A framework for trajectory segmentation by stable criteria and Brownian Bridge Movement Model

Alewijnse, S.P.A.

Award date:
2013

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science
Den Dolech 2, 5612 AZ Eindhoven
P.O. Box 513, 5600 MB Eindhoven
The Netherlands
www.tue.nl

Supervisors
dr. Kevin Buchin (TU/e)
dr. Michel A. Westenberg (TU/e)

Section
Algorithms and Visualization

Date
September 13, 2013

A Framework for Trajectory Segmentation by Stable Criteria and Brownian Bridge Movement Model

Master's Thesis

Sander P. A. Alewijnse

Abstract

Criterion-based segmentation is the problem of subdividing a trajectory into a small number of parts such that each part satisfies a global criterion. We present an algorithmic framework for criterion-based segmentation of trajectories that can efficiently process a large class of criteria. Our framework can handle criteria that are stable, in the sense that these do not change their validity along the trajectory very often. Our framework takes $O(n \log n)$ time for preprocessing and computation, where n is the number of data points. It improves upon the two previous algorithmic frameworks on criterion-based segmentation, which could only handle decreasing monotone criteria, or had a quadratic running time, respectively. Furthermore, we propose a new segmentation method based on the dynamic Brownian Bridge Movement Model. This segmentation method has only one parameter: the segment penalty factor, which can be chosen automatically or interactively using a so-called stability diagram. We finally show how to combine the dBBMM-based method with criteria.

Acknowledgements

First of all, I would like to thank my supervisors Kevin Buchin and Michel Westenberg, for their guidance and feedback. Furthermore, I would like to thank Maike Buchin, Stef Sijben, Bart Kranstauber, Kamran Safi, Erik Willems, and Mark de Berg for the helpful discussions we had. Also, I would like to thank Andrea Kölzsch for her contribution to the case study on geese. Moreover, I would like to thank the geese Adri and Kees, and the fishers [1] Ricky and Leroy for providing trajectory data, and their captors, in particular respectively Helmut Kruckenberg and Scott LaPoint.

Finally, I would like to thank my parents and sister for supporting me throughout my studies. You have always been there to listen, motivate and aid me otherwise. I would also like to thank my friends, without whom my student life would not have been the same. In particular, Quirijn Bouts and Alex ten Brink, with whom I worked on numerous interesting projects.

Sander P. A. Alewijnse

Eindhoven
September 13, 2013

[1] North American member of the weasel family: Martes pennanti

Contents

Chapter 1

Introduction

Over the past few years movement tracking devices have become widely available for all kinds of applications. Many cars have a GPS receiver installed for navigation purposes. In sports, modern-day hikers, bikers and runners can track their movements using GPS trackers. Nowadays it is even possible for smartphones to measure their location, which enables the development of all kinds of smartphone-applications that make use of this information.

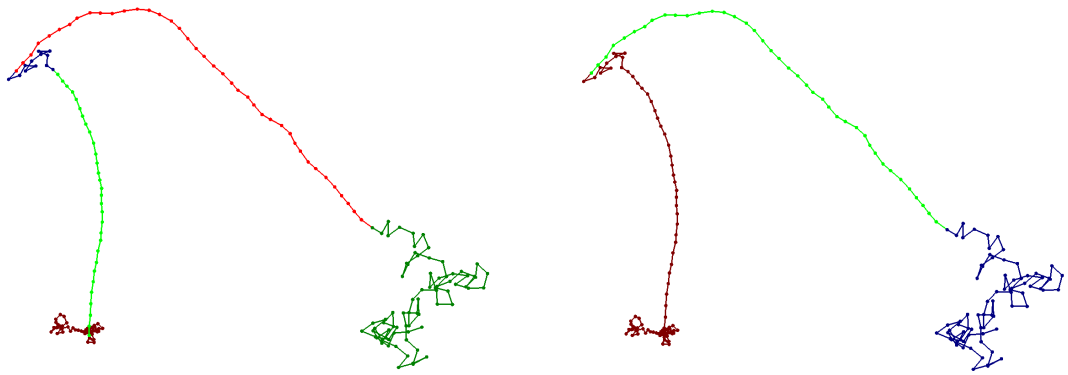
The recent advances in movement tracking technology are not just a significant development in the consumer electronics sector. A broad variety of scientific disciplines, including traffic analysis, geography, market research, surveillance, security and ecology, show an increasing interest in movement patterns of entities moving in various spaces over various times scales.

Tracking devices record a so-called trajectory, which is a series of timestamped locations; that is, latitude-longitude pairs accompanied by a timestamp. Most devices also record other attributes, such as velocity, acceleration and bounds on the inaccuracies of the recorded values. Currently, the amount of recorded data is rapidly increasing, and methods are needed for processing and analyzing these data.

There are numerous analysis tasks concerning trajectory data. Many of those tasks are of a geometric nature and are studied in computational geometry. Computational geometry is the branch of computer science dedicated to algorithms that solve problems, concerning geometric objects such as points, lines and planes [10]. Computational geometry emerged from the field of algorithms design and analysis about forty years ago, and it has grown into a very active research discipline ever since, mainly due to the great technological advances in its application domains, including computer graphics, robotics and geographic information systems (GIS).

Various problems concerning trajectories have been studied in this field of research. A basic, yet complicated problem is defining a good metric that measures the similarity between trajectories [1], which can be used in for instance the search for similar subtrajectories [4]. Another example is the popular places problem [3], which is about finding the most frequently visited regions, given a number of trajectories.

We study the following important analysis task: finding a *segmentation* of a trajectory. Segmenting a trajectory means "splitting" the trajectory into pieces, which are called *segments*. There are two conflicting goals of segmenting a trajectory. First of all the movement within each segment should be homogeneous in some sense. Note that this homogeneity can be defined in multiple ways, for instance, the speed, heading or location should not vary much inside a segment. Secondly the number of segments should be small, which is equivalent to segments being long. Determining the best tradeoff between the two goals is not an easy process and depends on the context in which the segmentation is used [17]. In Figure 1.1 this issue is illustrated by two segmentations of the same trajectory, which are segmented at different scales (different tradeoffs between homogeneity and number of segments).



(a) Small scale (b) Large scale

Figure 1.1: Two segmentations of the same trajectory at different scales. Segments are shown in different colors.

In this thesis we focus on criterion-based segmentation. In this setting the number of segments is minimized subject to the constraint that each segment is homogeneous according to a formal *criterion*. There are numerous criteria that can be used. For example, we can bound the maximal speed range, or require that each segment fits in a disk of a certain radius.

Usually, segmentation is part of a larger analysis process. It is often combined with classification of the segments to give the segments a meaning. For instance the segments of an bird trajectory could be classified into resting, flying, and eating. An example of a classified segmentation is shown in Figure 1.2.

We have developed a new framework for criterion-based segmentation. Compared to other efficient criterion-based methods it allows for a much broader class of criteria. It can for instance handle outlier-tolerant criteria. Furthermore, our framework incorporates the classification of segments in the segmentation method, which allows us to enforce rules based on segment classes, such as restrictions of the form "a hunting segment must be followed by a resting segment or an eating segment".

An important issue regarding the criterion-based segmentation method is determining the exact criteria and their parameter values. That is why we have developed a novel segmentation method that does not rely on specific parametrized criteria, but on a statistical movement model: the dynamic Brownian Bridge Movement Model (dBBMM) [15, 16].

Given a trajectory (a sequence of measured points) the dBBMM models the movement in between the measured points. The model has only one parameter: the Brownian motion variance, which corresponds to the animal's mobility along the trajectory.

