) the impacts of polygon intersection algorithms FastRPolyInter and CPolyInter and the edge number m of inscribed regular polygons to the effectiveness and efficiency of algorithms CISED-S and CISED-W.

5.1 Experimental Setting

Real-life Trajectory Datasets. We use four real-life datasets shown in Table 2 to test our solutions.

(1) Truck trajectory data (Truck) is the GPS trajectories collected by trucks equipped with GPS sensors in China during a period from Mar. 2015 to Oct. 2015. The sampling rate varied from 1s to 60s.

(2) Service car trajectory data (ServiceCar) is the GPS trajectories collected by a car rental company during Apr. 2015 to Nov. 2015. The sampling rate was one point per 3–5 seconds, and each trajectory has around $114.1K$ points.

(3) GeoLife trajectory data (GeoLife) is the GPS trajectories collected in GeoLife project [29] by 182 users in a period from Apr. 2007 to Oct. 2011. These trajectories have a variety of sampling rates, among which 91% are logged in each 1-5 seconds or each 5-10 meters per point.

(4) Private car trajectory data (PrivateCar) is a small set GPS trajectories collected with a high sampling rate of one point per second by our team members in 2017. There are 10 trajectories and each trajectory has around 11.8K points.

Algorithms and implementation. We implement four line simplification algorithms, *i.e.,* our CISED-S and CISED-W, DPSED [11] (the most effective existing LS algorithm using SED), and SQUISH-E [14] (the most efficient existing LS algorithm using SED). We also implement the polygon intersection algorithms, CPolyInter and our FastRPolyInter. All algorithms were implemented with Java. All tests were run on an x64-based PC with 8 Intel(R) Core(TM) i7-6700 CPU @ 3.40GHz and 8GB of memory, and each test was repeated over 3 times and the average is reported here.

5.2 Experimental Results

5.2.1 Evaluation of Compression Ratios

In the first set of tests, we evaluate the impacts of parameter m on the compression ratios of our algorithms CISED-S and CISED-W, and compare the compression ratios of CISED-S and CISED-W with DPSED and SQUISH-E.

The compression ratio is defined as follows: Given a set of trajectories $\{\dddot{\mathcal{T}}_1, \dots, \dddot{\mathcal{T}}_M\}$ and their piecewise line representations $\{\overline{\mathcal{T}_1}, \dots, \overline{\mathcal{T}_M}\}$, the compression ratio of an algorithm is $(\sum_{j=1}^{M} |\overline{\mathcal{T}}_j|)/(\sum_{j=1}^{M} |\dddot{\mathcal{T}}_j|)$. By the definition, *algorithms with lower compression ratios are better*.

Exp-1.1: Impacts of parameter m on compression ratios. To evaluate the impacts of the number m of edges of polygons on the compression ratios of algorithms CISED-S and CISED-W, we fixed the error bounds $\epsilon = 60$ meters, and varied m from 4 to 40. The results are reported in Figure 11.

(1) Algorithms CISED-S and CISED-W using FastRPolyInter have the same compression ratios as their counterparts using CPolyInter for all cases.

(2) When varying m, the compression ratios of CISED-S and CISED-W decrease with the increase of m on all datasets.

(3) When varying m, the compression ratios of algorithms CISED-S and CISED-W decrease (a) fast when $m < 12$, (b) slowly when $m \in [12, 20]$, and (c) very slow when $m > 20$. Hence, *the region range of* $[12, 20]$ *is the good candidate region for* m *in terms of compression ratios*. Here the compression ratio of m=12 is only on average 100.88% of m=20.

Exp-1.2: Impacts of the error bound ϵ on compression ratios. To evaluate the impacts of ϵ on compression ratios, we fixed m=16, the middle of $[12, 20]$, and varied ϵ from 10 meters to 200 meters on the entire four datasets, respectively. The results are reported in Figure 12 .

(1) When increasing ϵ, the compression ratios of all these algorithms decrease on all datasets.

(2) PrivateCar has the lowest compression ratios, compared with Truck, ServiceCar and GeoLife, due to its highest sampling rate, Truck has the highest compression ratios due to its lowest sampling rate, and ServiceCar and GeoLife have the compression ratios in the middle accordingly.

(3) Algorithm CISED-S has better compression ratios than SQUISH-E, and is comparable with DPSED, on all datasets and for all ϵ. The compression ratios of CISED-S are on average (91.8%, 79.3%, 71.9%, 72.7%) and (113.2%, 109.2%, 108.0%, 109.1%) of SQUISH-E and DPSED on (Truck, ServiceCar, GeoLife, PrivateCar), respectively. For example, when $\epsilon = 40$ meters, the compression ratios of SQUISH-E, CISED-S and DPSED are (31.3%, 19.9%, 8.0%, 4.9%), (30.0%, 16.1%, 5.8%, 3.6%) and (26.9%, 14.7%, 5.4%, 3.4%) on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

(4) Algorithm CISED-W has the best compression ratios on all datasets and for all ϵ. The compression ratios of CISED-W are on average (64.4%, 57.7%, 53.8%, 54.6%) and (79.2%, 79.5%, 80.9%, 82.0%) of SQUISH-E and DPSED on (Truck, ServiceCar, GeoLife, PrivateCar), respectively. For example, when ϵ = 40 meters, the compression ratios of CISED-W are (21.8%, 11.5%, 4.3%, 2.7%) on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

Exp-1.3: Impacts of trajectory sizes on compression ratios. To evaluate the impacts of trajectory size, *i.e.,* the number of data points in a trajectory, on compression ratios, we chose 10 trajectories from Truck, ServiceCar, GeoLife and PrivateCar, respectively, fixed m=16 and ϵ=60 meters, and varying the size $|\dddot{\mathcal{T}}|$ of trajectories from $1K$ points to $10K$ points. The results are reported in Figure 13.

