Chapter 1
Trajectory Preprocessing

Wang-Chien Lee and John Krumm

Abstract A spatial trajectory is a sequences of (x,y) points, each with a time stamp. This chapter discusses low-level preprocessing of trajectories. First, it discusses how to reduce the size of data required to store a trajectory, in order to save storage costs and reduce redundant data. The data reduction techniques can run in a batch mode after the data is collected or in an on-line mode as the data is collected. Part of this discussion consists of methods to measure the error introduced by the data reduction techniques. The second part of the chapter discusses methods for filtering spatial trajectories to reduce measurement noise and to estimate higher level properties of a trajectory like its speed and direction. The methods include mean and median filtering, the Kalman filter, and the particle filter.

1.1 Introduction

Owing to the rapid advent of wireless communication and mobile computing technologies, the vision of pervasive computing is becoming a reality. Mobile devices, including smart phones, PDAs, navigational systems on vehicles, and RFIDs on cargos, have played a growing important role in various applications in our daily life. Nowadays, many of these mobile devices have location positioning and wireless communicating capabilities and thus are able to locally log or dynamically report their locations to the server.[1] Indeed, there is a tremendous demand for location tracking of moving objects from various *location-based services (LBS)*,

Wang-Chien Lee
Department of Computer Science and Engineering, The Pennsylvania State University, University Park, PA 16802, USA. e-mail: wlee@cse.psu.edu

John Krumm
Microsoft Research, Redmond, WA 98052, USA. e-mail: jckrumm@microsoft.com

[1] We call these mobile devices (and their carriers) *location-aware moving objects* or *moving objects* in short.

ranging from fleet management, traffic information services, transportation logistics, location-based games, to location-based social networks. According to forecast of the 2011-2015 LBS market made by Pyramid Research, revenue of the global location-based services market is expected to reach US10.3 billions in 2015, up from 2.8 billions in 2010.[2]

To support LBS applications, the database community has made a tremendous research effort on the development of mobile object databases (MODs) in the past decade [33, 14, 13, 28]. In addition to the conventional search functions of moving objects, *trajectory management* are essential operations of MODs since many LBS applications require analyzing and mining moving phenomenons/patterns of the monitored objects. Thus, trajectories of moving objects, i.e. their geographical-temporal traces, are often treated as first-class citizens in MODs. Due to the need to acquire and preprocess trajectory data before loading them into the MODs, in this book chapter, we review a number of issues and techniques for trajectory preprocessing, including trajectory data generation, filtering and reduction.

Based on Wikipedia, a trajectory is the path that a moving object follows through space as a function of time. To capture the accurate and complete trajectory of a moving object, however, is very difficult and expensive due to the inherent limitations of data acquisition and storage mechanisms. As a result, the continuous movement of an object is usually obtained in an approximate form as discrete samples of *spatio-temporal location points* (or simply *location points*). Supposedly the more sample points are acquired in a trajectory, the more accurate the trajectory is. However, adopting high sampling rates in acquiring the location points of moving objects to generate the trajectories may result in a massive amount of data leading to enormous overheads in data storage, communications and processing. Take the Taipei eBus system as an example.[3] The system tracks the trajectories of about 4000 buses daily, covering 287 bus routes in the greater Taipei metropolitan area. The locations of buses tracked in the system are transmitted to the system server every 15-25 seconds, generating millions of sampled data points daily. As the positioning technology and processing power of data acquisition mechanisms continue to advance rapidly, the problem of data explosion gets only worse. Hence, it's a mandate to employ the data reduction techniques in trajectory preprocessing.

In addition to data reduction, trajectories can benefit from filtering to reduce noise and estimate higher-level properties like speed and direction. Since trajectories are normally measured by a sensor, they inevitably have some error, including occasional outliers. Simple techniques like mean and median filtering can reduce these errors. In addition to error reduction, certain filters like the Kalman filter and particle filter can also give error estimates and inferences on speed and direction.

Figure 1.1 shows a high-level system model for typical location-based services. As shown, the system consists of three components: 1) the location server, 2) moving objects, and 3) LBS applications. As in most pervasive computing applications, we assume wireless communications between the server and moving objects. In

[2] See http://www.pyramidresearch.com/store/Report-Location-Based-Services.htm

[3] http://www.e-bus.taipei.gov.tw/

such systems, the locations of tracked moving objects are reported to the location server in accordance with the adopted reporting schemes, e.g. periodically. The location point data (which form trajectories of moving objects) are then uploaded to the moving object databases. On the other hand, the LBS applications submit queries to the location server to retrieve moving objects of interests (as well as their attributes such as locations and other phenomenons/patterns discovered from moving behaviors of objects) to meet various application needs.

As discussed earlier, systems in support of location based services naturally generate enormous volumes of data with measurement noise. Consequently, the data reduction and filtering techniques are particularly important for cleansing, transmission, and storage of trajectory data in location based services. Even though object movement is continuous, the representation of object trajectories is inevitably in a discrete form due to the nature of sampling-based data acquisition approach. Thus, an intuitive strategy to reduce the volumes of trajectory data is to reduce the sampling rate of data acquisition or to reduce the number of sample points in the trajectory representation. However, the question is whether we are able to discard some sample points without sacrificing the quality of trajectory data required for supporting the targeted applications. Additionally, what techniques can be used to effectively filter measurement noise not only in the raw location points of trajectories but also in high-level properties of trajectories such as direction and speed. Fortunately, due to the linear characteristics of the underlying transportation infrastructure, object movements in many LBS applications exhibit predictable patterns. As a result, many redundant and erroneous information can be removed from the trajectory without compromising much of the application requirements.

In the following, we review some trajectory data reduction strategies for LBSs. We first consider the location update scenarios and then review the data reduction strategies under these scenarios. Accordingly, we classify the data reduction strategies into two categories: 1) off-line compression and 2) on-line reporting.

After discussing data reduction, we review filtering techniques, including mean and median filtering, the Kalman filter, and the particle filter.

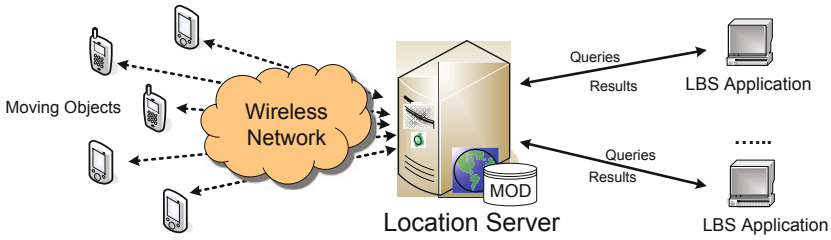


Fig. 1.1 A high-level system model for typical location-based services. The locations of tracked moving objects are reported to the location server via wireless communications. The LBS applications submit queries to the server to retrieve moving object data for analysis or other application needs.

1.2 Trajectory Data Generation

A trajectory is the path that a moving object follows through space as a function of time. Thus, it can be captured as a time-stamped series of location points, denoted as $\{\langle x_1, y_1, t_1\rangle, \langle x_2, y_2, t_2\rangle, ..., \langle x_N, y_N, t_N\rangle\}$ where x_i, y_i represent geographic coordinates of the moving object at time t_i and N is the total number of elements in the series. To generate the trajectory, a moving object needs to acquire its coordinates x, y at time t. There are many positioning technologies, e.g. global positioning system (GPS), that can be employed to determine the location of a moving object. For example, most of the smart phones already have a built-in GPS receiver and thus can easily derive their own locations. Augmenting the GPS, some positioning techniques have used wifi access points as the underlying reference system to determine the location of a moving object. Additionally, even in situations where the GPS signal is poor and there is no nearby wifi access point to serve as a positioning reference, the "dead reckoning" techniques can be used to estimate the position of a moving object by advancing from a known position using course, speed, time and distance to be traveled. In other words, where a moving object will be at a certain time can be derived based upon known or estimated speeds over elapsed time, and course. Notice that dead reckoning relies on accurate estimation of speed, elapsed time and direction, which can be measured by using accelerometers and g-sensors built in many mobile devices today. Thus, while the traditional navigational methods of dead reckoning for location acquisition have been replaced by modern positioning technologies, it is still very useful for generating trajectory data, especially when GPS reception is lost, e.g. in a tunnel. Since positions of moving objects calculated by dead reckoning is based on previous positions and estimated distance and directions, the errors in subsequent locations are cumulative. Therefore, the dead reckoning methods only serve as a remedy when the more accurate modern positioning techniques are not applicable.

Given that the time-stamped geographical coordinates can be sampled arbitrarily by a moving object, the next question is whether the moving object needs to report all the sampled trajectory data to the location server for upload to the mobile object database. Obviously the answer is dependent on the application requirements. Since the data acquisition occurs at the moving object, we assume that the location data it possesses have the highest precision. On the contrary, the applications may allow some imprecision based on their requirements. Thus, the data precision at the location server is not expected to be as high as what the moving object has. In summary, the data at the moving object are considered as precise and the required data precision at the location server is determined by the supported LBS applications.

Generally speaking, there are two categories of data reduction techniques reported in the literature of moving object and trajectory management. These approaches aim to reduce the communication and storage overhead of trajectory data representation while not to compromise much precision in the new data representation of trajectory. The basic idea behind data reduction techniques in the first category, called *batched compression techniques*, is to first collect the full set of sampled location data points and then compress the data set by discarding redundant loca-

tion points for transmission to the location server. Because the full data set is taken into consideration by the compression algorithms, the results tend to approaching the global optimal better than the techniques in the other category. These batched compression techniques are very suitable for off-line uploading and analysis of trajectories at various Web 2.0 sites such as Everytrail[4] and Bikely[5]. Take Everytrail as an example, it provides trajectory logging tools on iphone and Android phones for users to record their trips. By uploading a trip trajectory (usually after the trip is completed), a user can annotate the trajectory with pictures and travelogues for sharing with her friends. Thus, the batched compression techniques can be used to reduce the transmission and storage overheads for the user and the hosting server.