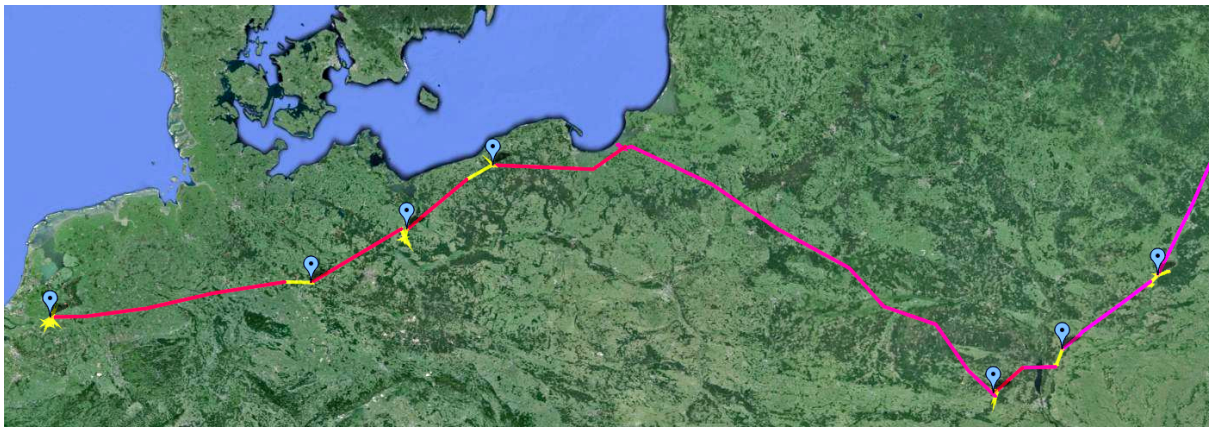


Figure 1.2: Segmentation of geese data. Red/pink indicates migration flight, yellow stopovers. Blue markers indicate the end of a stopover.

In the research field of the movement ecology, this model has been used for various trajectory analysis tasks. For instance, the model was used on trajectories of simultaneously moving animals to compute where animals encounter each other, and whether they avoid, attract or follow each other [5]. Furthermore, it has been used to estimate the home range of animals, which is (intuitively) the area in which animals are expected to spend most time. All this information can be of great importance in the organization of wildlife conservation.

Our novel segmentation method based on the dBBMM is essentially a model fitting algorithm, which returns a segmentation as part of the model-fit. The model fitting procedure is guaranteed to be optimal with respect to a quality metric called the information criterion. Choosing the "right" information criterion is a complex procedure which requires domain-knowledge. We have developed means to provide insight in the process of picking it. Figure 1.1 shows two segmentations returned by the algorithm for different information criteria. Furthermore we have combined the dBBMM-based segmentation method with the criterion-based segmentation methods to get the best of both worlds.

1.1 Related work

There has been a lot of research on computational trajectory analysis [13], even on the segmentation problem alone. Many segmentation algorithms were developed, each formalizing the segmentation goals in their own way. Previous work focused on finding a semantic annotation of the trajectory [14], profile based segmentation [11] and criterion-based segmentation [6]. In this thesis we focus on the latter type of segmentation.

There are two ways in which trajectories have been handled in this context: *continuous* and *discrete*. In the continuous variant the trajectory is interpolated to obtain a continuous trajectory that can be split into segments at any point on its interpolated curve. In the discrete variant the segments have to start and end at recorded points. Hence, a discrete segmentation is only based on the recorded data and not on (possibly incorrectly) interpolated points in between. On the other hand, a continuous segmentation is more flexible in terms of splitting points than in the discrete problem. However, it is unclear when or whether this added flexibility improves the segmentation result. Therefore we focus on the discrete segmentation problem. Others have developed methods that apply to both the discrete and the continuous problem.

Buchin *et al.* [6] have developed an algorithmic framework that computes a (discrete or continuous) segmentation given a *decreasing monotone*[1] criterion. Many simple criteria concerning homogeneity of location, speed and heading are in this class. Also disjunctions and conjunctions of those simple criteria belong to this class of criteria. In this framework, a segmentation can be computed in $O(n \log n)$ time for discrete and continuous problem, where n is the number of data points.

Their framework has been used to segment animal tracks of migrating geese [7]. In this setting, the segmentation is based on multiple criteria, each of them defining a class corresponding to a *movement state*, in this case stopover and migration flight, which provided means to classify the segments.

There are many meaningful criteria that are not decreasing monotone. These criteria are in general harder to handle. Aronov *et al.* [2] showed that finding a valid segmentation given a non-decreasing-monotone criterion is a hard problem in the continuous setting. For several specific non-decreasing-monotone criteria, most importantly the standard deviation criterion (with linear interpolation), they present a polynomial time segmentation algorithm.

For a generic non-decreasing-monotone criterion in the discrete setting they present a simple algorithm with running time $\Theta(n^2)$. We note that this algorithm is not very suitable for trajectories with a large number of data points, due to the quadratic running time. In practice heuristics have been applied to handle criteria that are not decreasing monotone [7].

All the related work described above deals with discrete trajectories, or with continuous trajectories that are assumed to be completely known. For the analysis of animal trajectories this is often not realistic. Therefore studies have focused on estimating those continuous trajectories, given a set of discrete measurements (that might even contain inaccuracies). In this context the *Brownian Bridge Movement Model* (BBMM) [15] proved to be a very useful model.

The BBMM is based on the properties of a conditional random walk between successive pairs of locations, dependent on the time between locations, the distance between locations, and the Brownian motion variance, which is related to the animal's mobility. This Brownian motion variance is assumed to be a constant in the BBMM, and it can be estimated by a maximum likelihood method.

However, different behavioral states in animal movement correspond to different values of the Brownian motion variance [16]. Hence, for many real world trajectories the Brownian motion variance cannot be assumed to be constant. There is an extended version of the BBMM that assumes a varying instead of a constant Brownian motion variance. It is called the dynamic Brownian Bridge Movement Model (dBBMM). Kranstauber *et al.* [16] proposed a windowing method to fit a discrete trajectory with the dBBMM. The method estimates the Brownian motion variance at each point along the trajectory.

[1] These criteria are called monotone in [6].

1.2 Results and organisation

In Chapter 2 we discuss our criterion-based segmentation framework, which allows for handling a broad class of criteria: the *stable* criteria, in $O(n \log n)$ time, with n the number of data points. This includes decreasing and increasing monotone criteria and Boolean combinations of them. Furthermore, our framework allows for efficient outlier handling. We can approximate an outlier-tolerant criterion [2] at the expense of an extra factor $\log n$ in the running time.

The framework also allows for segmentation by movement states. In this setting the segmentation is combined with classification. The concept of movement states allows for a broad range of additional segmentation rules. First of all, rules can be defined for breaking ties. The criterion-based segmentation problem only minimizes the number of segments. Segmentations with equal segment count are considered equal. However, in practice some segmentations are better than others, despite having equal segment count, for example because of the exact location of segment boundaries. Optimization rules that distinguish between those segmentations can be formalized in terms of movement states.

Secondly, we can add restrictions on the state transitions by enforcing rules of the form "in every segmentation movement state A can only be followed by movement state B, C or D". The framework also allows for penalization of certain state transitions, instead of forbidding those transitions.

We have also developed means for interactive parameter selection, the stability diagram. Furthermore, we did a case study on data of migrating geese, in which we applied the methods we developed.

Chapter 3 is about our novel segmentation method which is based on the dBBMM. This method is in its essence a method that estimates the Brownian motion variance along the trajectory. Changes in this variance define the segments. The variances are chosen in such a way that they optimize a certain information criterion goal function, which consists of a part that measures the quality of the segmentation, and a part that equals the number of segments times a penalty factor. The penalty factor is the only parameter of the model. We provide means to find suitable values for this parameter in the form of a stability diagram.

Finally, we show how to add criteria to the dBBMM-based method. First of all we show how to add an increasing monotone criterion. Secondly we show that the whole criterion-based framework that we presented in Chapter 2, can be combined with the dBBMM-based segmentation method. The segmentation algorithm concerning this combined method is fundamentally different from the original dBBMM-based method, and is basically an extension/adaptation of the framework of Chapter 2.

Chapter 2

Criterion-based segmentation

In this chapter a new framework for criterion-based segmentation is presented. First we formalize the criterion-based segmentation problem in Section 2.1. In Section 2.2 we discuss two important criteria classes: the decreasing and increasing monotone criteria. Furthermore, we state some basic properties of those criteria.