(1) The compression ratios of these algorithms from the best to the worst are CISED-W, DPSED, CISED-S and SQUISH-E, on all datasets and for all sizes of trajectories.

(2) The size of input trajectories has few impacts on the compression ratios of LS algorithms on all datasets.

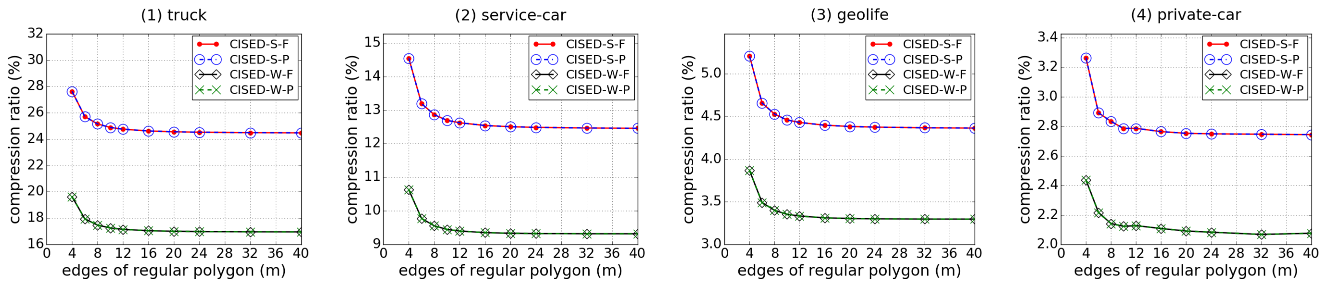


Figure 11. Evaluation of compression ratios: fixed error bound with $\epsilon = 60$ meters and varying m.

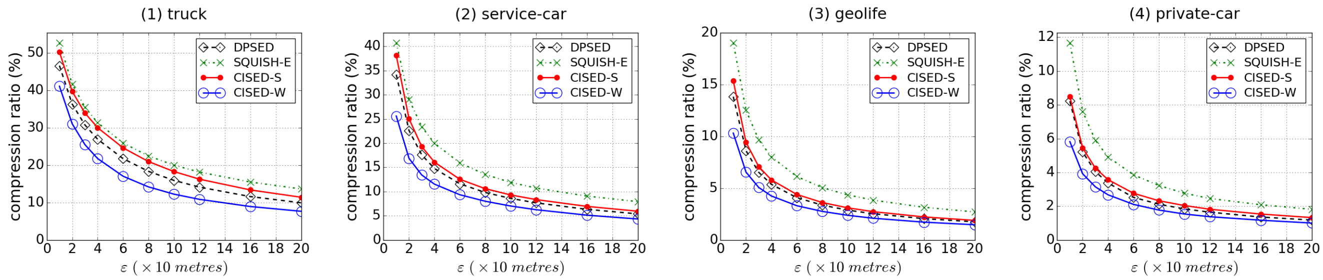


Figure 12. Evaluation of compression ratios: fixed with $m = 16$ and varying error bound ϵ.

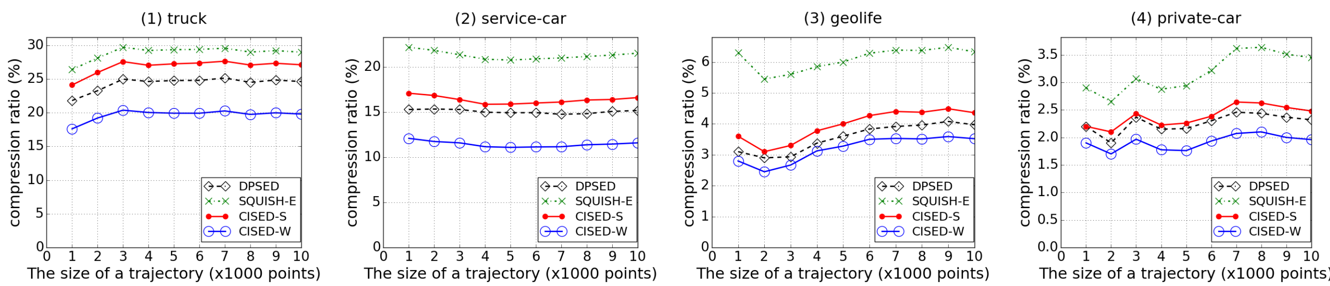


Figure 13. Evaluation of compression ratios: fixed with $m = 16$ and $\epsilon = 60$ meters, and varying the size of trajectories.