For many LBS applications which require timely updates of the moving objects' locations, e.g. fleet management and traffic monitoring applications, the batched compression techniques may not be applied directly. In these applications, the location server needs to know the whereabouts of moving objects constantly. Since continuous location updates of moving objects is infeasible, data reduction of the sample points in a trajectory is usually achieved on-line by selective updates of the locations based on specified precision requirements. Thus, this category of trajectory data reduction techniques is named as *on-line data reduction techniques*. Two ideas are usually exploited in this category of on-line data reduction techniques: (i) use a line segment to fit as many location points in a trajectory as possible; (ii) predict the object movements and report only those location points deviating significantly from the prediction. Techniques based on (i) are able to capture the geometric properties of trajectories pretty well with linear approximation. On the other hand, techniques based on (ii) may capture additional features, such as speed and headings, of object movements and use them in prediction. Based on previously reported location of a moving object, the server is able to predict its next move even though the predicted location may not be exactly accurate. Later in this chapter we discuss the Kalman filter and particle filter, which can both be used for trajectory prediction. By obtaining a prediction model from the server, the moving object applies the same prediction algorithm on the previously reported object locations to figure out where the server perceives as its current location. As a result, by comparing the location perceived by the server and its true location (acquired from its positioning mechanism locally), the moving object is able to decide whether a location update should be reported to the server in order to calibrate the precision of object location and trajectory.

Figure 1.2 illustrates a simple update policy, called *point policy*, that models the tracked object as a jumping point [5]. This policy assumes that the object jumps to a distant point from its current location, stays around the new location for a while, then jumps to another remote location and stays there for a while. The process repeats in the trajectory of the tracked moving object. As shown in the figure, the object moves to location A and sends a location update to the location server. At this point, a circular neighborhood of radius r is set. As long as the object moves within the neighborhood of location A, no update report is sent to the server. When the object

[4] http://www.everytraiil.com

[5] http://www.bikely.com

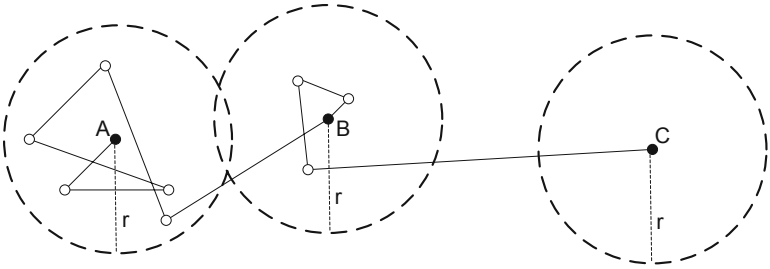


Fig. 1.2 The point update policy. A moving object does not update its new locations as long as they are within the error threshold of the previously reported location.

moves beyond the neighborhood of A to arrive at location B, a new location update is reported to the server. Similarly, a neighborhood of radius r is set at location B. Accordingly, there is no location update issued until the object moves to location C.

In summary, the batched compression techniques rely solely on the moving object to decide which sample points can be discarded. Globally optimized precision in the compressed trajectories may be retained but the required time constraints on reports may be challenging to meet. On the other hand, the on-line reduction techniques require collaboration between the moving object and location server to meet the requirement of updating moving object locations timely while optimizing the precision locally. Notice that the ideas behind the two categories of data reduction techniques can be combined depending on the time and precision requirements.

1.3 Performance Metrics and Error Measures

The primary goal of the trajectory data reduction techniques is to reduce the data size of trajectory representation without compromising much of its precision. Additionally, for the on-line data reduction techniques, the location of an object needs to be reported to the server if the imprecision of the predicted location goes beyond an application-dependent error threshold. Thus, there is a need to find appropriate metrics and error measures for use in algorithms and performance evaluation. The following are the main performance metrics often used to evaluate the efficiency and effectiveness of the trajectory data reduction techniques:

- Processing time: the execution time spent to run a trajectory data reduction algorithm;
- Compression rate: the ratio in the size of an approximate trajectory vs. the size of its original trajectory;
- Error measure: the deviation of an approximate trajectory from its original trajectory.

Among them, the processing time assesses how efficiently a trajectory data reduction technique processing a given trajectory data set. On the other hand, the com-

pression rate and error measure are used to assess the effectiveness of the examined technique. Notice that there may be a tradeoff between these two effectiveness metrics. Thus, trajectory data reduction techniques are usually compared in a plot of these two metrics in order to find the Pareto front.

From the above, we can observe that there is a room to further define different error measures while the definition of compression rate is quite straightforward. For the rest of the section, we discuss two error measures, namely, *perpendicular Euclidean distance* and *time synchronized Euclidean distance*, that are widely used in literature since they have an implication in specifying the imprecision allowed by application and the performance [24, 27, 6].

To specify the allowed imprecision, *distance-based error measure* is a natural choice due to its simplicity and ability to deal with positions of points in multi-dimensional space. Take the error threshold used in on-line data reduction techniques as an example. The distance between a location on the original trajectory acquired from the positioning mechanism and the estimated location on the approximated trajectory intuitively represents how closely the estimated location approximates the original location.[6] With the same reasoning, the aggregated distance between the approximated trajectory and the original trajectory can be used to measure the error introduced by the data reduction process. As mentioned earlier, one of the error measure is to compute the perpendicular Euclidean distances, i.e. the shortest distance, from each of the sampled location points in the original trajectory to the approximated trajectory. As such, we can measure the error by the average or total distances.

Figure 1.3 illustrates the computation of error measure based on the perpendicular Euclidean distance between the original trajectory acquired by a moving object and an approximated trajectory generated by applying one of the trajectory data reduction algorithms. As shown in the figure, the original trajectory is represented by a series of time-stamped location points denoted by $\{p_0, p_1, ..., p_{16}\}$ where p_i is the location of the moving object at time t_i. On the other hand, the approximated trajectory, reduced from the original trajectory, consists of three location points, p_0, p_5 and p_{16}. Notice that the approximated trajectory can also be repre-

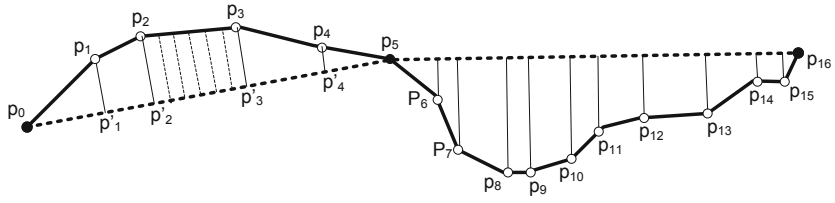


Fig. 1.3 Error measure based on perpendicular Euclidean distance. This error measure takes into account the geometric relationship of the trajectories. However, the temporal factor is not incorporated in this error measure.

[6] Note that here we ignore the inherent noises and imprecision from the location acquisition mechanism and assume the measured original location to be precise.

sented as two line segments $\overline{p_0p_5}$ and $\overline{p_5p_{16}}$. Thus, the approximated trajectory can be interpreted as the path in which the object moved from p_0 to p_5 and then to p_{16}. Based on this interpretation, the sampled location points on the subtrajectories $\{p_0, p_1, ..., p_5\}$ and $\{p_5, p_6, ..., p_{16}\}$ should be projected to the corresponding line segments $\overline{p_0p_5}$ and $\overline{p_5p_{16}}$ for error measurement. Figure 1.3 shows the estimated location points $p'_0, p'_1, ...$ and p'_5 on the line segment $\overline{p_0p_5}$, corresponding to $p_0, p_1, ...$ and p_5, respectively. Notice that the perpendicular Euclidean distance between a location point on the original trajectory to the approximated trajectory is the shortest distance between the point and the approximated trajectory. Thus, the error between the approximated trajectory and the original trajectory can be calculated by summing up the distances of the projection or computing their average distance. Notice that the error measurements by total or average are actually quite sensitive to the number of sampled location points in the original trajectory [24]. A remedy to this deficiency is to take into consideration all possible location points on the original trajectory instead of limiting the error measure to only the sampled location points. This can be achieved by interpolating some *pseudo sampled points* on the original trajectory. For example, five pseudo sampled points between p_2 and p_3 and their projection on the line segment $\overline{p_0p_5}$ (indicated by the five dash lines) are illustrated in Figure 1.3. When an infinite number of pseudo sample points are considered, the area between the original trajectory and the approximated trajectory naturally measures the error between them.

The aforementioned approach elegantly captures the error in the approximated trajectory using perpendicular Euclidean distance. The idea of projecting each possible points in the original trajectory onto the line segments of the approximated trajectory, nevertheless, takes only geometric properties of the trajectories into account. The temporal factor of object movement in the trajectories is not considered in the projection. Notice that a sampled data point $\langle x, y, t \rangle$ in the original trajectory denotes the time t when the moving object are located at x, y. Thus, there is a need to also consider the temporal factor in the projection.

The *time synchronized Euclidian distance* has been proposed as a new error measure for approximated trajectories generated by trajectory data reduction algorithms [24, 27]. The intuition is that the movement projected on the approximated trajectory should be synchronized in terms of "time" with the actual movement on the original trajectory. Consider an original trajectory represented by n sampled location points. It can also be seen as consisting of $n-1$ line segments. Given one of those line segments, even though there is no sampled location points acquired on this line segment, most applications implicitly assume that the object moves in a constant speed along the specific line segment. This interpretation of object moving behavior has been made earlier on the approximated trajectory as well. Since the approximated trajectory is actually a subset of the original trajectory, their location points can be used naturally for time and spatial synchronization of the represented object movement on both trajectories. Consider a line segment on an approximated trajectory and its corresponding subtrajectory on the original trajectory, their end points are the same and thus synchronized. Moreover, the projection of the sampled location points on the original trajectory onto the corresponding line segment on the

approximated trajectory can be determined proportionally by using the time interval spent to move from a location point to a subsequent location point as the weight. As such, the time synchronized location points on the approximated trajectory can be easily determined. Finally, the distance between a location point on the original trajectory and the corresponding time synchronized location points can be derived accordingly.