In Sections 2.3-2.5 our new segmentation method is presented. Our approach is somewhat similar to the method by Aronov *et al.* [2]. They presented (as a side note) a simple algorithm for discrete segmentation based on any computable criterion. The first step of this method is to compute the *start-stop matrix*. The storage of this matrix takes $\Theta(n^2)$ space. The second step is the computation of the actual segmentation from this matrix using a simple Dynamic Program (DP) with running time $\Theta(n^2)$.

Our approach consists of two steps with similar goals. First we construct a *compressed start-stop matrix*. This data structure can efficiently test for any candidate segment whether it satisfies the criterion. The compressed start-stop matrix is of size $O(n)$ and can be computed in $O(n \log n)$ time for a broad class of criteria. It is discussed in more detail in Section 2.3. In Section 2.4 we discuss how to compute this data structure for several specific criteria.

In the second step the actual segmentation is computed from this compressed start-stop matrix in $O(n \log n)$. This algorithm is described in Section 2.5. Moreover, we can combine the segmentation with classification by movement states and put in extra segmentation rules for the movement states, as is described in Section 2.6.

Choosing the segmentation criterion parameters is crucial but also difficult in practice, because it requires domain knowledge. To fine-tune the parameters, the segmentation needs to be computed multiple times, each time using a slightly different setting of parameters. To make this interactive process easier, we introduce the *stability diagram* in Section 2.7, which supports the user with some visual feedback.

We have experimented with our framework and present a case study on data of migrating geese in Section 2.8.

2.1 Criterion-based segmentation

Throughout the thesis we use the following definition of a trajectory.

Definition 1 *A* trajectory τ *is given by a sequence of* n *triples* (x_i, y_i, t_i)*, where* (x_i, y_i) *is the location of a moving entity (e.g., an animal) at time* t_i*. We denote the timestamped locations of* τ *by* $\tau(i) = (x_i, y_i)$*.*

We treat a trajectory as a discrete sequence of timestamped locations, hence a subtrajectory can only start and end at recorded time stamps. A subtrajectory of τ starting at time t_i and ending at time t_j is denoted by $\tau[i, j]$. If a subtrajectory τ' is completely covered by a subtrajectory τ'' we denote this by $\tau' \subseteq \tau''$.

This thesis is about *segmenting* trajectories, which is defined as follows:

Definition 2 *A* segmentation *of a trajectory is a partition of a trajectory τ in subtrajectories called* segments. *These segments are disjoint (except for their endpoints) and cover the whole trajectory τ.*

We use k to denote the number of segments and s_i for $i = 0, 1, \ldots, k$ to denote the end points of the ith segment. The points $\tau(s_i)$ for $i = 0, 1, \ldots, n$ are called *splitting points*. A segmentation is thus given by $\tau[s_0, s_1]$, $\tau[s_1, s_2]$, $\ldots$, $\tau[s_{k-1}, s_k]$, with $0 = s_0 < s_1 < \cdots < s_k = n$.

Our segmentation approach is based on *criteria*, which are defined in Definition 3.

Definition 3 *A* criterion *C is a function that maps the set of subtrajectories (candidate segments) to the Booleans.*

The value of C for candidate segment $\tau[i, j]$ is denoted by $C(i, j)$. The idea is that criteria indicate whether a candidate segment is "homogeneous enough". A candidate segment that satisfies C is called a *valid segment*, and a segmentation consisting only of valid segments is called a *valid segmentation*. The goal of *criterion-based* segmentation is to find a valid segmentation of τ with a minimal number of segments, given a trajectory τ and a criterion C. Such a segmentation is called *optimal*.

2.2 Criteria

Not all criteria can be handled in the same way. Different criteria can require fundamentally different segmentation algorithms. Previous work focused mainly on the class of *decreasing monotone* criteria, which can be handled in a similar way [6].

Definition 4 *A criterion is* decreasing monotone *if and only if it has the property that if it holds on a certain candidate segment τ', it also holds on every subsegment $\tau'' \subseteq \tau'$ of that segment. This is depicted in Figure 2.1.*

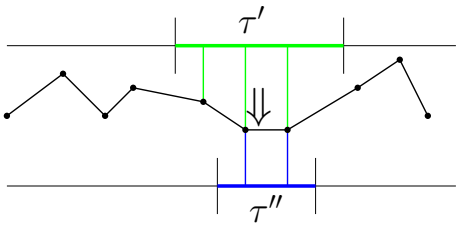


Figure 2.1: For decreasing monotone criteria the validity of τ' implies the validity of τ''.

There are numerous examples of this kind. For instance, criteria that bound the range of a trajectory attribute (such as speed and heading) are decreasing monotone. Requiring that a segment fits a fixed size disk yields a decreasing monotone criterion as well. Furthermore, combinations of decreasing monotone criteria are decreasing monotone criteria too, as is stated in Theorem 1.

Theorem 1 *[6, Theorem 15] A combination of conjunctions and disjunctions of decreasing monotone criteria is a decreasing monotone criterion.*

Consider the following greedy strategy to segment a trajectory. Start at the beginning of the trajectory, and make the first segment as long as possible according to the criterion. Now start the second segment at the end of the first one, and also make it as long as possible. Repeat the process until the end of the trajectory is reached. Previous work showed that applying this greedy strategy results in an optimal result in case of a decreasing monotone criterion.

Theorem 2 *[6, Theorem 4] For decreasing monotone criteria, the greedy segmentation strategy yields an optimal segmentation.*

Our framework can also handle a fundamentally different class of criteria: the class of the *increasing monotone* criteria.

Definition 5 *A criterion is* increasing monotone *if and only if it has the property that if it holds on a certain candidate segment τ', it also holds on every supersegment $\tau'' \supseteq \tau'$ of that segment. This is depicted in Figure 2.2.*

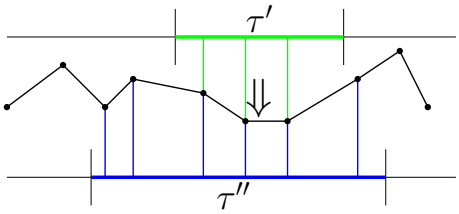


Figure 2.2: For increasing monotone criteria the validity of τ' implies the validity of τ''.

Important examples are the minimum length and duration criterion that place lower bounds on length and duration of a segment, respectively. Furthermore, combinations of increasing monotone criteria are increasing monotone criteria. This is proven in Theorem 3.

Theorem 3 *A combination of conjunctions and disjunctions of increasing monotone criteria is an increasing monotone criterion.*

Proof. Let C_1 and C_2 be increasing monotone criteria. We show that the conjunction $C_1 \wedge C_2$ and the disjunction $C_1 \vee C_2$ are increasing monotone. First we consider the conjunction. Let τ' be a subtrajectory of τ. Assume that $C_1 \wedge C_2$ is satisfied by τ'. Let τ'' be a supertrajectory of τ'. τ'' satisfies criterion C_1, because τ' satisfies C_1. The same holds for C_2. This implies that $C_1 \wedge C_2$ is satisfied by τ''. Hence $C_1 \wedge C_2$ is increasing monotone.

Now the consider the disjunction. Let τ' be a subtrajectory of τ. Assume that $C_1 \vee C_2$ is satisfied by τ'. Let τ'' be a supertrajectory of τ'. Without loss of generality assume that criterion C_1 is satisfied by τ'. Then, C_1 is also satisfied by τ''. Hence $C_1 \vee C_2$ is satisfied by τ''. This proves that $C_1 \vee C_2$ is increasing monotone. □

Segmentation based on increasing monotone criteria alone has a meaningless result: either the whole trajectory is "segmented" into one segment, or the trajectory cannot be segmented at all. However, Boolean combinations of increasing and decreasing monotone criteria can yield meaningful results. For a combination of increasing and decreasing monotone criteria, the greedy strategy does *not* always yield an optimal result, as is proven in Theorem 4.