5.2.2 Evaluation of Average Errors

In the second set of tests, we first evaluate the impacts of parameter m on the average errors of algorithms CISED-S and CISED-W, then compare the average errors of our algorithms CISED-S and CISED-W with DPSED and SQUISH-E.

Given a set of trajectories $\{\ddot{\mathcal{T}}_1, \ldots, \ddot{\mathcal{T}}_M\}$ and their piecewise line representations $\{\overline{\mathcal{T}_1}, \ldots, \overline{\mathcal{T}}_M\}$, and point $P_{j,i}$ denoting a point in trajectory $\ddot{\mathcal{T}}_j$ contained in a line segment $\mathcal{L}_{l,i} \in \overline{\mathcal{T}_l}$ ($l \in [1, M]$), then the average error is $\sum_{j=1}^{M}\sum_{i=0}^{M} d(P_{j,i}, \mathcal{L}_{l,i}) / \sum_{j=1}^{M} |\ddot{\mathcal{T}}_j|$.

Exp-2.1: Impacts of parameter m on average errors. To evaluate the impacts of parameter m on average errors of algorithms CISED-S and CISED-W, we fixed the error bounds $\epsilon = 60$ meters, and varied m from 4 to 40. The results are reported in Figure 14.

(1) Algorithms CISED-S and CISED-W using FastRPolyInter have the same average errors as their counterparts using CPolyInter on all datasets and for all m.

(2) When varying m, the average errors of CISED-S and CISED-W increase with the increase of m on all datasets.

(3) When varying m, it is similar to compression ratios, and the average errors increases (a) fast when $m < 12$, (b) slowly when $m \in [12, 20]$, and (c) very slowly when $m > 20$. *The range of* $[12, 20]$ *is also the good candidate region for* m *in terms of errors*. Here the average error of $m = 12$ is only on average 98.4% of $m = 20$.

Exp-2.2: Impacts of the error bound ϵ on average errors. To evaluate the average errors of these algorithms, we fixed m=16, and varied ϵ from 10 meters to 200 meters on the entire Truck, ServiceCar, GeoLife and PrivateCar, respectively. The results are reported in Figure 15.

(1) Average errors increase with the increase of ϵ.

(2) Algorithms CISED-S and CISED-W have lager average errors compared with DPSED and SQUISH-E. The average errors of CISED-S and CISED-W are on average (96.9%, 119.3%, 127.7%, 137.9%) and (197.4%, 210.1%, 207.5%, 217.5%) of DPSED and (160.5%, 188.2%, 215.2%, 180.3%) and (326.7%, 331.1%, 349.7%, 284.2%) of SQUISH-E on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

(3) When the error bound of CISED-W is set as the half of CISED-S, the average errors of algorithm CISED-W are on average (93.8%, 86.0%, 81.4%, 79.4%) of CISED-S on (Truck, ServiceCar, GeoLife, PrivateCar), respectively, meaning that the large average errors of CISED-W are caused by its cone *w.r.t.* ϵ compared with the narrow cone *w.r.t.* $\epsilon/2$ of CISED-S.

Exp-2.3: Impacts of trajectory sizes on average errors. To evaluate the impacts of trajectory sizes on average errors, we chose the same 10 trajectories from Truck, ServiceCar, GeoLife and PrivateCar, respectively. We fixed m=16 and $\epsilon =$

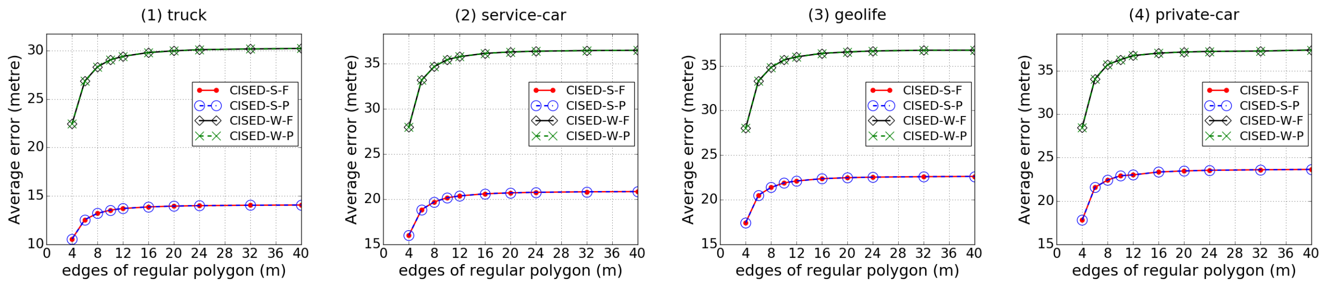


Figure 14. Evaluation of average errors: fixed error bound with $\epsilon = 60$ meters and varying m.