Figure 1.4 illustrates the idea of time synchronized Euclidean distance. As shown, the location points on the approximated trajectory, i.e. p_0, p_5 and p_{16}, are already synchronized by time. The other sampled location points, e.g. p_1, p_2, p_3 and p_4, are projected to time synchronized location points p'_1, p'_2, p'_3 and p'_4, on the line segment $\overline{p_0 p_5}$. The projection can be computed easily. For example, the coordinates of p'_1, i.e. x_1, y_1, can be derived as follows.

$$x_1 = x_0 + \frac{t_1 - t_0}{t_5 - t_0} \cdot (x_5 - x_0)$$

and

$$y_1 = y_0 + \frac{t_1 - t_0}{t_5 - t_0} \cdot (y_5 - y_0)$$

To eliminate the sensitivity of the time synchronized Euclidean distance to the number of sampled location points, we can interpolate pseudo sampled points on the original trajectory similar to what we discussed earlier regarding the error measure based on the perpendicular Euclidean distance. As shown in Figure 1.4, five pseudo sampled points and their projection to the line segments $\overline{p_0 p_5}$ are shown by dash lines. It is also worth noting that the the lengths of line segments on the approximated trajectory are indicators of the time intervals spent instead of distances moved between sampled location points on the original trajectory. By comparing Figure 1.3 and Figure 1.4, it's easy to observe the difference between these two error measures for approximated trajectories.

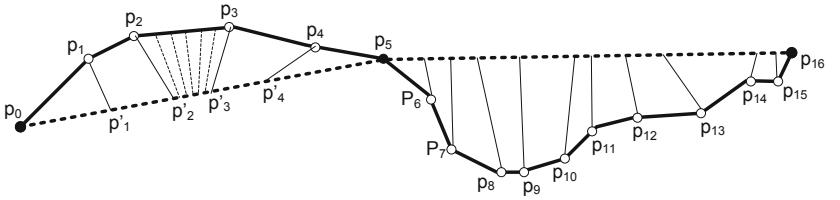


Fig. 1.4 Error measure based on time synchronized Euclidean distance. This error measure takes into account both the geometric relationship and temporal factor of the trajectories.

1.4 Batched Compression Techniques

Given a trajectory that consists of a full series of time-stamped location data points, a batched compression algorithm aims to generate an approximated trajectory by discarding some location points with negligible error from the original trajectory. This is similar to the *line generalization* problem, which has been well studied in the computer graphics and cartography research communities [18, 32, 25, 6]. Works on cartographic line generalization aim to derive small-scale map data from the large-scale high-granularity data. As a result, they can be used to reduce the number of location points in trajectories and thus save storage space.

Some of the line generalization algorithms are very simple in nature. The idea is to retain a fraction of the location points in the original trajectory, without considering the redundancy or other relationships between neighboring data points. For example, the *uniform sampling* algorithm may keep the every i-th location points (e.g. 5th, 10th, 15th, etc) and discard the other points [27]. Since the original trajectory is acquired as a sample of the true trajectory, the new trajectory generated by the uniform sampling process basically is an approximated trajectory with a more coarse granularity. The uniform sampling approach is very efficient computationally, but it may not be useful for certain applications that require better capture of some special trajectory details.

Notice that every location point in the original trajectory may contain different amount of information required to represent the trajectory and that some neighboring location points may contain redundant information, the location points in an approximated trajectory can be selected based on other criteria instead of uniform sampling. A well-known algorithm, called *Douglas-Peucker (DP)*, can be used to approximate the original trajectory [9, 15]. The idea is to replace the original trajectory by an approximate line segment. If the replacement does not meet the specified error requirement, it recursively partitions the original problem into two subproblems by selecting the location point contributing the most errors as the *split point*. This process continues until the error between the approximated trajectory and the original trajectory is below the specified error threshold. The DP algorithm aims to preserve directional trends in the approximated trajectory using the perpendicular Euclidean distance as the error measure.

Figure 1.5 illustrates the first two steps of the Douglas-Peucker algorithm when it is applied on the same trajectory in earlier examples. As shown, in the first step (see Figure 1.5 (a)), the starting point p_0 and end point p_{16} are selected to generate an approximate line segment $\overline{p_0p_{16}}$. The perpendicular Euclidean distance from each sampled location point on the original trajectory to the approximate line segment $\overline{p_0p_{16}}$ is derived. Since some of the perpendicular error distances are greater than the pre-defined error distance threshold, the sampled location point deviating the most from $\overline{p_0p_{16}}$, i.e. p_9 in this example, is chosen as the split point. As a result, in the second step of the algorithm (see Figure 1.5 (b)), a trajectory p_0, p_9, p_{16} is used to approximate the original trajectory. In this step, the original problem is divided into two subproblems where the line segment $\overline{p_0p_9}$ is to approximate the subtrajectory $\{p_0, p_1, ..., p_9\}$ and the line segment $\overline{p_9p_{16}}$ is to approximate the other subtrajectory

$\{p_9, p_{10}, ..., p_{16}\}$. As shown, in the first subproblem, several sampled location points have their perpendicular error distances to $\overline{p_0 p_9}$ greater than the pre-defined error distance threshold. Therefore, p_5, the sampled location point deviating the most from $\overline{p_0 p_9}$, is chosen as the split point and the split subtrajectories are processed recursively until all the sampled location points have perpendicular distances to their approximate line segments within the error threshold. On the other hand, in the second subproblem, the perpendicular distances of all the sample points to the line segment $\overline{p_9 p_{16}}$ are smaller than the error threshold. Therefore, further splitting is not necessary.

The Douglas-Peucker algorithm is widely used in cartographic and computer graphic applications. Several studies have analyzed and evaluated various line generalization algorithms mathematically and perceptually and ranked the Douglas-Peucker algorithm highly [18, 32, 25]. Many cartographers considers the Douglas-Peucker algorithm as one of the most accurate line generalization algorithms available but some think it is too costly in terms of processing time. The time complexity of the original Douglas-Peucker algorithm is $O(N^2)$ where N is the number of trajectory location points. Several improvements have been proposed for implementation of the Douglas-Peucker algorithm and reduce its time complexity to $O(NlogN)$ [15].

As we discussed earlier, the error measure of perpendicular Euclidean distance used in the Douglas-Peucker algorithm only takes into account the geometric aspect of the trajectory representation. Unfortunately, it does not capture the important temporal aspect of the trajectories very well. To address this issue, Meratina and de

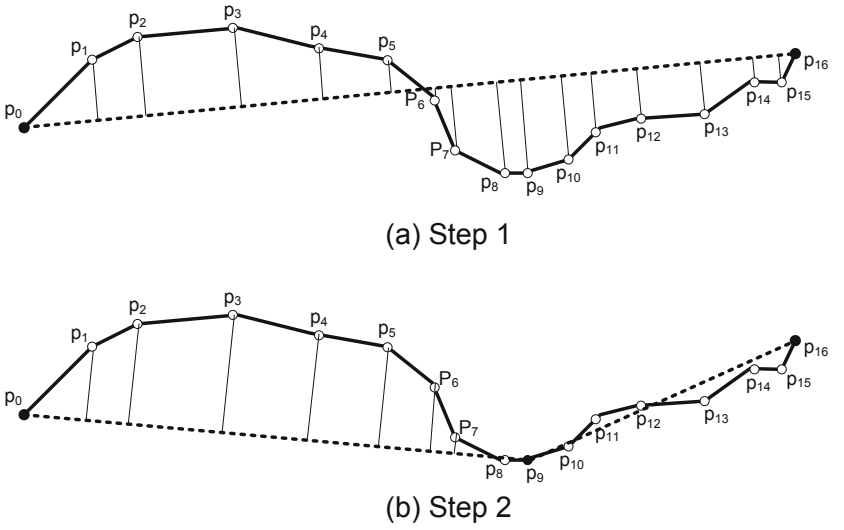


Fig. 1.5 The Douglas-Peucker algorithm. Line segments are used to approximate the original trajectory. The original trajectory is split into two subtrajectories by selecting the location point contributing the most errors as the split point. In Step (1), p_9 is selected as the split point. In Step (2), p_3 is selected as the split point.

By propose to adopt a new error metrics, called *time-distance ratio metric*, to replace the perpendicular Euclidean distance in the Douglas-Peucker algorithm [24].[7] They claim that the improvement is important because this new error measure is not only more accurate but also taking into consideration both geometric and temporal properties of object movements. A modified Douglas-Peucker algorithm, called the *top-down time-ratio (TD-TR)* algorithm, is proposed in [24] because the Douglas-Peucker algorithm decomposes the trajectory approximation problem in a top-down fashion [19].

The Douglas-Peucker algorithm is based on a heuristic that selects the most deviating location points for inclusion in the approximated trajectory in order to lower the introduced error. However, there is no guarantee that the selected split points are the best choices. To ensure that the approximated trajectory is optimal, dynamic programming technique can be employed even thought its computational cost is expected to be high. The Bellman's algorithm [3] applies the dynamic programming technique to approximate a continuous function $g(x)$ by a finite number of line segments. Even though the algorithm considers a one-dimensional value space, it can be generalized to compute an approximated trajectory in the two-dimensional spatial space. The optimization problem is formulated as to minimize the "area" between the original function and the approximate line segments. In this algorithm, including more line segments in the approximated trajectory fits the original trajectory better but is less effective in terms of compression rate. Thus, the Bellman's algorithm can also adopt a penalty to control the tradeoff between compression rate and quality.

Since Bellman's algorithm approximates a continuous function, it can not handle loops, which may occur in trajectory data. Therefore, to employ the Bellman's algorithm for trajectory data reduction, the trajectories with loops need to be segmented first to eliminate loops. Additionally, the original Bellman's algorithm has a time complexity of $O(N^3)$ where N is the number of trajectory location points, which is very expensive when compared to the Douglas-Peucker algorithm. An improved implementation has been proposed to reduce its time complexity to $O(N^2)$ [23].

A natural complement to the top-down Douglas-Peucker algorithm is the *bottom-up* algorithm which, starting from the finest possible approximation of a trajectory, merges line segments in the approximation until some stopping criteria is met. Given a trajectory of N location points, the algorithm first creates $N/2$ line segments, which represent the finest possible approximation of the trajectory. Next, by calculating the cost of merging each pair of adjacent line segments, the algorithm begins to iteratively merge the lowest-cost pair. When a pair of adjacent line segments are merged, the algorithm needs to perform some book-keeping to make sure the cost of merging the new line segment with its right and left neighbors are considered. The algorithm has been used extensively to support a variety of time series data mining tasks and thus can be extended for trajectory data reduction [19, 22, 20, 21].

[7] The time-distance ratio metric is the same as the time synchronized Euclidean distance discussed in Section 1.3.

1.5 On-Line Data Reduction Techniques

The batched compression algorithms, especially the Bellman's algorithm, are expected to produce high-quality approximations due to the access of the whole trajectory. However, they are not as practical as the on-line algorithms in realistic application scenarios. For example, a fleet management application may require trajectory data from tracked moving objects, e.g. trucks, to be reported in a timely fashion back to the fleet control center in order to support multiple continuous queries on truck status in real time. To address the issue of excessive trajectory data continuously generated, there is a demand for on-line trajectory data reduction techniques.