Theorem 4 *For a combination of decreasing and increasing monotone criteria, the greedy segmentation strategy does not necessarily yield an optimal segmentation.*

Proof. Consider the trajectory $\tau[0,5]$ in Figure 2.3. It is a regularly sampled track ($t_i = i$ for all $i = 0,\ldots,5$) of an object that is moving with constant acceleration ($\tau(i) - \tau(i-1) = i$ for all $i = 1,\ldots,5$) starting at $\tau(0) = (0,0)$. Assume that the criterion is a conjunction of a duration criterion D (increasing monotone) and a bounded speed range criterion S (decreasing monotone). Criterion D requires a minimum segment duration of 3 and criterion S allows for a maximum speed range of 4, where the speed is estimated by forward-differentiation.

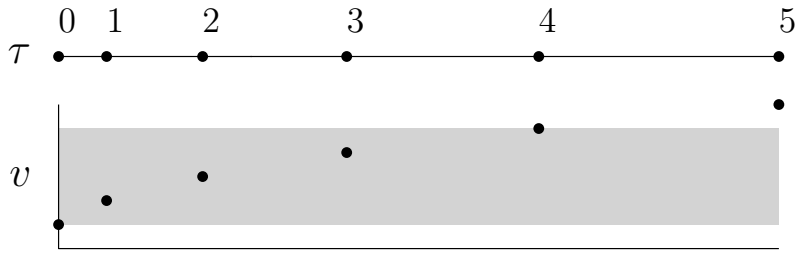


Figure 2.3: Trajectory τ and speed v along τ. The gray box has height 4 and indicates that $\tau[0,4]$ satisfies S.

In this case, the greedy strategy picks $\tau[0,4]$ as first segment, since this is the longest segment starting at t_0 that satisfies $D \wedge S$. There is no segment starting at t_4 that satisfies D, so the greedy strategy fails to find a segmentation. However, there is an (optimal) segmentation $\tau[0,2], \tau[2,5]$ which satisfies $D \wedge S$. □

Our framework is not limited to decreasing or increasing monotone criteria. We can handle a more general class of criteria: the *stable* criteria. Stable criteria do not change their validity along the trajectory very often. This is formalized in Section 2.3.2. This class of criteria contains Boolean combinations of increasing and decreasing monotone criteria.

2.3 Compressed start-stop matrix

2.3.1 Start-stop matrix

A *start-stop matrix* stores the relation between a trajectory τ and a criterion C. Consider the parameter space of the set of subtrajectories of τ. For any candidate segment $\tau[i,j]$, the start parameter i is associated with the column index and the stop parameter j with the row index of the matrix. So a matrix entry (i,j) in the upper left triangle ($i \leq j$) represents a candidate segment. Each of those is assigned a value $C(i,j)$, which is true if the criterion C is satisfied by the candidate segment and false otherwise. A candidate segment is called part of the *free space* if it satisfies C and it is part of the *forbidden space* otherwise.

Consider a segmentation of τ into a sequence of segments $\tau[s_0,s_1]$, …, $\tau[s_{k-1},s_k]$. Consecutive segments $\tau[s_i,s_{i+1}], \tau[s_{i+1},s_{i+2}]$ share a trajectory point. This means that the row index of the matrix entry corresponding to $\tau[s_{i+1},s_{i+2}]$ is equal to the column index of the matrix entry corresponding to $\tau[s_i,s_{i+1}]$. Hence the matrix entries corresponding to the segments $\tau[s_i,s_{i+1}]$ together with the entries (s_i,s_i) on the main diagonal form a staircase. Furthermore, a segmentation is valid if and only if the non-diagonal vertices of the staircase lie in the free space. See Figure 2.4 for an example of a valid segmentation.

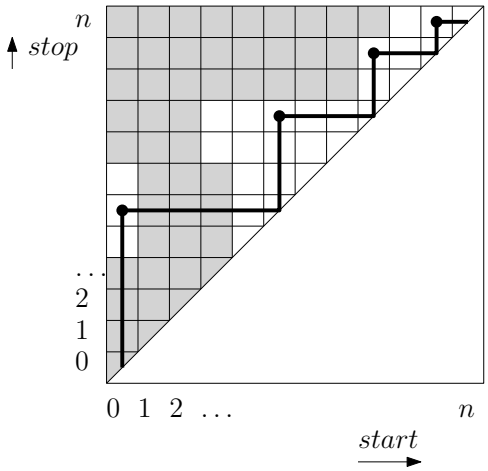


Figure 2.4: A start-stop matrix and an optimal segmentation into four segments. The free space is white, the forbidden space is gray. The segmentation is valid, because the four vertices corresponding to segments (indicated by dots) lie in the free space.

2.3.2 Compressing the start-stop matrix

For many criteria the start-stop matrix can be compressed significantly by applying run-length encoding to each row. Run-length encoding is a simple form of data compression in which *runs* are stored in a compressed form [12]. A run is a sequence of consecutive values that are equal. In our case, we have runs of *true* values and runs of *false* values, which we call *blocks* and *gaps*, respectively. Runs are stored as pair of *value* and *count*. We call this row-wise run-length encoded start-stop matrix the *compressed start-stop matrix*.

Consider the start-stop matrix for a decreasing monotone criterion. The property "if C is satisfied by a certain segment, it is also satisfied by every subsegment" implies that all matrix entries to the right of a true value must be true. Recall that a matrix entry (i', j) right of a matrix entry (i, j) corresponds to $\tau[i', j]$ being a subsegment of $\tau[i, j]$. A row of a decreasing monotone start-stop matrix hence consists of at most two runs: an optional gap followed by an optional block. An example of a start-stop matrix for a decreasing monotone criterion is shown in Figure 2.5(a).

In a similar way the start-stop matrix for an increasing monotone criterion can be compressed. The increasing monotone property implies that all matrix entries to the left of a true value must be true. A row of such a start-stop matrix hence consists of an optional block followed by an optional gap. An example is shown in Figure 2.5(b).

Using the compressed instead of the uncompressed start-stop matrix for decreasing and increasing monotone criteria reduces the storage to $O(n)$. In Section 2.4 we show that this compressed start-stop matrix can be computed in $O(n \log n)$ for many decreasing and increasing monotone criteria.

Stable criteria

Our framework is not limited to decreasing and increasing monotone criteria. In fact, it can handle any criterion that has a computable compressed start-stop matrix. The running time of the algorithm (described in Section 2.5) that computes the optimal segmentation from a compressed start-stop matrix is $O((\lambda + n) \log n)$, where λ is the number of blocks in the compressed start-stop matrix.

The λ measures the stability of the criterion. If it is low, the criterion does not change its validity along the trajectory very often. The most interesting class of criteria is the class of criteria that have a compressed start-stop matrix with $\lambda = O(n)$ blocks. We call those criteria *stable*. Furthermore, we use the following more general definition.

Definition 6 *A criterion is* λ-stable *if and only if it has the property that* $\sum_{j=0}^{n} v(j) = \lambda$, *where* $v(j)$ *denotes the number of times the validity of the candidate segments* $\tau[i, j]$ *changes, for* $i = 0, 1, \ldots, j-1$.

We have already seen two examples of stable criteria:

Theorem 5 *A decreasing monotone criterion is a* λ*-stable criterion, with* $\lambda \leq n$, *where* n *is the number of points on the trajectory.*

Theorem 6 *An increasing monotone criterion is a* λ*-stable criterion, with* $\lambda \leq n$, *where* n *is the number of points on the trajectory.*

There are also stable criteria that are neither decreasing monotone or increasing monotone. In the next sections some various examples are given.

2.3.3 Combining and transforming compressed start-stop matrices

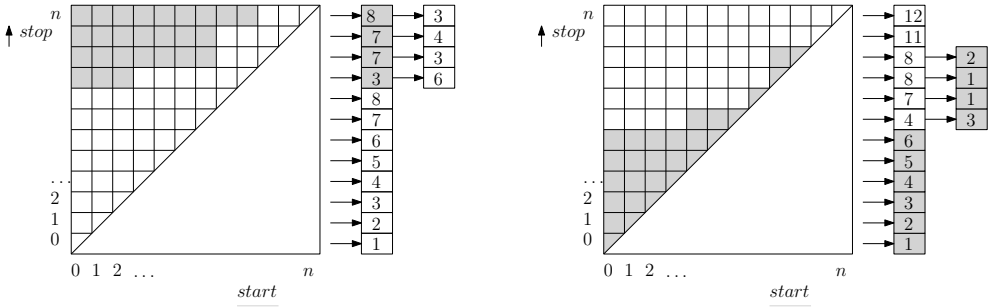
Decreasing and increasing monotone criteria can be combined to get compound criteria, which can be more effective at segmenting trajectories than singleton criteria, as has been demonstrated in [7]. There are two ways to combine criteria: the *conjunction* and *disjunction*.