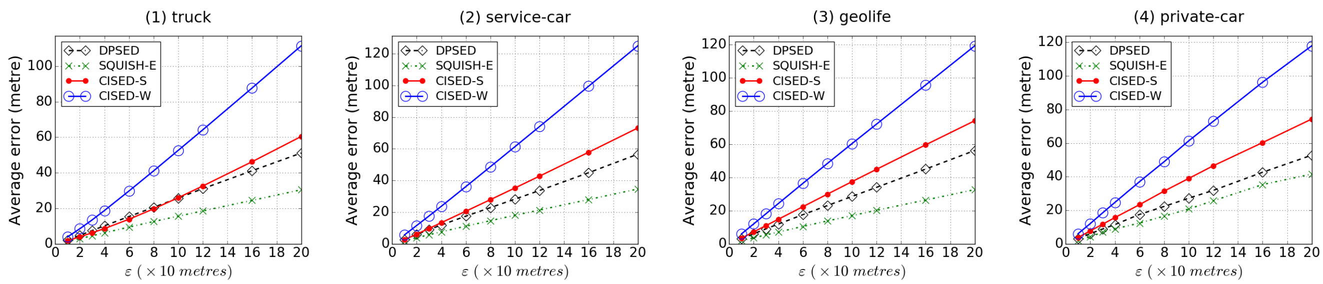


Figure 15. Evaluation of average errors: fixed with $m = 16$ and varying error bound ϵ.

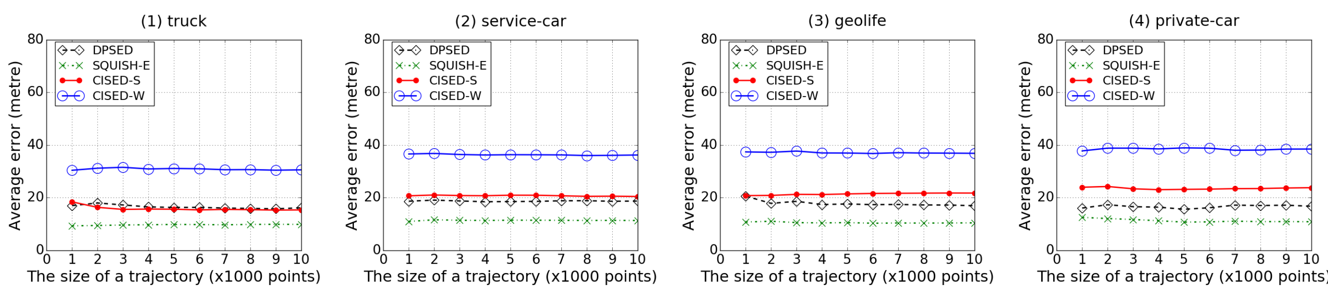


Figure 16. Evaluation of average errors: fixed with $m = 16$ and $\epsilon = 60$ meters, and varying the size of trajectories.

60 meters, and varying the size $|\dddot{\mathcal{T}}|$ of trajectories from $1K$ points to $10K$ points. The results are reported in Figure 16.

(1) The average errors of these algorithms from the smallest to the largest are SQUISH-E, DPSED, CISED-S and CISED-W, on all datasets and for all sizes.

(2) The size of input trajectories has few impacts on the average errors of LS algorithms on all datasets.

5.2.3 *Evaluation of Efficiency*

In the last set of tests, we evaluate the impacts of parameter m on the efficiency of algorithms CISED-S and CISED-W, and compare the efficiency of our approaches CISED-S and CISED-W with algorithms DPSED and SQUISH-E.

Exp-3.1: Impacts of algorithm FastRPolyInter **and parameter m on efficiency.** To evaluate the impacts of algorithm FastRPolyInter and parameter m on algorithms CISED-S and CISED-W, we equipped CISED-S and CISED-W with FastRPolyInter and CPolyInter, respectively, fixed $\epsilon = 60$ meters, and varied m from 4 to 40. The results are reported in Figure 17 and 18.

(1) The algorithms CISED-S and CISED-W spend the most time in the executing of polygon intersections. For all m, the execution time of algorithms CPolyInter and FastRPolyInter is on average (92.5%, 93.5%, 96.0%, 97.0%) and (89.0%, 90.5%, 92.5%, 96.5%) of the entire compression time on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

(2) FastRPolyInter runs faster than CPolyInter on all datasets and for all m. The execution time of algorithms CISED-S-FastRPolyInter and CISED-W-FastRPolyInter is one average 81.3% their counterparts with CPolyInter.

(3) When varying m, the execution time of algorithms CISED-S-FastRPolyInter, CISED-S-CPolyInter, CISED-W-FastRPolyInter and CISED-W-CPolyInter increases approximately linearly with the increase of m on all the datasets.

(4) The running time of $m = 12$ is on average 71.2% of $m = 20$ for algorithms CISED-S and CISED-W on all datasets.

Exp-3.2: Impacts of the error bound ϵ on efficiency. To evaluate the impacts of ϵ on efficiency, we fixed $m = 16$, and varied ϵ from 10 meters to 200 meters on Truck, ServiceCar, GeoLife and PrivateCar, respectively. The results are reported in Figure 19.