While it's important to reduce the data size of trajectories in order to alleviate storage and communication overheads as well as the computational workload at the location server, there may be certain trajectory properties to be preserved for application needs. Therefore, the on-line trajectory data reduction techniques needs to select some negligible location points intelligently in order to retain a satisfactory approximated trajectory.

One of the essential requirement for on-line processing algorithms is to be able to make efficient on-line decisions when a location point is acquired, i.e. to decide whether to retain the location point in the trajectory or not. The *reservoir sampling* algorithms [30] is well suited for processing trajectory data. The basic idea behind the reservoir sampling algorithms is to maintain a reservoir of size R (or greater than R) which are used to to generate an approximated trajectory of size R. Since the location points in an on-going trajectory are acquired continuously, we do not know in advance the final size of the trajectory. Thus, the key issue is how to select without replacement an approximated trajectory of size R, i.e. once a location point is discarded, there is no way to get it back into the reservoir.

The reservoir algorithm works as follows. It puts the first R location points in the reservoir and decide whether to insert a new location point into the reservoir when it is acquired. Suppose that the k-th location point is acquired (where $k > R$). The algorithm randomly decides, with a probability of R/k, whether this location point should be included as a candidate point in the final approximated trajectory. If the decision is positive, one of the R existing candidates in the reservoir is discarded randomly to make space for the new location point. As such, the algorithm always maintains only R location points in the reservoir, which form a random sample of the original trajectory. Evidently, the reservoir algorithm always maintains a uniform sample of the evolving trajectory without even knowing the eventual trajectory size. Overall, the time complexity is $O(R(1+logN/R))$, where N is the trajectory size.

While the reservoir sampling algorithm is efficient, it does not consider the sequential, spatial and temporal properties of a trajectory. Since all the location points included in the final trajectory are determined randomly and independently, temporal locality and spatial locality in nearby location points are not considered. The *sliding window* algorithm developed for time series data mining can be adapted for trajectory approximation [19, 24]. The idea is to fit the location points in a growing sliding window with a valid line segment and continue to grow the sliding window (and its corresponding line segment) until the approximation error exceeds some

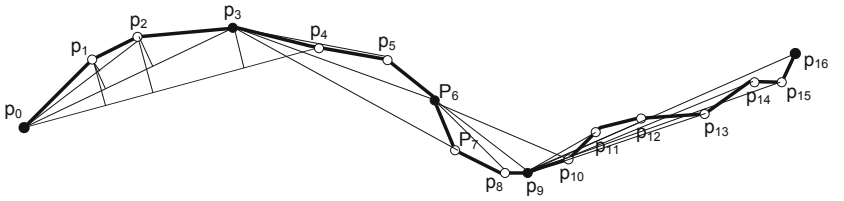


Fig. 1.6 The sliding window algorithm. The idea is to fit the location points in a growing sliding window with a valid line segment and continue to grow the sliding window and its corresponding line segment until the approximation error exceeds some error bound.

error bound. The algorithm first initializes the first location point of a trajectory as the *anchor point* p_a and then starts to grow the sliding window (i.e. by including the next location point in the window). When a new location point p_i is added to the sliding window, the line segment $\overline{p_a p_i}$ is used to fit the subtrajectory consisting of all the location points within the sliding window. As long as the distance errors for all the location points in the sliding window derived against the potential line segment $\overline{p_a p_i}$ are smaller than the user-specified error threshold, the sliding window grows by including the next location point p_{i+1}. Otherwise, the last valid line segment $\overline{p_a p_{i-1}}$ is included as part of the approximated trajectory and p_i is set as the new anchor point. The algorithm continues until all the location points in the original trajectory are visited.

Figure 1.6 illustrates the sliding window algorithm. First, p_0 is set as the anchor point and the initial sliding window is $\{p_0, p_1\}$. Next, p_2 is added into the sliding window. Since $\overline{p_0 p_2}$ fits $\{p_0, p_1, p_2\}$ very well, the sliding window grows into $\{p_0, p_1, p_2, p_3\}$. Again, all the location points within the sliding window do not have error greater than a pre-determined error threshold, i.e. $\overline{p_0 p_3}$ fits $\{p_0, p_1, p_2, p_3\}$ sufficiently well. Thus, the algorithm continues to grow the sliding window into $\{p_0, p_1, p_2, p_3, p_4\}$. This time, the errors for some location points in the sliding window, i.e. p_1, p_2 and p_3, are greater than the error threshold. Thus, the last valid line segment, i.e. $\overline{p_0 p_3}$, is included as a part of the approximated trajectory. Next, the anchor point and the sliding window are reset as p_3 and $\{p_3, p_4\}$, respectively. The algorithm continues to process the rest of the trajectory and then eventually chooses to fit $\{p_3, p_4, p_5, p_6\}$ with $\overline{p_3 p_6}$, $\{p_6, p_7, p_8, p_9\}$ with $\overline{p_6 p_9}$, and $\{p_9, p_{10}, ..., p_{16}\}$ with $\overline{p_9 p_{16}}$. Thus, the final approximated trajectory is $\{p_0, p_3, p_6, p_9, p_{16}\}$.

Meratnia and de By have applied the sliding window algorithm for on-line trajectory data reduction [24]. They consider both of the perpendicular Euclidean distance and time synchronized Euclidean distance as error measures and rename them as *Before Open Window (BOPW)* algorithms, because the location points included in the final approximate trajectory are located before those that result in excessive error. Moreover, Meratnia and de By also apply the heuristic of the Douglas-Peucker algorithm in the open window algorithm. Instead of choosing the location points that result in the longest valid line segments, the new algorithm, called *Normal Opening Window (NOPW)*, chooses location points with the highest error within their sliding

window as the *closing point* of the approximating line segment as well as the new anchor point. As it is in the Douglas-Peucker algorithm, this heuristic works very well in reducing the approximation error. With the new anchor point, the NOPW algorithm continues to process the rest of the trajectory.

Figure 1.7 illustrates the NOPW algorithm. First, p_0 is set as the anchor point and the initial open window is $\{p_0, p_1\}$. Next, p_2 is added into the open window. Similar to the illustration in Figure 1.6, $\overline{p_0p_2}$ and $\overline{p_0p_3}$ respectively fits $\{p_0, p_1, p_2\}$ and $\{p_0, p_1, p_2, p_3\}$ sufficiently well, because all the location points within these windows do not have error greater than a pre-determined error threshold. When the opening window grows into $\{p_0, p_1, p_2, p_3, p_4\}$, the errors for p_1, p_2 and p_3 are greater than the error threshold. Instead of choosing p_3 as the closing point, the NOPW algorithm chooses p_2 as the closing point to include the line segment $\overline{p_0p_2}$ as a part of the approximated trajectory. Then, the anchor point and the opening window are reset as p_2 and $\{p_2, p_3, p_4, p_5\}$, respectively. The algorithm continues to process the rest of the trajectory and then eventually chooses $\{p_2, p_2, p_5, p_8, p_{16}\}$ as the approximated trajectory.

1.6 Trajectory Data Reduction Based on Speed and Direction

The data reduction techniques described earlier all use a subset of the location points in the original trajectory as an approximation. In these algorithms, the approximation error, measured by variants of Euclidean distances such as perpendicular Euclidean distance or time synchronized Euclidean distance, are used to select data points that represents the original trajectory as close as possible. In [27], Potamias et. al argue that a data point should be included in the approximated trajectory as long as it reveals changes in the course of a trajectory. As long as the location of an incoming data point can be predicted (e.g. by interpolation or dead reckoning) from the previous movement, this data point can be safely discarded without significant loss in accuracy since it contributes little information. They also argue that, in addition to spatial positions, changes in *speed* and *direction* are key factors for

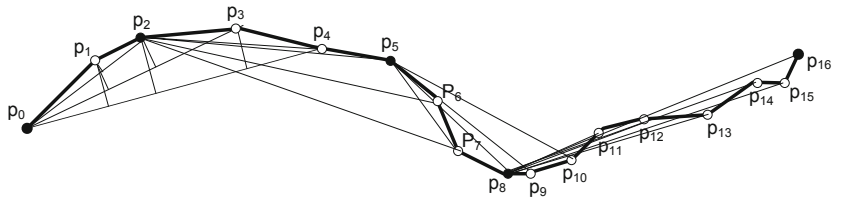


Fig. 1.7 The open window algorithm. The idea is similar to the sliding window algorithm but it applies the heuristic of the Douglas-Peucker algorithm to choose location points with the highest error within their sliding window as the closing point of the approximating line segment as well as the new anchor point.

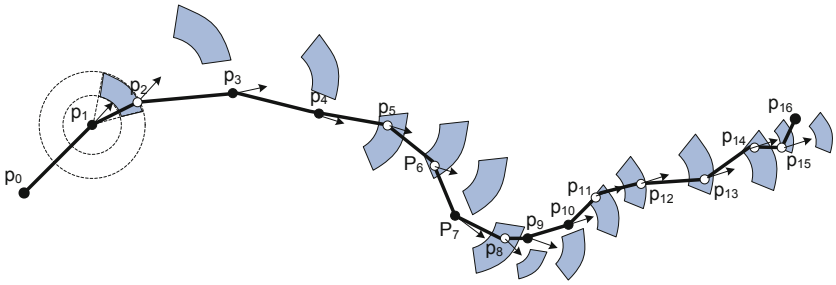


Fig. 1.8 The construction of the safe area and the data reduction strategy in a threshold-guided sampling algorithm. Only the location points fallen out of the projected safe areas are reported to the location server.

predicting locations in a trajectory.[8] Therefore, a decision to include a particular location point or not needs to take into account changes in speed and direction.

In many real life scenario, e.g. driving on a highway, moving objects usually do not make dramatic speed and direction changes. In other words, a moving object may likely to move in the same speed and direction (with some minor changes) for some time. Thus, the current location of a moving object can usually be predicted with little cost by using the speed, direction and time from the last observed location(s).