More general, given a λ_1-stable criterion C_1 and a λ_2-stable criterion C_2 and their compressed start-stop matrices, the compressed start-stop matrix of $C_1 \wedge C_2$ can be computed efficiently. The criterion $C_1 \wedge C_2$ is satisfied by a candidate segment $\tau[i, j]$ if and only if C_1 and C_2 are satisfied. The key observation is that the free space of the start-stop matrix of $C_1 \wedge C_2$ equals the *intersection* of the free space of the start-stop matrices of C_1 and C_2. This implies that the criterion $C_1 \wedge C_2$ is $\lambda_\wedge$-stable, with $\lambda_\wedge \leq \lambda_1 + \lambda_2$. The run-length encoded form of this intersection can computed per row taking in total $O(\lambda_1 + \lambda_2 + n)$ time. An example is given in Figure 2.5.

Similarly, the compressed start-stop matrix of $C_1 \vee C_2$ equals the *union* of the free space of the start-stop matrices of C_1 and C_2. The criterion $C_1 \vee C_2$ is hence $\lambda_\vee$-stable, with $\lambda_\vee \leq \lambda_1 + \lambda_2$. This union can also be computed in $O(\lambda_1 + \lambda_2 + n)$ time. This is summarized in the following lemmas.

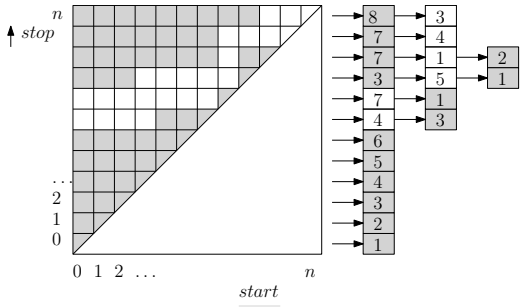
Lemma 7 *Given a* λ_1*-stable criterion* C_1 *and a* λ_2*-stable criterion* C_2, *the criteria* $C_1 \wedge C_2$ *is* $\lambda_\wedge$*-stable, with* $\lambda_\wedge \leq \lambda_1 + \lambda_2$. *Given the compressed start-stop matrices of* C_1 *and* C_2 *the compressed start-stop matrix of* $C_1 \wedge C_2$ *can be computed in* $O(\lambda_1 + \lambda_2 + n)$ *time.*

Lemma 8 *Given a* λ_1*-stable criterion* C_2 *and a* λ_2*-stable criterion* C_2, *the criteria* $C_1 \vee C_2$ *is* $\lambda_\vee$*-stable, with* $\lambda_\vee \leq \lambda_1 + \lambda_2$. *Given the compressed start-stop matrices of* C_1 *and* C_2 *the compressed start-stop matrix of* $C_1 \vee C_2$ *can be computed in* $O(\lambda_1 + \lambda_2 + n)$ *time.*



(a) Decreasing monotone criterion C_1

(b) Increasing monotone criterion C_2



(c) Criterion $C_1 \wedge C_2$

Figure 2.5: Two start-stop matrices and their conjunction.

Criteria can also be transformed by applying *negation*. Given a λ-stable criterion and its compressed start-stop matrix the compressed start-stop matrix of the negation of the criterion can be computed efficiently: Change the forbidden space to free space and vice versa. For a row with an even number of runs this does not change anything to the number of blocks. For a row with an odd number of runs this could decrease or increase the number of blocks by one. Hence the resulting criterion is $\lambda_\neg$-stable, with $\lambda_\neg \leq \lambda + n$. The computation takes $O(\lambda + n)$ time.

Lemma 9 *The negation of a λ-stable criterion C is a $\lambda_\neg$-stable criterion, with $\lambda_\neg \leq \lambda + n$. Given the compressed start-stop matrix of C the compressed start-stop matrix of $\neg C$ can be computed in $O(\lambda + n)$ time.*

We can formulate complex compound criteria that consist of a number of simple criteria that are combined using conjunctions, disjunctions and possibly transformed (at any level in the criterion expression) using negations. We call such a compound criterion a Boolean combination. The following theorem is a direct consequence of Theorems 5, 6, and Lemmas 7, 8 and 9.

Theorem 10 *A Boolean combination of a constant number of decreasing and increasing monotone criteria is a stable criterion. Given the compressed start-stop matrices of all those criteria the compressed start-stop matrix of the Boolean combination can be computed in $O(n)$ time, where n is the number of points on the trajectory.*

Applying negation to a decreasing or increasing monotone criterion has a special property, which is useful in Section 2.4, where we show how to compute the compressed start-stop matrix for decreasing and increasing monotone criteria.

Lemma 11 *The negation of a decreasing monotone criterion is an increasing monotone criterion and vice versa.*

Proof. Let τ be a trajectory. Let C be a decreasing monotone criterion. Assume for the purpose of contradiction that $\neg C$ is not increasing monotone. This means that there is a candidate segment $\tau[i,j]$ that does not satisfy C, for which a supersegment $\tau[i',j']$ exists with $i' \leq i$ and $j' \geq j$ that satisfies C. However, validity of $\tau[i',j']$ implies validity of $\tau[i,j]$ by the decreasing monotonicity of C. This is a contradiction. The proof of the other direction is analogous. □

2.4 Computing the compressed start-stop matrix

The compressed start-stop matrix for a decreasing monotone criterion can be computed using the algorithm *ComputeLongestValid*. Given a trajectory data set τ and a decreasing monotone criterion C, the algorithm computes for every trajectory index j the smallest index i for which $\tau[i,j]$ satisfies the criterion. This index is stored in LV_j. Given LV_j it is straightforward to compute the actual compressed start-stop matrix in $O(n)$ time. Note that for *increasing monotone* criteria the compressed start-stop matrix can be computed using the same algorithm. Simply replace $\neg C$ by C in *ComputeLongestValid* and negate the resulting compressed start-stop matrix. The correctness of this method is a direct consequence of Lemma 11.

Algorithm *ComputeLongestValid*(C, τ)
1. $i \leftarrow n$;
2. Initialize empty $\mathcal{D}_C$;
3. **for** $j \leftarrow n$ **to** 0
4. **do while** $i \geq 0 \wedge \mathcal{D}_C.SegmentIsValid()$
5. **do** $i \leftarrow i - 1$;
6. $\mathcal{D}_C.Extend(i)$;
7. $LV_j \leftarrow i + 1$
8. $j \leftarrow j - 1$;
9. $\mathcal{D}_C.Shorten(j)$;

The algorithm *ComputeLongestValid* computes LV_j by moving two pointers i and j backwards over all n points of the trajectory τ. Both pointers start at the last point of the trajectory. Pointer i is moved backwards until the segment $\tau[i,j]$ is not valid. At that moment we can conclude that LV_j is equal to $i+1$. Then the pointer j is moved one step and the next LV_j is determined in a similar fashion. Note that it is not necessary to reset pointer i to j, because C is decreasing monotone.

Testing whether $\tau[i,j]$ satisfies a criterion is not a straightforward task. Therefore the data structure $\mathcal{D}_C$ is included in the algorithm to keep track of the validity of candidate segment $\tau[i,j]$. The actual form of this data structure depends on the kind of decreasing monotone criterion C that is considered. It allows for three operations. First of all it can be queried for the validity of $\tau[i,j]$ using the *SegmentIsValid* function. Secondly the segment $\tau[i,j]$ can be extended by one point at the start, when i is decreased by 1. Thirdly the segment $\tau[i,j]$ can be shortened by one point at the end, when j is decreased by 1.