(1) All algorithms are not very sensitive to ϵ on any datasets, and algorithm DPSED is more sensitive to ϵ than the other three algorithms. The running time of DPSED decreases a little bit with the increase of ϵ, as the increment of ϵ decreases the number of partitions of the input trajectory.

(2) Algorithms CISED-S and CISED-W are obviously faster than DPSED and SQUISH-E for all cases. They are on average (20.7, 14.2, 18.2, 10.0) times faster than DPSED, and (2.7, 2.8, 3.4, 2.9) times faster than SQUISH-E on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

Exp-3.3: Impacts of trajectory sizes on efficiency. To eval-

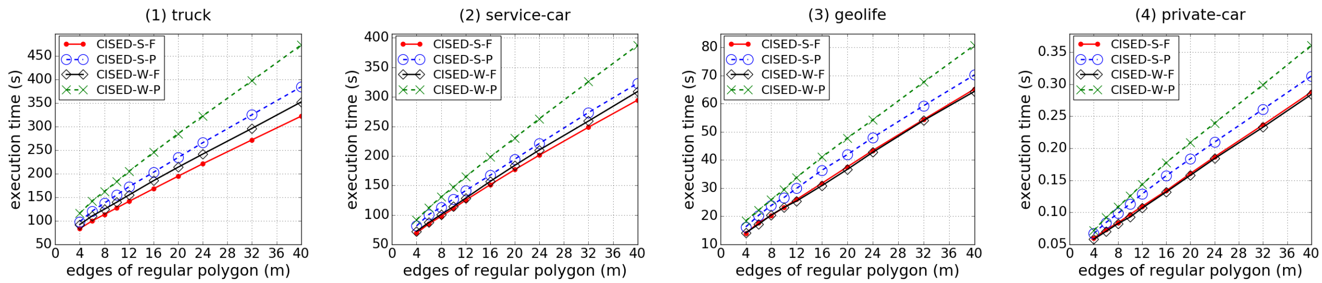


Figure 17. Evaluation of running time: fixed error bound with $\epsilon = 60$ meters, and varying m. Here "F" denotes our fast regular polygon intersection algorithm FastRPolyInter, and "P" denotes CPolyInter, repectively.

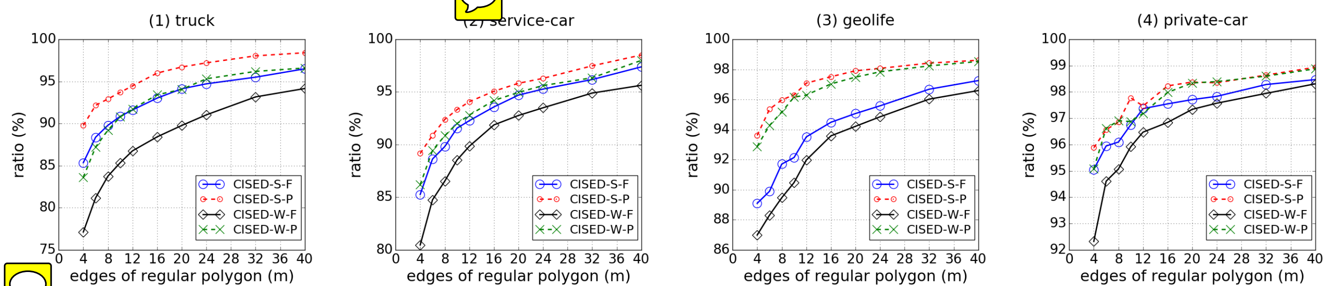


Figure 18. Evaluation of running time of polygon intersection algorithms: fixed error bound with $\epsilon = 60$ meters, and varying m.

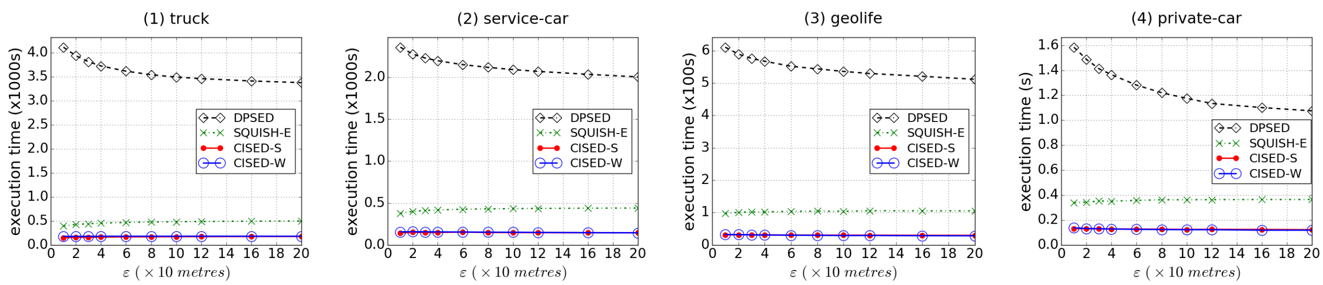


Figure 19. Evaluation of running time: fixed with $m = 16$ and varying error bounds ϵ.