Based on the specified speed and direction tolerance thresholds, Potamias et. al propose *threshold-guided sampling* algorithms to reduce redundant data points in trajectories [27]. The basic idea is to use a *safe area* derived from the last two locations and a given thresholds to efficiently determine whether a newly acquired location point contains important information. If the new data point, as predicted, is located within the safe area, then this location point is considered as redundant and thus can be discarded. On the other hand, if the location point is fallen outside the safe region, it is included in the approximate trajectory since considerable movement change has happened.

A key issue in the threshold-guided sampling algorithms is the construction of the safe area, which is derived based on the last known location point of the trajectory. Using the last observed speed and the speed tolerance threshold, a circular area representing all possible points where the moving object may possibly locate, if it maintains the anticipated speed range, can be derived. On the other hand, the direction and the direction deviation threshold can be used to determine a partial plane that captures all possible directions the object may move towards. The safe area is then obtained by intersection of the above-described two areas.

Figure 1.8 illustrates the construction of the safe area and the data reduction strategy in a threshold-guided sampling algorithm on our running example. We assume that p_0 and p_1 are included in the approximated trajectory and that the speed and direction of the moving object at p_1 are known. Upon acquisition of the location point

[8] To maintain consistency and integrity of discussions in the chapter, here we use the term "direction" to refer to "heading" considered in [27].

p_2, the safe area based on the speed/direction at p_1, the specified speed/direction thresholds, and the time interval between p_2 and p_1, a fan-shape safe area for p_2 is derived. Since p_2 has fallen in its safe area as predicted, the information carried by p_2 is not considered as important. Thus, p_2 is discarded. Next, upon acquisition of the location point p_3, the safe area for p_3 is derived. This time, p_3 has fallen outside the safe area and thus been considered as an important location point for the trajectory. The same decision process has been performed on all the rest of the trajectory location points. The final approximated trajectory in this example is $\{p_0, p_1, p_3, p_4, p_7, p_9, p_{10}, p_{16}\}$.

Observed that the decisions made based on the above-described safe areas are vulnerable to the problem of error propagation, which also exists in dead reckoning, Potamias et. al consider an alternative scheme of constructing a safe area. Instead of constructing the safe area using the last two points included in the approximated trajectory to derive the speed and direction, the new scheme derive the speed and direction from the last two actual location points acquired. Since this scheme is also susceptible to error propagation, when the object movement exhibits a smooth but significant change in the object's direction, Potamias et. al further adopt the joint safe area intersect by the two safe area schemes described earlier.

While the threshold-guided sampling algorithms may achieve significant trajectory data reduction, they may not be effective under the constraint of limited memory. Therefore, Potamias et. al propose another on-line sampling algorithm, called *STTrace*, to obtain an approximated trajectory under a given memory of known and constant size [27]. The idea is to insert data points into the sample memory based on the movement features (e.g., speed and direction) as in the aforementioned threshold-guided sampling algorithms. However, once the memory used to maintain the approximated trajectory is full, we need to decide whether to evict an existing data point (and which one) in order to accommodate a new data point. To address this issue, SSTrace adopts a deletion scheme based on time-synchronous Euclidean distance to discard a data point that results in the least distortion to the maintained approximated trajectory.

In addition to [27], Meratnia and de By also exploit the speed information hidden in the trajectories [24]. By analyzing the derived speeds at subsequent segments of a trajectory, they propose to use the speed difference of two subsequent segments as a criteria to decide whether the location point between the two segments should be retained in the approximated trajectory. Accordingly, a new class of spatio-temporal algorithms are obtained by integrating the speed difference threshold and the time synchronized Euclidean distance with the top-down algorithms and open window algorithms.

Finally, Hung and Peng propose a model-driven data acquisition technique that reports the speeds of a moving object [17]. They develop a kernel regression algorithm and derive a set of kernel functions to model a time series of speeds readings.

1.7 Trajectory Filtering

Spatial trajectories are never perfectly accurate, due to sensor noise and other factors. Sometimes the error is acceptable, such as when using GPS to identify which city a person is in. In other situations, we can apply various filtering techniques to the trajectory to smooth the noise and potentially decrease the error in the measurements. This section explains and demonstrates some conventional filtering techniques using sample data.

It is important to note that filtering is not always necessary. In fact, we rarely use it for GPS data. Filtering is important in those situations where the trajectory data is particularly noisy, or when one wants to derive other quantities from it, like speed or direction.

1.7.1 Sample Data

To demonstrate some of the filtering techniques in this chapter, we recorded a trajectory with a GPS logger, shown in Figure 1.9. The GPS logger recorded 1075 points at a rate of one per second during a short walk around the Microsoft campus in Redmond, Washington USA. For plotting, we converted the latitude/longitude points to (x,y) in meters. While the walk itself followed a casual, smooth path, the recorded trajectory shows many small spikes due to measurement noise. In addition, we manually added some outliers to simulate large deviations that sometimes appear in recorded trajectories. These outliers are marked in Figure 1.9. We will use this data to demonstrate the effects of the filtering techniques we describe below.

1.7.2 Trajectory Model

The actual, unknown trajectory is denoted as a sequence of coordinates $\mathbf{x}_i = (x_i, y_i)^T$. The index i represents time increments, with $i = 1...N$. The boldface $\mathbf{x}_i$ is a two-element vector representing the x and y coordinates of the trajectory coordinate at time i.

Due to sensor noise, measurements are not exact. This error is usually modeled by adding unknown, random Gaussian noise to the actual trajectory points to give the known, measured trajectory, whose coordinates are given as vectors $\mathbf{z}_i$ as

$$\mathbf{z}_i = \mathbf{x}_i + \mathbf{v}_i \tag{1.1}$$

The noise vector $\mathbf{v}_i$ is assumed to be drawn from a two-dimensional Gaussian probability density with zero mean and diagonal covariance matrix R, i.e.

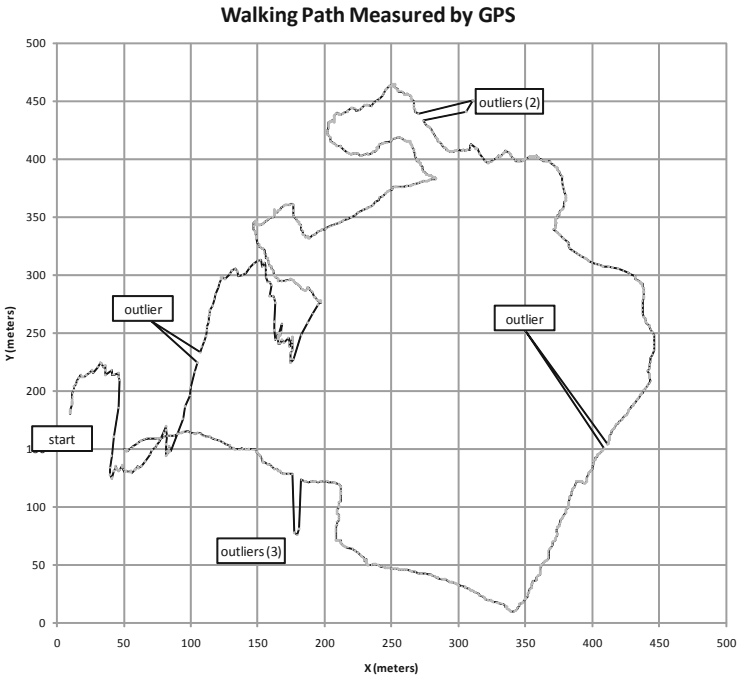


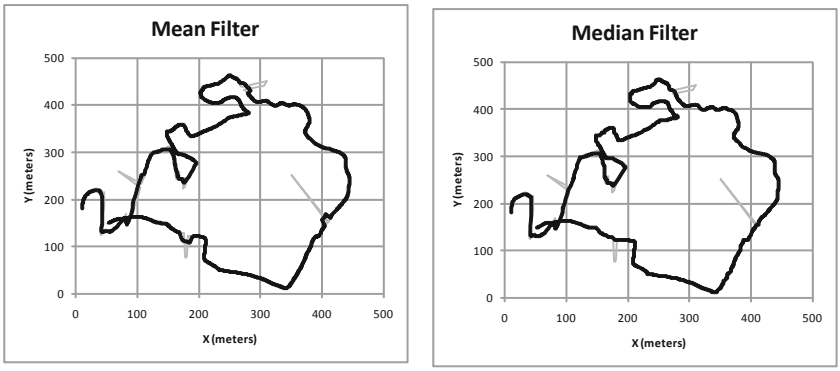
Fig. 1.9 This is a trajectory recorded by a GPS logger. The outliers were inserted later for demonstration.

$$\mathbf{v}_i \sim N(\mathbf{0}, R) \qquad R = \begin{bmatrix} \sigma^2 & 0 \\ 0 & \sigma^2 \end{bmatrix} \tag{1.2}$$

With the diagonal covariance matrix, this is the same as adding random noise from two different, one-dimensional Gaussian densities to x_i and y_i separately, each with zero mean and standard deviation σ. It is important to note that Equation (1.1) is just a model for noise from a location sensor. It is not an algorithm, but an approximation of how the measured sensor values differ from the true ones. For GPS, the Gaussian noise model above is a reasonable one [7]. In our experiments, we have observed a standard deviation σ of about four meters.

1.8 Mean and Median Filters

One simple way to smooth noise is to apply a mean filter. For a measured point $\mathbf{z}_i$, the estimate of the (unknown) true value is the mean of $\mathbf{z}_i$ and its $n-1$ predecessors in time. The mean filter can be thought of as a sliding window covering n temporally adjacent values of $\mathbf{z}_i$. In equation form, the mean filter is



(a) Results of mean filter (b) Results of median filter

Fig. 1.10 The dark curve show the result of the mean filter in (a) and the median filter in (b). One advantage of the median filter is that it is less affected by outliers. The gray curve shows the original measured trajectory.

$$\hat{\mathbf{x}}_i = \frac{1}{n} \sum_{j=i-n+1}^{i} \mathbf{z}_j \tag{1.3}$$

This equation introduces another notational convention: $\hat{\mathbf{x}}_i$ is the estimate of $\mathbf{x}_i$.

Figure 1.10(a) shows the result of the mean filter with $n = 10$. The resulting curve is smoother.

The mean filter as given in Equation (1.3) is a so-called "causal" filter, because it only depends on values in the past to compute the estimate $\hat{\mathbf{x}}_i$. In fact, all the filters discussed in this chapter are causal, meaning they can be sensibly applied to real time data as it arrives. For post-processing, one could use a non-causal mean filter whose sliding window takes into account both past and future values to compute $\hat{\mathbf{x}}_i$.