The algorithm *ComputeLongestValid* consists of at most $O(n)$ steps in which the interval $\tau[i,j]$ is extended or shortened. The running time of the algorithm depends on the precise data structure that is used for $\mathcal{D}_C$. Let $E(n)$, $S(n)$ and $V(n)$ denote the running times of respectively the *Extend*, *Shorten* and *SegmentIsValid* operations. The following theorem follows directly.

Theorem 12 *The procedure ComputeLongestValid computes for every trajectory index j the smallest index i for which $\tau[i,j]$ satisfies the decreasing monotone criterion C in $O(n(E(n)+S(n)+V(n)))$ time, where n is the number of points on the trajectory, and E, S, and T are the running times of respectively the* Extend, Shorten *and* SegmentIsValid *operations on the data structure $\mathcal{D}_C$ as described above.*

The following sections list some basic decreasing monotone criteria and the data structure $\mathcal{D}_C$ that is used for computing the corresponding compressed start-stop matrices.

2.4.1 Range criterion on attribute

There is a large class of criteria of the form "for all points in the segment attribute a should be within a certain range of size α". Alternatively such a criterion can be seen as an upper bound of α on the difference between the maximal and the minimal value of attribute a over all points in the segment.

The data structure $\mathcal{D}_C$ keeps track of the this minimal and maximal element. For monotone attributes, such as duration, traveled distance and number of points on the segment, this is easy, because they are monotone. They increase along the trajectory. The minimal and maximal element of the segment hence correspond to respectively the first and last element. Keeping track of those elements when extending or shortening the candidate segment takes constant time. We test the validity of the current candidate segment by comparing the difference between minimal and maximal element with α in constant time.

For non-monotone attributes, such as speed, data structure $\mathcal{D}_C$ is slightly more complicated. We use an ordered multiset data structure such as a balanced binary search tree [9] to keep track of all attribute values of the current candidate segment. Extending and shortening the candidate segment correspond to respectively inserting and deleting an attribute value. Testing whether $\tau[i,j]$ is valid consists of a query for the maximal and minimal element in the multiset and comparing their difference with α. Using a balanced binary search tree all three operations take $O(\log n)$ time.

If the value of attribute a of all points on a segment, including both endpoints, are required to lie within a range of α, situations can arise in which no segmentation is possible. For instance, when the difference in a between consecutive points $\tau(i)$ and $\tau(i+1)$ is larger than α. One could argue that it would be better to allow segmentations with a splitting point at $\tau(i)$ or $\tau(i+1)$ than to forbid all segmentations. This is equivalent to ignoring either the first or the last point of a segment in the computation of the minimal size range.

The algorithm *ComputeLongestValid* requires only a small change to make this possible. If the starting point of a segment is excluded we simply need to decrease all LV_j values by one. If the stopping point of a segment is excluded, all LV_j need to be shifted one step, that is: $LV_j \leftarrow LV_{j-1}$.

In practice, it can be very useful to allow for a constant number c of outliers that do not need to lie within the range of size α. This does not affect decreasing monotonicity. A similar data structure is used as before. We consider the $c+1$ canonical (ending at attribute values that are present) ranges that cover all values except for c outliers. See for example Figure 2.6, where two outliers are allowed. Finding those ranges takes $O(c \log n)$ time. If the smallest range is less than α the candidate segment is valid.

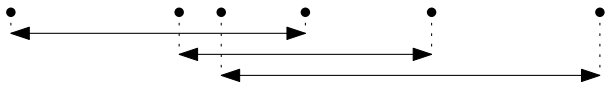


Figure 2.6: An ordered set of values and its corresponding $c+1$ canonical ranges that cover all but c elements ($c = 2$).

2.4.2 Lower bound / Upper bound on attribute

Another class of criteria is of the form "for all points in the segment attribute a should be $\geq \gamma$ (or $\leq \gamma$)". These criteria are especially useful in compound criteria. To maintain the validity of the candidate segment $\tau[i, j]$ we simply keep track of the number of elements that have $a < \gamma$. Adding and removing a point respectively to or from $\tau[i, j]$ take $O(1)$ time. Testing validity consists of testing whether the number of points with $a < \gamma$ is zero, which takes constant time as well.

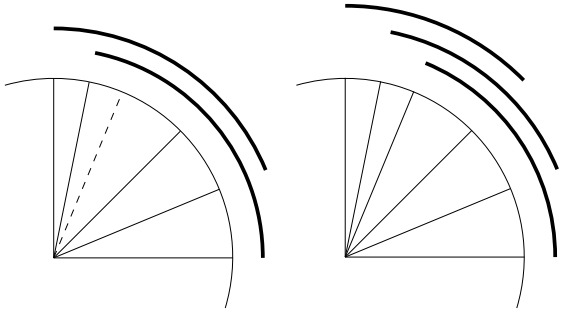
This approach can easily be extended to handle a constant number c of outliers. In that case only the testing procedure changes. The number of points with $a < \gamma$ has to be compared to c instead of zero. The running times remain the same, and are independent of c.

2.4.3 Angular range criterion on attribute

The range criterion for *angular* attributes, such as heading and turning angle, is similar to the range criterion for ordinary attributes. The only difference is that the value range is wrapped around, or stated differently: the attribute values are computed modulo 2π. For instance, the heading values $\pi/6$ and $11\pi/6$ can be covered by a range of size $\pi/6 - 11\pi/6 \mathrm{mod} 2\pi = \pi/3$. If the upper bound on the angular range α is less than π, the approach is similar to the ordinary range criterion. We maintain a minimal and maximal element that differ less than π (modulo 2π) and span all values in between.

However, if the upper bound α is larger than π the circular nature of the angular domain prevents us from maintaining a meaningful maximal and minimal element. In that case we keep track of the largest gap (empty interval) between consecutive angular values instead. The smallest angular range that can cover all the attribute values has size $360° - g$, where $g \in [0, 2\pi)$ is the angle of this largest gap. To keep track of this largest gap we use two ordered multiset structures that both store the set of gaps. The sets are ordered by respectively size and angular order. Testing validity corresponds to a query for the largest gap g and comparing $360° - g$ with α. Extending and shortening correspond to respectively splitting and merging a gap, both taking $O(\log n)$ time using the set structures.

The data structure can be extended to allow for a constant number c of outliers. Instead of storing gap-intervals that span exactly two values, we store intervals that span exactly $2 + c$ values. Testing validity remains unchanged. However, the extend and shorten operations have to change. Inserting a value changes c intervals and create one new interval. Deleting a value changes c intervals and delete one. Both operations take $O(c \log n)$ time. An example is given in Figure 2.7 for two outliers.



(a) Before insertion (b) After insertion

Figure 2.7: Inserting the dashed value in the angular range data structure, which allows two outliers. Only the changed intervals are shown.

2.4.4 Disk criterion

A disk criterion has the form "all points in the segment can be covered by a circle with fixed radius r". Data structure $\mathcal{D}_C$ keeps track of the *smallest enclosing circle* c_{enc} of the points on the segment. The query corresponds to comparing the radius of the smallest enclosing circle to r.

It is not straightforward to maintain the smallest enclosing circle efficiently in a dynamic setting. The asymptotically fastest technique lifts the problem to convex programming over a halfspace intersection [8]. Insertions, deletions and smallest enclosing circle query are guaranteed to take only polylogarithmic time. Using this data structure the algorithm *ComputeLongestValid* has a near-linear running time.

However, the complex data structure described in [8] has high constant factors in the running time. We therefore maintain an *approximate* smallest enclosing circle using our method described below. Using an approximate smallest enclosing circle instead of an exact algorithm can cause differences in the segmentation, but the differences (if any are present) are mostly insignificant. Note that the approximation ratio can be chosen arbitrary close to one.

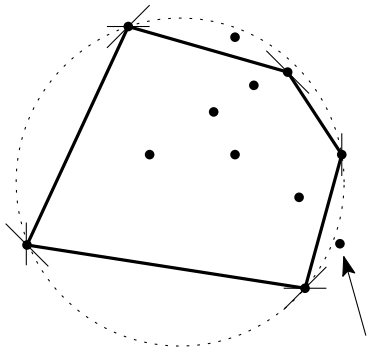


Figure 2.8: The 4-approximate convex hull of a point set and its smallest enclosing circle.