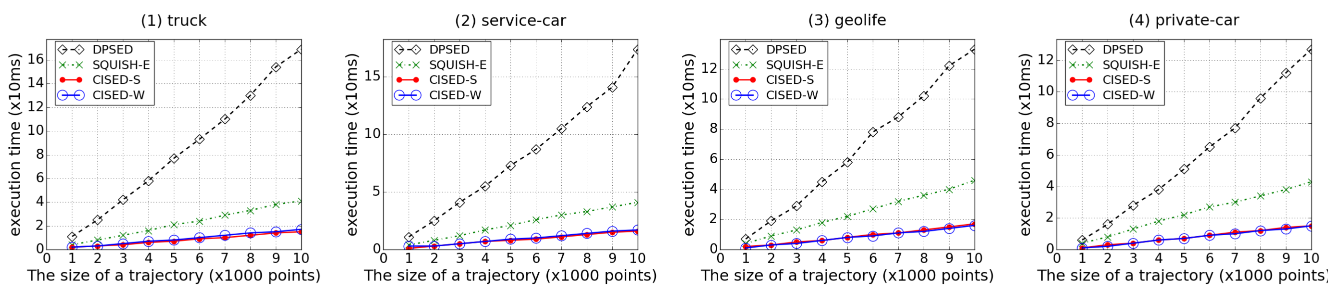


Figure 20. Evaluation of running time: fixed with $m = 16$ and $\epsilon = 60$ meters, and varying the size of trajectories.

uate the impacts of trajectory sizes on execution time, we chose 10 trajectories from Truck, ServiceCar, GeoLife and PrivateCar, respectively, fixed $m = 16$ and $\epsilon = 60$ meters, and varying the size $|\dddot{\mathcal{T}}|$ of trajectories from $1K$ points to $10K$ points. The results are reported in Figure 20.

(1) Algorithms CISED-S and CISED-W scale well with the increase of the size of trajectories on all datasets, and have a linear running time, while algorithm DPSED does not. This is consistent with their time complexity analyses.

(2) Algorithms CISED-S and CISED-W are the fastest SED enabled LS algorithms, and are (5.5–10.6, 7.3–10.6, 5.2–8.4, 6.0–8.5) times faster than DPSED, and (2.0–2.8, 2.4–2.7, 2.8–3.0, 2.8–4.0) times faster than SQUISH-E on the selected $1K$ to $10K$ points datasets (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

(3) The efficiency advantage of algorithms CISED-S and CISED-W increases with the increase of trajectory sizes.

Summary. From these tests we find the following.

(1) Polygon intersection Algorithms. FastRPolyInter runs faster than CPolyInter, and is as effectiveness as CPolyInter.

(2) Parameter m. The compression ratio decreases with the increase of m, and the running time increases approximately linearly with the increase of m. In practice, the range of $[12, 20]$ is a good candidate region for parameter m.

(3) Compression ratios. Algorithm CISED-S is comparable with DPSED and algorithm CISED-W is better than DPSED. Both of them are better than SQUISH-E. The compression

ratios of algorithms CISED-S and CISED-W are on average (91.8%, 79.3%, 71.9%, 72.7%) and (64.4%, 57.7%, 53.8%, 54.6%) of SQUISH-E and (113.2%, 109.2%, 108.0%, 109.1%) and (79.2%, 79.5%, 80.9%, 82.0%) of DPSED on (Truck, ServiceCar, GeoLife, PrivateCar), respectively.

(4) Average errors. Algorithms CISED-S and CISED-W both have higher average errors than the other algorithms for the benefit of efficieny, and CISED-W has obvious higher average errors than CISED-S as the former essentially forms spatio-temporal cones with a radius of ϵ.

(5) Running time. Algorithms CISED-S and CISED-W are on average (20.7, 14.2, 18.2, 10.0) and (2.7, 2.8, 3.4, 2.9) times faster than algorithms DPSED and SQUISH-E on (Truck, ServiceCar, GeoLife, PrivateCar), respectively. The efficiency advantage of algorithms CISED-S and CISED-W also increases with the increase of the trajectory size.

6 Related Work

The idea of piece-wise line simplification comes from computational geometry. Its target is to approximate a given finer piece-wise linear curve by another coarser piece-wise linear curve, which is typically a subset of the former, such that the maximum distance of the former to the later is bounded by a user specified bound ϵ. The optimal methods that find the minimal number of points or segments have the time complexity of $O(n^2)$ where n is the number of the original points [2]. They are time-consuming and impractical for large inputs [6]. Hence, many studies have been targeting at finding the sub-optimal results. In particular, the state-of-the-art sub-optimal line simplification approaches fall into three categories, *i.e., batch algorithms*, *online algorithms* and *one-pass algorithms*.

For trajectory compression, there are two types of widely used distance metrics: perpendicular Euclidean distances (PED) and synchronous Euclidean distances (SED). Mostly existing line simplification algorithms use PED, and more attentions are needed to be paid on SED. SED was first introduced in the name of *time-ratio distance* in [11], and formally presented in [19] as the *synchronous Euclidean distance*.

We next introduce these line simplification algorithms from the aspect of the three categories.