One disadvantage of the mean filter is that it introduces lag. If the true underlying value $\mathbf{x}_i$ changes suddenly, the estimate from the mean filter will respond only gradually. So while a larger sliding window (larger value of n) makes the estimates smoother, the estimates will also tend to lag changes in $\mathbf{x}_i$. One way to mitigate this problem is to use a weighted mean, where more recent values of $\mathbf{z}_i$ are given more weight.

Another disadvantage of the mean filter is its sensitivity to outliers. From Figure 1.10(a), it is clear that the artificially introduced outliers noticeably pull away the estimated curve from the data. In fact, it is possible to find an outlier value to pull the mean to any value we like.

One way to mitigate the outlier problem is to use a median filter rather than a mean filter. The median filter simply replaces the mean filter's mean with a median. The equation for the median filter that corresponds to the mean filter in Equation (1.3) is

$$\hat{\mathbf{x}}_i = median\{\mathbf{z}_{i-n+1}, \mathbf{z}_{i-n+2}, \ldots, \mathbf{z}_{i-1}, \mathbf{z}_i\} \tag{1.4}$$

Figure 1.10(b) shows the result of the median filter, where it is clear that it is less sensitive to outliers and still gives a smooth result.

The mean and median filters are both simple and effective at smoothing a trajectory. They both suffer from lag. More importantly, they are not designed to help estimate higher order variables like speed. In the next two sections, we discuss the Kalman filter and the particle filter, two more advanced techniques that reduce lag and can be designed to estimate more than just location.

1.9 Kalman Filter

The mean and median filters use no model of the trajectory. More sophisticated filters, like the Kalman and particle filters, model both the measurement noise (as given by Equation (1.1)) and the dynamics of the trajectory.

For the Kalman filter, a simple example is smoothing trajectory measurements from something arcing through the air affected only by gravity, such as a soccer ball. While measurements of the ball's location, perhaps from a camera, are noisy, we can also impose constraints on the ball's trajectory from simple laws of physics. The trajectory estimate from the Kalman filter is a tradeoff between the measurements and the motion model. Besides giving estimates that obey the laws of physics, the Kalman filter gives principled estimates of higher order motion states like speed.

The subsections below develop the model for the Kalman filter for the example trajectory from above. We use notation from the book by Gelb, which is one of the standard references for Kalman filtering [1].

1.9.1 Measurement Model

While the mean and median filters can only estimate what is directly measured, the Kalman filter can estimate other variables like speed and acceleration. In order to do this, the Kalman formulation makes a distinction between what is measured and what is estimated, as well as formulating a linear relationship between the two.

As above, we assume that the measurements of the trajectory are taken as noisy values of x and y:

$$\mathbf{z}_i = \begin{pmatrix} z_i^{(x)} \\ z_i^{(y)} \end{pmatrix} \tag{1.5}$$

Here $z_i^{(x)}$ and $z_i^{(y)}$ are noisy measurements of the x and y coordinates.

The Kalman filter gives estimates for the state vector, which describes the full state of the object being tracked. In our case, the state vector will include both the object's location and velocity:

$$\mathbf{x}_i = \begin{pmatrix} x_i \\ y_i \\ s_i^{(x)} \\ s_i^{(y)} \end{pmatrix} \tag{1.6}$$

The elements x_i and y_i are the true, unknown coordinates at time i, and $s_i^{(x)}$ and $s_i^{(y)}$ are the x and y components of the true, unknown velocity at time i. The Kalman filter will produce an estimate of $\mathbf{x}_i$, which includes velocity, even though this is not directly measured. The relationship between the measurement vector $\mathbf{z}_i$ and the state vector $\mathbf{x}_i$ is

$$\mathbf{z}_i = H_i \mathbf{x}_i + \mathbf{v}_i \tag{1.7}$$

where H_i, the measurement matrix, translates between $\mathbf{x}_i$ and $\mathbf{z}_i$. For our example, H_i expresses the fact that we are measuring x_i and y_i to get $z_i^{(x)}$ and $z_i^{(y)}$, but we are not measuring velocity. Thus,

$$H_i = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix} \tag{1.8}$$

H_i also neatly accounts for the dimensionality difference between $\mathbf{x}_i$ and $\mathbf{z}_i$. While the subscript on H_i means it could change with time, it does not in our example.

The noise vector $\mathbf{v}_i$ in Equation (1.7) is the same as the zero-mean, Gaussian noise vector in Equation (1.2). Thus Equation (1.7) is how the Kalman filter models measurement noise. In fact, Gaussian noise has been proposed as a simple model of GPS noise [7], and for our example it would be reasonable to set the measurement noise σ to a few meters.

1.9.2 Dynamic Model

If the first half of the Kalman filter model is measurement, the second half is dynamics. The dynamic model approximates how the state vector $\mathbf{x}_i$ changes with time. Like the measurement model, it uses a matrix and added noise:

$$\mathbf{x}_i = \Phi_{i-1} \mathbf{x}_{i-1} + \mathbf{w}_{i-1} \tag{1.9}$$

This gives $\mathbf{x}_i$ as a function of its previous value $\mathbf{x}_{i-1}$. The system matrix Φ_{i-1} gives the linear relationship between the two. For the example problem, we have

$$\Phi_{i-1} = \begin{bmatrix} 1 & 0 & \Delta t_i & 0 \\ 0 & 1 & 0 & \Delta t_i \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.10}$$

Here Δt_i is the elapsed time between the state at time i and time $i-1$. Recalling the state vector from Equation (1.6), the top two rows of the system matrix say that $x_i = x_{i-1} + \Delta t_i s_i^{(x)}$ and similarly for y_i. This is standard physics for a particle with constant velocity.

The bottom two rows of system matrix say $s_i^{(x)} = s_{i-1}^{(x)}$ and $s_i^{(y)} = s_{i-1}^{(y)}$, which means the velocity does not change. Of course, we know this is not true, or else the trajectory would be straight with no turns. The dynamic model accounts for its own inaccuracy with the noise term $\mathbf{w}_{i-1}$. This is another zero-mean Gaussian noise term. For our example, we have

$$\mathbf{w}_i \sim N(\mathbf{0}, Q_i) \qquad Q_i = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & \sigma_s^2 & 0 \\ 0 & 0 & 0 & \sigma_s^2 \end{bmatrix} \tag{1.11}$$

With the first two rows of zeros, this says that the relationship between location and velocity (e.g. $x_i = x_{i-1} + \Delta t_i s_i^{(x)}$) is exact. However, the last two rows say that the assumption in the system matrix about constant velocity is not quite true, but that the velocity is noisy, i.e. $s_i^{(x)} = s_i^{(x)} + N(0, \sigma_s^2)$. This is how the Kalman filter maintains its assumption about the linear relationship between the state vectors over time, yet manages to account for the fact that the dynamic model does not account for everything.

1.9.3 Entire Kalman Filter Model

The Kalman filter requires a measurement model and dynamic model, both discussed above. It also requires assumptions about the initial state and uncertainty of the initial state. Here are the all the required elements:

H_i – measurement matrix giving measurement $\mathbf{z}_i$ from state $\mathbf{x}_i$, Equation (1.8).
R_i – measurement noise covariance matrix, Equation (1.2).
Φ_{i-1} – system matrix giving state $\mathbf{x}_i$ from $\mathbf{x}_{i-1}$, Equation (1.10).
Q_i – system noise covariance matrix, Equation (1.11).
$\hat{\mathbf{x}}_0$ – initial state estimate.
P_0 – initial estimate of state error covariance.

The initial state estimate can usually be estimated from the first measurement. For our example, the initial position came from $\mathbf{z}_0$, and the initial velocity was taken as zero. For P_0, a reasonable estimate for this example is

$$p_0 = \begin{bmatrix} \sigma^2 & 0 & 0 & 0 \\ 0 & \sigma^2 & 0 & 0 \\ 0 & 0 & \sigma_s^2 & 0 \\ 0 & 0 & 0 & \sigma_s^2 \end{bmatrix} \tag{1.12}$$

The value of σ is an estimate of the sensor noise for GPS. For our example, we set $\sigma = 4$ meters based on earlier experiments with our particular GPS logger. We set $\sigma_s = 6.62$ meters/second , which we computed by looking at the changes in velocity estimated naively from the measurement data.

1.9.4 Kalman Filter

For the derivation of the Kalman filter, see [1]. The result is a two-step algorithm that first extrapolates the current state to the next state using the dynamic model. In equations, this is

$$\hat{\mathbf{x}}_i^{(-)} = \Phi_{i-1}\hat{\mathbf{x}}_{i-1}^{(+)} \tag{1.13}$$

$$P_i^{(-)} = \Phi_{i-1}P_{i-1}^{(+)}\Phi_{i-1}^T + Q_{i-1} \tag{1.14}$$

The terms in Equation (1.13) should be familiar. The $*^{(-)}$ superscript refers to the extrapolated estimate of the state vector, and the $*^{(+)}$ superscript refers to the estimated value of the state vector. Equation (1.14) is interesting in that it concerns an extrapolation P_i of the covariance of the state vector, giving some idea of the error associated with the state vector.

The first step of the Kalman filter is pure extrapolation, with no use of measurements. The second step incorporates the current measurement to make new estimates. The equations are

$$K_i = P_i^{(-)}H_i^T(H_iP_i^{(-)}H_i^T + R_i)^{-1} \tag{1.15}$$

$$\hat{\mathbf{x}}_\mathbf{i}^{(+)} = \hat{\mathbf{x}}_i^{(-)} + K_i(\mathbf{z}_i - H_i\hat{\mathbf{x}}_i^{(-)}) \tag{1.16}$$

$$P_i^{(+)} = (I - K_iH_i)P_i^{(-)} \tag{1.17}$$

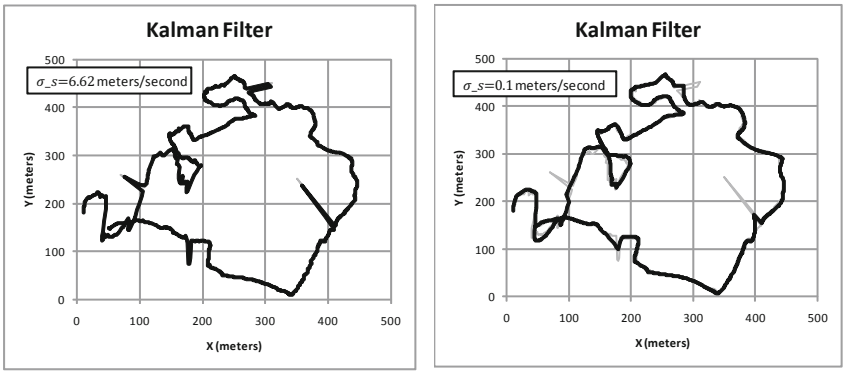
Equation (1.15) gives the Kalman gain matrix K_i. It is used in Equation (1.16) to give the state estimate $\hat{\mathbf{x}}_i^{(+)}$ and in Equation (1.17) to give the state covariance estimate $P_i^{(+)}$.