The idea of the approximation scheme is to maintain a constant size approximate convex hull and compute the radius r' of its smallest enclosing circle c'_{enc} when the structure is queried. The approximate convex hull consists of the (at most $2k$) extreme points in k regularly sampled directions. We call this the k-approximate convex hull. An example is shown in Figure 2.8. Thin lines are drawn through the points on the hull to indicate the directions in which the points are extreme. In Figure 2.8 the approximate enclosing circle is not equal to the smallest enclosing circle, since there is a point that is not enclosed by the approximate enclosing circle. This point is indicated by an arrow. However, Theorem 13 states that the two circles never differ a lot.

Theorem 13 *Given a point set P. Let r and r' be the radii of the smallest enclosing circles of respectively P and its k-approximate convex hull. Then,*

$$1 \geq \frac{r'}{r} \geq 1 - \frac{1}{2\cot(\frac{\pi}{2k})} = 1 - O\left(\frac{1}{k}\right).$$

Proof. The upper bound on r'/r is not hard to prove. The radius r' is at most r, because the set of points that c'_{enc} encloses is a subset of the set of points that c_{enc} encloses. Hence $1 \geq r'/r$.

The lower bound requires some more insight. Let points p and q be consecutive points on the k-approximate convex hull of P. The situation is depicted in Figure 2.9. The lines ℓ_p and ℓ_q define two of the $2k$ regularly oriented empty half-planes of the k-approximate convex hull. All points reside inside the (lower) half-planes defined by ℓ_p and ℓ_q, because p and q are extreme points.

Consider a point s which lies on c_{enc}, but not inside c'_{enc}. Point s has to lie outside the k-approximate convex hull. Without loss of generality we assume that s resides in the triangle ΔpqI, with I the intersection point of ℓ_p and ℓ_q. The distance between s and $\overline{pq}$ is an upper bound on the difference between r and r'. Note that the distance from point s to the line segment $\overline{pq}$ is at most the distance between I and $\overline{pq}$.

The angle α between ℓ_p and ℓ_q is fixed and is equal to $\pi\left(1 - \frac{1}{k}\right)$. Thus, given p, q and α the point I is located on a circle arc that goes through p and q. Hence the distance between I and $\overline{pq}$ is maximal when it is located at the bisector of $\overline{pq}$. In that case the distance equals $\frac{\|\overline{pq}\|}{2\tan(\alpha/2)} \geq r - r'$. Combining this with the fact that $\|\overline{pq}\|$ is a lower of r yields

$$r' \geq r - \frac{\|\overline{pq}\|}{2\tan(\alpha/2)} \geq r\left(1 - \frac{1}{2\tan(\alpha/2)}\right).$$

Note that $\tan(\alpha/2) = \tan(\frac{\pi}{2}\left(1 - \frac{1}{k}\right)) = \cot(\frac{\pi}{2k})O(k)$. Substitution yields the result stated in the theorem. □

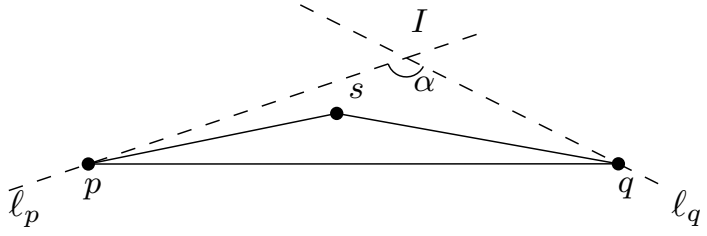


Figure 2.9: The distance between I and $\overline{pq}$ is at most $\frac{\|\overline{pq}\|}{2\tan(\alpha/2)}$.

To maintain the k-approximate convex hull the pointset is stored k times, each point set ordered in a different regularly sampled direction. Using an efficient set structure [9], insertion and deletion take $O(\log n)$ time per set, so $O(k \log n)$ time in total. Getting the k-approximate convex hull also takes $O(k \log n)$ time. We compute the smallest enclosing disk for this set of points with a randomized incremental algorithm in $O(k)$ time [10].

2.4.5 A fraction of outliers instead of a constant number

In the previous sections we discussed how to allow a constant number of outliers per segment in a decreasing or increasing monotone criterion. Allowing a *fraction* of points per segment to be outlier is more useful in practice. Changing the constant number of outliers to a fraction results in a criterion that is no longer increasing monotone [2]. In fact, it can even be non-stable.

Theorem 14 *Let C be a decreasing or increasing monotone criterion. Allowing a number of outliers proportional to the number of points in a segment can make the criterion non-stable.*

Proof. Consider a trajectory with a certain attribute a that has alternating values: $a(0) = 1, a(1) = -1, a(2) = 1$ and so on. Assume that we want to segment according to a range criterion on attribute a with range 1. Allowing a fraction of $1/2 - \varepsilon$ outliers (with small enough $\varepsilon > 0$) yields a completely alternating validity of segments. That is, for any starting point i and any end point i' we have that $\tau[i, i']$ is valid (or invalid), $\tau[i, i' + 1]$ is invalid (or valid), and so on. Hence the compressed start-stop matrix would have $\Omega(n^2)$ blocks and the criterion would be non-stable. □

However, we can approximate the fraction of outliers criterion. The idea behind the approximation is as follows. The fraction of outliers criterion of the form "C except for a fraction f of the points" is equivalent to:

$$\begin{aligned} & C \quad \vee \\ ((C \text{ except for 1 outlier }) \wedge (\text{ number of points on segment } & \geq 1/f)) \quad \vee \\ ((C \text{ except for 2 outliers }) \wedge (\text{ number of points on segment } & \geq 2/f)) \quad \vee \\ & \ldots \quad \vee \\ ((C \text{ except for } nf \text{ outliers }) \wedge (\text{ number of points on segment } & \geq n)) \end{aligned}$$

This compound criterion has $O(n)$ clauses of stable criteria, so it is not necessarily a stable criterion. However, in practice this combination of criterion can already have acceptable performance, since f is usually very small. Furthermore, we can leave out some of the last clauses if we assume a maximum segment length.

We can also properly approximate the criterion by using less terms and allowing some slack in the number of outliers. We can formulate an approximation criterion such that the fraction of allowed outliers for any segment is between $(1 - \varepsilon)f$ and $(1 + \varepsilon)f$. The approximation criterion consists of several clauses similar to the exact criterion above. The first clause C remains unchanged. For the second clause we pick

$$(C \text{ except for 1 outlier }) \wedge \left(\text{ number of points on segment } \geq \frac{1}{(1+\varepsilon)f} \right).$$

The extra slack in the allowed number of outliers enables us to use larger minimum length in the third clause. It is easy to see that the first two clauses correctly bound the number of outliers for all segments with number of points $\leq \frac{1}{(1-\varepsilon)f}$. Hence the third clause is:

$$\left(C \text{ except for } \frac{1+\varepsilon}{1-\varepsilon} \text{ outliers } \right) \wedge \left(\text{ number of points on segment } \geq \frac{1}{(1-\varepsilon)f} \right).$$

The next clauses are chosen in a similar way. The $(i-2)$-th clause is of the form:

$$\left(C \text{ except for } \left(\frac{1+\varepsilon}{1-\varepsilon}\right)^i \text{ outliers } \right) \wedge \left(\text{ number of points on segment } \geq \left(\frac{1+\varepsilon}{1-\varepsilon}\right)^i \frac{1}{(1+\varepsilon)f} \right).$$

To cover all segment lengths, the last clause (the i_{max}-th clause) should satisfy

$$\left(\frac{1+\varepsilon}{1-\varepsilon}\right)^{i_{max}} \frac{1}{(1+\varepsilon)f} \geq n.$$

Hence the total number of clauses i_{max} is bounded as follows

$$i_{max} = O\left(\log_{\left(\frac{1+\varepsilon}{1-\varepsilon}\right)}(nf(1+\varepsilon))\right) = O\left(\log_{\left(\frac{1+\varepsilon}{1-\varepsilon}\right)} n\right).$$

This implies that the total number of blocks in the compressed start-stop matrix is $O\left(2^{(\log n)/(\log((1+\varepsilon)/(1-\varepsilon)))}\right)$. For fixed ε this means that the criterion is $O(n \log n)$-stable. The result is summarized in the following theorem.