Batch algorithms. The batch algorithms adopt a global distance checking policy that requires all trajectory points are loaded before compressing starts. These batch algorithms can be either top-down or bottom-up.

Top-down algorithms, e.g., Ramer [20] and Douglas-Peucker [4], recursively divide a trajectory into sub-trajectories until the stopping condition is met. Bottom-up algorithms, e.g., Theo Pavlidis' algorithm [17], is the natural complement of the top-down ones, which recursively merge adjacent sub-trajectories with the smallest distance, initially $n/2$ sub-trajectories for a trajectory with n points, until the stopping condition is met. The distances of newly generated line segments are recalculated during the process. These batch algorithms originally only support PED, but are easy to be extended to support SED [11]. The batch nature and high time complexities make batch algorithms impractical for online scenarios and resource-constrained devices [8].

Online algorithms. The online algorithms adopt a constrained global distance checking policy that restricts the checking within a sliding or opening window. Constrained global checking algorithms do not need to have the entire trajectory ready before they start compressing, and are more appropriate than batch algorithm for compressing trajectories for online scenarios.

Several line simplification algorithms have been developed, e.g., by combining DP or Theo Pavlidis' with sliding or opening windows for online processing [11]. These methods still have a high time and/or space complexity, which significantly hinders their utility in resource-constrained mobile devices [9]. BQS [9] and SQUISH-E [14] further optimize the opening window algorithms. BQS [9] fast the processing by picking out at most eight special points from an open window based on a convex hull, which, however, hardly supports SED. The SQUISH-E [14] algorithm is an combination of opening window and bottom-up online algorithm. It uses a doubly linked list Q to achieve a better efficiency. Although SQUISH-E supports SED, it is not one-pass, and has a relatively poor compression ratio.

One-pass algorithms. The one-pass algorithms adopt a local distance checking policy. They do not need a window to buffer the previously read points as they process each point in a trajectory once and only once. Obviously, the one-pass algorithms run in linear time and constant space.

The n–th point routine and the routine of random-selection of points [23] are two naive one-pass algorithms. In these routines, for every fixed number of consecutive points along the line, the n–th point and one random point among them are retained, respectively. They are fast, but are obviously not error bounded. In Reumann-Witkam routine [21], it builds a strip paralleling to the line connecting the first two points, then the points within this strip compose one section of the line. The Reumann-Witkam routine also runs fast, but has limited compression ratios. The sector intersection (SI) algorithm [24], [27] was developed for graphic and pattern recognition in the late 1970s, for the approximation of arbitrary planar curves by linear segments or finding a polygonal approximation of a set of input data points in a 2D Cartesian coordinate system. [5] optimized algorithm SI by considering the distance between a potential end point and the initial point of a line segment, and the Sleeve algorithm [28] in the cartographic discipline essentially applies the same idea as the SI algorithm. Moreover, fast BQS [9] (FBQS in short), the simplified version of BQS, has a linear time complexity. The authors of this article also developed an One-Pass ERror Bounded (OPERB) algorithm [8]. However, all existing one-pass algorithms use PED [5], [8], [9], [24], [27], [28], while this study focuses on SED.

Semantics based trajectory compression methods are orthogonal to line simplification based methods (see [8] for more details), and may be combined with each other to further improve the effectiveness of trajectory compression.

7 Conclusions

We have proposed CISED-S and CISED-W, two one-pass error bounded strong and weak trajectory simplification algorithms using the synchronous distance. We have also experimentally verified that algorithms CISED-S and CISED-W are both efficient and effective. They are three times faster than SQUISH-E, the most efficient existing LS algorithm using SED enabled. In terms of compression ratio, CISED-S is comparable with DPSED, the existing LS algorithm with the best compression ratio, and is 21.1% better than SQUISH-E on average, and CISED-W is on average 19.6% and 42.4% better than DPSED and SQUISH-E, respectively.

ACKNOWLEDGMENTS

This work is supported in part by NSFC (U1636210), 973 program (2014CB340300) and NSFC (61421003).