Applying these equations to the example trajectory gives the plot in Figure 1.11(a).

1.9.5 Kalman Filter Discussion

One of the advantages of the Kalman filter over the mean and median filters is its lack of lag. There is still some intrinsic lag, because the Kalman filter depends on previous measurements for its current estimate, but it also includes a dynamic model

to keep it more current. Another advantage is the richer state vector. In our example, it includes velocity as well as location, making the Kalman filter a principled way to estimate velocity based on a sequence of location measurements. It is easy to add acceleration as another part of the state vector. Yet another advantage is the Kalman filter's estimate of uncertainty in the form of the covariance matrix $P_i^{(+)}$ given in Equation 1.17. Knowledge of the uncertainty of the estimate can be used by a higher-level algorithm to react intelligently to ambiguity, possibly by invoking another sensor or asking a user for help.



(a) Kalman filter with $\sigma_s = 6.62$ meters/second (b) Kalman filter with $\sigma_s = 0.1$ meters/second

Fig. 1.11 The dark curve show the result of the Kalman filter. In (a), the process noise σ_s comes from an estimate on the original noisy data. The process noise in (b) is much smaller, leading to a smoother filtered trajectory.

One of the mysteries of the Kalman filter is the process noise, which is embodied as σ_s in our example. In our example, this represents how much the tracked object's velocity changes between time increments. In reality, this is difficult to estimate in many cases, including our example trajectory of a pedestrian. A larger value of σ_s represents less faith in the dynamic model relative to the measurements. A smaller value puts more weight on the dynamic model, often leading to a smoother trajectory. This is illustrated in Figure 1.11(b) where we have reduced the value of σ_s from our original value of 6.62 meters/second to 0.1 meters/second. The resulting trajectory is indeed smoother and less distracted by outliers.

One of the main limitations of the Kalman filter is the requirement that the dynamic model be linear, i.e. that the relationship between $\mathbf{x}_{i-1}$ and $\mathbf{x}_i$ be expressed as a matrix multiplication (plus noise). Sometimes this can be solved with an extended Kalman filter, which linearizes the problem around the current value of the state. But this can be difficult for certain processes, like a bouncing ball or an object constrained by predefined paths. In addition, all the variables in the Kalman filter model are continuous, without a convenient way to represent discrete variables like the mode of transportation or goal. Fortunately, the particle filter fixes these problems, and we discuss it next.

1.10 Particle Filter

The particle filter is similar to the Kalman filter in that they both use a measurement model and a dynamic model. The Kalman filter gains efficiency by assuming linear models (matrix multiplication) plus Gaussian noise. The particle filter relaxes these assumptions for a more general, albeit generally less efficient, algorithm. But, as shown by Hightower and Borriello, particle filters are practical for tracking even on mobile devices [16].

The particle filter gets its name from the fact that it maintains a set of ”particles” that each represent a state estimate. There is a new set of particles generated each time a new measurement becomes available. There are normally hundreds or thousands of particles in the set. Taken together, they represent the probability distribution of possible states. A good introduction to particle filtering is the chapter by Doucet et al. [8], and this section uses their notation.

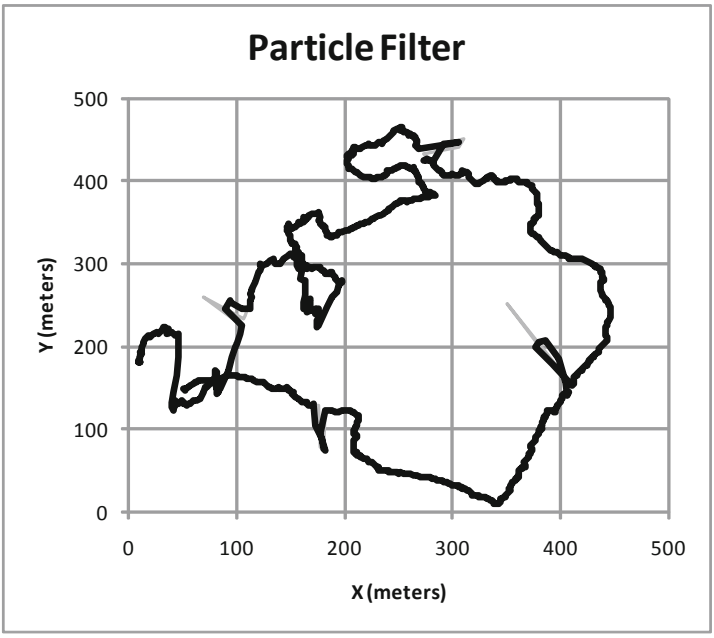


Fig. 1.12 This is the result of the particle filter. It is similar to the result of the Kalman filter in Figure 3(a) since it uses the same measurement model, dynamic model, and noise assumptions.

As in the previous section on Kalman filtering, this section shows how to apply particle filtering to our example tracking problem.

1.10.1 Particle Filter Formulation

As with the Kalman filter discussed above, the particle filter makes estimates $\hat{\mathbf{x}}_i$ of a sequence of unknown state vectors $\mathbf{x}_i$, based on measurements $\mathbf{z}_i$. For our example, these vectors are formulated just as in the Kalman filter. As a reminder, the state vector $\mathbf{x}_i$ has four scalar elements representing location and velocity. The measurement vector $\mathbf{z}_i$ has two elements representing a location measurement with some degree of inaccuracy.

The particle filter's measurement model is a probability distribution $p(\mathbf{z}_i|\mathbf{x}_i)$ giving the probability of seeing a measurement given a state vector. This distribution must be provided to the particle filter. This is essentially a model of a noisy sensor which might produce many different possible measurements for a given state. It is much more general than the Kalman filter's measurement model, which is limited to $\mathbf{z}_i = H_i\mathbf{x}_i + \mathbf{v}_i$, i.e. a linear function of the state plus added Gaussian noise.

To stay consistent with the example, however, we will use the same measurement model as in the Kalman filter, writing it as

$$p(\mathbf{z}_i|\mathbf{x}_i) = N((x_i, y_i)^T, R_i) \tag{1.18}$$

This says that the measurement is a Gaussian distributed around the actual location with a covariance matrix R_i from Equation (1.2). The measurement ignores the velocity components in $\mathbf{x}_i$.

While this is the same measurement model that we used for the Kalman filter, it could be much more expressive. For instance, it might be the case that the location sensor's accuracy varies with location, such as GPS in an urban canyon. The particle filter's model could accommodate this.

In addition to the measurement model, the other part of the particle filter formulation is the dynamic model, again paralleling the Kalman filter. The dynamic model is also a probability distribution, which simply gives the distribution over the current state $\mathbf{x}_i$ given the previous state $\mathbf{x}_{i-1}$: $p(\mathbf{x}_i|\mathbf{x}_{i-1})$. The analogous part of the Kalman filter model is $\mathbf{x}_i = \Phi_{i-1}\mathbf{x}_{i-1} + \mathbf{w}_{i-1}$ (Equation (1.9)). The particle filter version is much more general. For instance, it could model the fact that vehicles often slow down when climbing hills and speed up going down hills. It can also take into account road networks or paths through a building to constrain where a trajectory can go. This feature of the particle filter has proved useful in many tracking applications.

It is not necessary to write out the dynamic model $p(\mathbf{x}_i|\mathbf{x}_{i-1})$. Instead, it is sufficient to sample from it. That is, given a value of $\mathbf{x}_{i-1}$, we must be able to create samples of $\mathbf{x}_i$ that adhere to $p(\mathbf{x}_i|\mathbf{x}_{i-1})$. For our example, we will use the same dynamic model as the Kalman filter, which says that location changes deterministically as a function of the velocity and that velocity is randomly perturbed with Gaussian noise:

$$\begin{aligned} x_{i+1} &= x_i + v_i^{(x)} \Delta t_i \\ y_{i+1} &= y_i + v_i^{(y)} \Delta t_i \\ v_{i+1}^{(x)} &= v_i^{(x)} + w_i^{(x)} \qquad w_i^{(x)} \sim N(0, \sigma_s^2) \\ v_{i+1}^{(y)} &= v_i^{(y)} + w_i^{(y)} \qquad w_i^{(y)} \sim N(0, \sigma_s^2) \end{aligned} \tag{1.19}$$

The above is a recipe for generating random samples of $\mathbf{x}_{i+1}$ from $\mathbf{x}_i$. Contrary to the Kalman filter, the particle filter requires actually generating random numbers, which in this case serve to change the velocity.

Finally, also paralleling the Kalman filter, we need an initial distribution of the state vector. For our example, we can say the initial velocity is zero and the initial location is a Gaussian around the first measurement with a covariance matrix R_i from Equation (1.2).

1.10.2 Particle Filter

The particle filter maintains a set of P state vectors, called particles: $\mathbf{x}_i^{(j)}, j = 1...P$. There are several versions of the particle filter, but we will present the Bootstrap Filter from Gordon [12]. The initialization step is to generate P particles from the initial distribution. For our example, these particles would have zero velocity and be clustered around the initial location measurement with a Gaussian distribution as explained above. This is the first instance of how the particle filter requires actually generating random hypotheses about the state vector. This is different from the Kalman filter which generates state estimates and uncertainties directly. We will call these particles $\mathbf{x}_0^{(j)}$.

With a set of particles and $i > 0$, the first step is "importance sampling," which uses the dynamic model $p(\mathbf{x}_i|\mathbf{x}_{i-1})$ to probabilistically simulate how the particles change over one time step. This is analogous to the extrapolation step in the Kalman filter in that it proceeds without regard to the measurement. For our example, this means invoking Equation (1.19) to create $\tilde{\mathbf{x}}_i$. The tilde ($\sim$) indicates extrapolated values. Note that this involves actually generating random numbers for the velocity update.

The next step computes"importance weights" for all the particles using the measurement model. The importance weights are

$$\tilde{w}_i^{(j)} = p(\mathbf{z}_i|\tilde{\mathbf{x}}_i^{(j)}) \tag{1.20}$$

Larger importance weights correspond to particles that are better supported by the measurement. The important weights are then normalized so they sum to one.

The last step in the loop is the "selection step" when a new set of P particles $\mathbf{x}_i^{(j)}$ is selected at random from the $\tilde{\mathbf{x}}_i^{(j)}$ based on the normalized importance weights. The probability of selecting an $\tilde{\mathbf{x}}_i^{(j)}$ for the new set of particles is proportional to its importance weight $\tilde{w}_i^{(j)}$. It is not unusual to select the same $\tilde{\mathbf{x}}_i^{(j)}$ more than once if it

has a larger importance weight. This is the last step in the loop, and processing then returns to the importance sampling step with the new set of particles.

While the particles give a distribution of state estimates, one can compute a single estimate with a weighted sum:

$$\tilde{\mathbf{x}}_i = \sum_{j=1}^{P} \tilde{w}_i^{(j)} \tilde{\mathbf{x}}_i^{(j)} \tag{1.21}$$

Applying the particle filter to our example problem gives the result in Figure 1.12. We used $P = 1000$ particles. This result looks similar to the Kalman filter result, since we used the same measurement model, dynamic model, and noise in both.

1.10.3 Particle Filter Discussion

One potential disadvantage of the particle filter is computation time, which is affected by the number of particles. More particles generally give a better result, but at the expense of computation. Fox gives a method to choose the number of particles based on bounding the approximation error [10].

Even though the particle filter result looks similar to the Kalman filter result in our example, it is important to understand that the particle filter has the potential to be much richer. As mentioned previously, it could be made sensitive to a network of roads or walking paths. It could include a discrete state variable representing the mode of transportation, e.g. walking, bicycling, in a car, or on a bus.

While it is tempting to add many variables to the state vector, the cost is often more particles required to make a good state estimate. One solution to this problem is the Rao-Blackwellized particle filter [26]. It uses a more conventional filter, like Kalman, to track some of the state variables and a particle filter for the others.

1.11 Summary

In this chapter, we discussed two low-level preprocessing tasks for spatial trajectory computing and data management: 1) how to reduce the data size for representing a trajectory; and 2) how to filter spatial trajectories to reduce measurement noise and to estimate higher level properties of a trajectory. For task 1, the data reduction techniques can run in a batch mode after the data is collected or in an on-line mode as the data is collected. Due to the inherent spatio-temporal characteristics in spatial trajectories, conventional error measure, e.g. the perpendicular Euclidean distance that has been widely used in many line generalization algorithms, does not work well in determining the location points to be included in the approximated trajectory. On the other hand, the time synchronized Euclidean distance, providing a more precise error measurement for approximated trajectories, has been incorporated into sev-

eral trajectory data reduction techniques recently. Moreover, research on trajectory data reduction techniques have been extended from focusing on location information to high-level properties of trajectories such as speed and directions. For task 2, trajectory filtering techniques are employed to reduce measurement noise. Additionally, they can be used to estimate high-level properties of a trajectory like its speed and direction and thus can possibly be integrated with trajectory data reduction techniques. Trajectory filtering methods, including mean and median filtering, the Kalman filter, and the particle filter, are important techniques for trajectory data preprocessing.

Many transportational and recreational activities have left very useful information in form of trajectories. In recent years, researchers have started to explore the semantics, e.g. activity types and transportation modes, behind various trajectories and thus proposed the notion of *semantic trajectories* [31, 2, 34, 11]. Accordingly, semantic compression techniques, while in its infancy, have been proposed [29, 4]. We anticipate more advanced data reduction and filtering techniques for semantic trajectories to be developed in the coming years as we obtain more in-depth understanding of these concepts.

References

1. A. Gelb, et al.: Applied Optimal Estimation. The MIT Press (1974)
2. Alvares, L., Bogorny, V., Kuijpers, B., de Macelo, J., Moelans, B., Palma, A.: Towards semantic trajectory knowledge discovery. Tech. rep., Hasselt University (2007)
3. Bellman, R.: On the Approximation of Curves by Line Segments Using Dynamic Programming. Communications of the ACM **4**(6), 284 (1961)
4. Bogorny, V., Valiati, J., Alvares, L.: Semantic-based Pruning of Redundant and Uninteresting Frequent Geographic Patterns. Geoinformatica **14**(2), 201–220 (2010)
5. Civilis, A., Jensen, C., Nenortaite, J., Pakalnis, S.: Efficient Tracking of Moving Objects with Precision Guarantee. In: IEEE International Conference on Mobile and Ubiquitus Systems: Networking and Services (MobiQuitous) (2004)
6. C.T. Lawson and S.S. Rvi and J.-H. Hwang: Compression and Mining of GPS Trace Data: New Techniques and Applications. Technical Report. Region II University Transportation Research Center
7. van Diggelen, F.: GNSS Accuracy: Lies, Damn Lies, and Statistics. GPS World (2007)
8. Doucet, A., Freitas, N., Gordon, N.: An Introduction to Sequential Monte Carlo Methods. In: Sequential Monte Carlo Methods in Practice., pp. 3–13. Springer: New York (2001)
9. Douglas, D., Peucker, T.: Algorithms for the Reduction of the Number of Points Required to Represent a Line or its Caricature. The Canadian Cartographer **10**(2), 112–122 (1973)
10. Fox, D.: Adapting the Sample Size in Particle Filters Through KLD-Sampling. The International Journal of Robotics Research **22**(12), 985–1003 (2003)
11. Giannotti, F., Nanni, M., Pedreschi, D., Renso, C., Trasarti, R.: Mining Mobility Behavior from Trajectory Data. In: International Conference on Computational Science and Engineering (CSE)., pp. 948–951 (2009)
12. Gordon, N.: Bayesian Methods for Tracking. Imperial College, University of London, London (1994)
13. Guting, R., M.H.Bohlen, Erwig, M., Jensen, C., Lorentzos, N., M, S., Vazirgiannis, M.: A Foundation for Representing and Querying Moving Objects. ACM Transaction on Database Systems (TODS) **25**, 1–42 (2000)

14. G"uting, R., Schneider, M.: Moving Object Databases. Morgan Kaufmann, San Francisco, CA (2005)
15. Hershberger, J., Snoeyink, J.: Speeding up the Douglas-Peucker Line simplification Algorithm. In: International Symposium on Spatial Data Handling, pp. 134–143 (1992)
16. Hightower, J., Borriello, G.: Particle Filters for Location Estimation in Ubiquitous Computing: A Case Study. In: in 6th International Conference on Ubiquitous Computing., pp. 88–106 (2004)
17. Hung, C.C., Peng, W.C.: Model Driven Traffic Data Acquisition in Vehicle Sensor Networks. In: International Conference of Parallen Processing (ICPP)., pp. 424–432 (2011)
18. Jenks, G.: Lines, Computers, and Human Frailties. Annuals of the Association of American Geographers **71**, 1–10 (1981)
19. Keogh, E., Chu, S., Hart, D., Pazzani, M.: An On-Line Algorithm for Segmenting Time Series. In: International Conference on Data Mining (ICDM), pp. 289–296 (2001)
20. Keogh, E., Pazzani, M.: An Enhanced Representation of Time Series which Allows Fast and Accurate Classification, Clustering and Relevance Feedback. In: International Conference of Knowledge Discovery and Data Mining (KDD)., pp. 239–241 (1998)
21. Keogh, E., Pazzani, M.: An On-Line Algorithm for Segmenting Time Series. In: Annual International ACM-SIGIR Conference on Research and Development in Information Retrieval (SIGIR). (1999)
22. Keogh, E., Smyth, P.: A Probabilistic Approach to Fast Pattern Matching in Time Series Databases. In: International Conference of Knowledge Discovery and Data Mining (KDD)., pp. 24–30 (1997)
23. Kleinberg, J., Tardos, E.: Algorithm Design. Addison Wesley, Reading, MA (2005)
24. Maratnia, N., de By, R.: Spatio-Temporal Compression Techniques for Moving Point Objects. In: International Conference on Extending Database Technology (EDBT), pp. 765–782 (2004)
25. McMaster, R.: Statistical Analysis of Mathematical Measures of Linear Simplification. The American Cartographer **13**, 103–116 (1986)
26. Murphy, K., Russell, S.: Rao-Blackwellised Particle Filtering for Dynamic Bayesian Networks. In: Sequential Monte Carlo Methods in Practice., pp. 499–515. Springer: New York (2001)
27. Potamias, M., Patroumpas, K., Sellis, T.: Sampling Trajectory Streams with Spatio-Temporal Criteria. In: International Conference on Scientific and Statistical Database Management (SSDBM), pp. 275–284 (2006)
28. Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M.: Indexing the Positions of Continuously Moving Objects. In: ACM International Conference on Management of Data (SIGMOD), pp. 331–342 (2000)
29. Schmid, F., Richter, K.F., Laube, P.: Semantic Trajectory Compression. In: International Symposium on Advances in Spatial and Temporal Databases (SSTD). (2009)
30. Vitter, J.: Random sampling with a reservoir. ACM Transactions on Mathematical Software (TOMS) **11**(1) (1985)
31. Wang, X., Tieu, K., Grimson, E.: Learning Semantic Scene Models by Trajectory Analysis. In: European Conference on Computer Vision (ECCV), pp. 110–123 (2006)
32. White, E.: Assessment of Line Generalization Algorithms. The American Cartographer **12**, 17–27 (1985)
33. Wolfson, O., Sistla, P., Xu, B., Zhou, J., Chamberlain, S., Yesha, Y., Rishe, N.: Tracking Moving Objects Using Database Technology in DOMINO. In: The Fourth Workshop on Next Generation Information Technologies and Systems (NGITS), pp. 112–119 (1999)
34. Yan, Z.: Towards Semantic Trajectory Data Analysis: A Conceptual and Computational Approach. In: Internation Conference on Very Large Data Base (VLDB) PhD Workshop. (2009)