REFERENCES

[1] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal data reduction with deterministic error bounds. *VLDBJ*, 15(3):211–228, 2006.
[2] W. Chan and F. Chin. Approximation of polygonal curves with minimum number of line segments or minimal error. *International Journal of Computational Geometry Applicaitons*, 6(1):378–387, 1996.
[3] Y. Chen, K. Jiang, Y. Zheng, C. Li, and N. Yu. Trajectory simplification method for location-based social networking services. In *LBSN*, pages 33–40, 2009.
[4] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, 1973.
[5] J. G. Dunham. Piecewise linear approximation of plannar curves. *PAMI*, 8, 1986.
[6] P. S. Heckbert and M. Garland. Survey of polygonal surface simplification algorithms. In *SIGGRAPH*, 1997.
[7] J. Hershberger and J. Snoeyink. Speeding up the douglas-peucker line-simplification algorithm. *Technical Report, University of British Columbia*, 1992.
[8] X. Lin, S. Ma, H. Zhang, T. Wo, and J. Huai. One-pass error bounded trajectory simplification. *PVLDB*, 10(7):841–852, 2017.
[9] J. Liu, K. Zhao, P. Sommer, S. Shang, B. Kusy, and R. Jurdak. Bounded quadrant system: Error-bounded trajectory compression on the go. In *ICDE*, 2015.
[10] C. Long, R. C.-W. Wong, and H. Jagadish. Direction-preserving trajectory simplification. *PVLDB*, 6(10):949–960, 2013.
[11] N. Meratnia and R. A. de By. Spatiotemporal compression techniques for moving point objects. In *EDBT*, 2004.
[12] R. Metha and V.K.Mehta. *The Principles of Physics*. S Chand, 1999.
[13] J. Muckell, J.-H. Hwang, C. T. Lawson, and S. S. Ravi. Algorithms for compressing gps trajectory data: an empirical evaluation. In *ACM-GIS*, 2010.
[14] J. Muckell, P. W. Olsen, J.-H. Hwang, C. T. Lawson, and S. S. Ravi. Compression of trajectory data: a comprehensive evaluation and new approach. *GeoInformatica*, 18(3):435–460, 2014.
[15] A. Nibali and Z. He. Trajic: An effective compression system for trajectory data. *TKDE*, 27(11):3138–3151, 2015.
[16] J. O'Rourke, C. B. Chien, T. Olson, and D. Naddor. A new linear algorithm for intersecting convex polygons. *Computer Graphics and Image Processing*, 19(4):384–391, 1982.
[17] T. Pavlidis and S. L. Horowitz. Segmentation of plane curves. *IEEE Transactions on Computers.*, 23(8):860–870, 1974.
[18] I. S. Popa, K. Zeitouni, VincentOria, and A. Kharrat. Spatio-temporal compression of trajectories in road networks. *GeoInformatica*, 19(1):117–145, 2014.
[19] M. Potamias, K. Patroumpas, and T. K. Sellis. Sampling trajectory streams with spatiotemporal criteria. In *SSDBM*, 2006.
[20] U. Ramer. An iterative procedure for the polygonal approximation of plane curves. *Comput. Graphics Image Processing*, 1:244–256, 1972.
[21] K. Reumann and A. Witkam. Optimizing curve segmentation in computer graphics. In *International Computing Symposium*, 1974.
[22] Shamos, M. Ian, and H. Dan. Geometric intersection problems. In *Symposium on Foundations of Computer Science*, pages 208–215, 1976.
[23] W. Shi and C. Cheung. Performance evaluation of line simplification algorithms for vector generalization. *Cartographic Journal*, 43(1):27–44, 2006.
[24] J. Sklansky and V. Gonzalez. Fast polygonal approximation of digitized curves. *Pattern Recognition*, 12:327–331, 1980.
[25] R. Song, W. Sun, B. Zheng, and Y. Zheng. Press: A novel framework of trajectory compression in road networks. *PVLDB*, 7(9):661–672, 2014.
[26] G. Trajcevski, H. Cao, P. Scheuermanny, O. Wolfsonz, and D. Vaccaro. On-line data reduction and the quality of history in moving objects databases. In *MobiDE*, 2006.
[27] C. M. Williams. An efficient algorithm for the piecewise linear approximation of planar curvers. *Computer Graphics and Image Processing*, 8:286–293, 1978.
[28] Z. Zhao and A. Saalfeld. Linear-time sleeve-fitting polyline. In *Proceedings of AutoCarto*, pages 214–223, 1997.
[29] Y. Zheng, X. Xie, and W. Ma. GeoLife: A collaborative social networking service among user, location and trajectory. *IEEE Data Eng. Bull.*, 33(2):32–39, 2010.

PLACE PHOTO HERE

Xuelian Lin is an assistant professor at the School of Computer Science and Engineering, Beihang University. He received his PhD and Master degree from Beihang University in 2013 and 2002, respectively. He was a visiting researcher of Baidu. His current research interests include distributed systems, large scale data management and processing systems.

PLACE PHOTO HERE

Jiahao Jiang is a master student in the School of Computer Science and Engineering, Beihang University. He received his BS degree in computer science and technology from Beihang University in 2016. His current research interests include databases and social data analysis.

PLACE PHOTO HERE

Shuai Ma is a professor at the School of Computer Science and Engineering, Beihang University, China. He obtained his PhD degrees from University of Edinburgh in 2010, and from Peking University in 2004, respectively. He was a postdoctoral research fellow in the database group, University of Edinburgh, a summer intern at Bell labs, Murray Hill, USA and a visiting researcher of MRSA. He is a recipient of the best paper award for VLDB 2010 and the best challenge paper award for WISE 2013. His current research interests include database theory and systems, social data and graph analysis, and data intensive computing.

PLACE PHOTO HERE

Yimeng Zuo is a master student in the School of Computer Science and Engineering, Beihang University. She received his BS degree in computer science and technology from Beihang University in 2015. Her current research interests include databases and social data analysis

PLACE PHOTO HERE

Chunming Hu is an associate professor at the School of Computer Science and Engineering, Beihang University. He received his PhD degree from Beihang University in 2006. His current research interests include distributed systems, system virtualization, large scale data management and processing systems.