Theorem 15 *Let C be a decreasing or increasing monotone criterion, which allows for ignoring a constant number of outliers. The fraction of outliers criterion can be approximated with a $2^{(\log n)/(\log((1+\varepsilon)/(1-\varepsilon)))}$-stable criterion, where n is the number of points on the trajectory. This approximation guarantees that the fraction of allowed outliers per segment is between $(1-\varepsilon)f$ and $(1+\varepsilon)f$.*

2.5 Computing the optimal segmentation

Before presenting our algorithm on compressed start-stop matrices, we discuss a simple algorithm that computes the optimal segmentation given an uncompressed start-stop matrix [2]. This dynamic programming algorithm is based on the following property:

Observation 1 *The optimal segmentation for $\tau[0,i]$ (if it exists) either consists of just one segment, or it is equal to an optimal sequence of segments for $\tau[0,j]$ appended with a segment $\tau[j,i]$, where j is an index such $\tau[j,i]$ is valid.*

Observation 1 allows us to transform the segmentation problem to a shortest path problem in an unweighted directed acyclic graph on n vertices, having the start-stop matrix as adjacency matrix. The Dynamic Program *SimpleComputeSegmentation* finds this shortest path from 0 to n. It computes the optimal segmentation Opt_i of $\tau[0,i]$ incrementally for $i = 0, \ldots, n$. Instead of storing the complete segmentation for each i, only the starting index of the last segment (*last*) and the total number of segments (*count*) is stored. The actual segmentation can be retrieved from the DP-table in $O(n)$ time.

Algorithm *SimpleComputeSegmentation*(τ, C)
1. $Opt_0.last \leftarrow nil$; $Opt_0.count \leftarrow 0$;
2. **for** $i \leftarrow 1$ **to** n
3. **do** $Opt_i.count \leftarrow \infty$
4. **for** each j for which $\tau[j,i]$ satisfies C
5. **do if** $Opt_j.count + 1 < Opt_i.count$
6. **then** $Opt_i.count \leftarrow Opt_j.count + 1$;
7. $Opt_i.last \leftarrow j$;

For each index i the algorithm loops over all valid candidate segments $\tau[j,i]$, while maintaining the optimum. The optimal segment in this context is the segment $\tau[j,i]$ for which Opt_j consists of the smallest number of segments.

Our approach is similar to *SimpleComputeSegmentation*: The algorithm *ComputeSegmentation* loops over all indices i and for each i it determines the optimal last segment. Our algorithm finds this optimal segment more efficiently. Instead of processing the valid starting indices one by one, a whole block of consecutive valid indices is processed at once. These blocks correspond to the blocks in the compressed start-stop matrix $\mathcal{S}$ at row i. To allow for an operation that efficiently processes a block of valid indices, *Opt* is stored in a balanced binary tree $\mathcal{T}$ instead of an array. A node in $\mathcal{T}$ corresponds to an entry in *Opt* and has three fields: *last*, *count* and *index*. The tree $\mathcal{T}$ is ordered on *index*.

Algorithm *ComputeSegmentation*$(\tau, \mathcal{S})$
1. Initialize empty $\mathcal{T}$;
2. Create new node ν_0;
3. $\nu_0.index \leftarrow 0$; $\nu_0.count \leftarrow 0$;
4. $\mathcal{T}.Insert(\nu_0)$
5. **for** $i \leftarrow 1$ **to** n
6. **do** Create new node ν;
7. $\nu.index \leftarrow i$;
8. $\nu.count \leftarrow \infty$;
9. **for** each block b at row i of $\mathcal{S}$
10. **do** $\nu' \leftarrow \mathcal{T}.GetMinimalCount(b)$;
11. **if** $\nu'.count + 1 < \nu.count$
12. **then** $\nu.count \leftarrow \nu'.count + 1$;
13. $\nu.last \leftarrow \nu'.index$;
14. $\mathcal{T}.Insert(\nu)$;

For each row i in the compressed start-stop matrix a new element storing the optimal segmentation for $\tau[0, i]$ is created and inserted in $\mathcal{T}$. Furthermore, the query algorithm *GetMinimalCount* is executed B_i times while maintaining the optimal starting index for the last segment, where B_i is the number of blocks at row i.

Algorithm *RetrieveSegmentation*$(\mathcal{T}, i)$
1. $\nu \leftarrow \mathcal{T}.Find(i)$;
2. $k \leftarrow \nu.count$;
3. **while** $k > 0$
4. **do** Output $[s_{k-1}, s_k] = [\nu.last, \nu.index]$;
5. $k \leftarrow k - 1$;
6. $\nu \leftarrow \mathcal{T}.Find(\nu.last)$;

When the tree $\mathcal{T}$ is computed the actual segmentation for $\tau[0, n]$ is retrieved using the algorithm *RetrieveSegmentation*. This takes $O(n \log n)$ time using the standard find operation for binary search trees. It is also possible to maintain cross pointers between nodes. That is, a node ν stores a pointer to the node ν' for which $\nu.last = \nu'.index$. Maintaining those cross pointers yields no increase in asymptotic running time. Using those pointers, the find operations take only constant time each, which results in a running time of $O(n)$ for retrieving the segmentation from $\mathcal{T}$.

The *GetMinimalCount* query is the crucial part of our algorithm. Its input is a block b at row i of the compressed start-stop matrix. Assume that block b covers the indices $b.begin, b.begin + 1, \ldots, b.end$. These indices are starting indices of valid candidate segments, which end at i.

The goal of the *GetMinimalCount* procedure is to find the candidate segment starting at $j \in [b.begin, b.end]$ for which Opt_j consists of the smallest number of segments. This corresponds to finding the node in $\mathcal{T}$ with minimal *count* over all nodes with $index \in [b.begin, b.end]$.

To find this node one could simply visit all nodes with $index \in [b.begin, b.end]$ and keep track of the one with minimal *count* in $O(n)$ time, but this can be done more efficiently. According to the binary search tree property the nodes with $index \in [b.begin, b.end]$ are located between the paths to $b.begin$ and $b.end$. Figure 2.10 shows an example.

A node ν is *between* the paths to $b.begin$ and $b.end$ if and only if it is in one of the following three sets

1. on the path to $b.begin$, with $\nu.index \geq b.begin$, or
2. on the path to $b.end$, with $\nu.index \leq b.end$, or
3. in the right or left subtree of a node in respectively set 1 or 2.

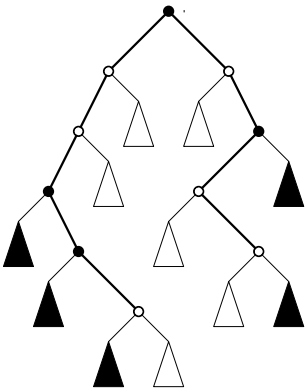


Figure 2.10: A tree $\mathcal{T}$ and two search paths. The nodes in the white subtrees and the white nodes on the paths are between the two search paths.

There are at most $O(\log n)$ nodes in set 1 and 2, and there can be a linear number of nodes in set 3. However, the size bound on sets 1 and 2 implies that the number of subtrees that define set 3 is at most $O(\log n)$. To speed up the search we augment [9] the balanced binary search tree $\mathcal{T}$. Each node ν is augmented with the fields $\nu.min_{count}$ and $\nu.argmin_{count}$, which are equal to respectively the minimal value of *count* over all nodes in the subtree rooted at ν and a pointer to the node where this minimum occurs. This enables us to find the node with minimal *count* in a subtree (given its root) in constant time.

Algorithm *GetMinimalCount*(b)
1. $\nu_{split} \leftarrow$ node where search paths to $b.begin$ and $b.end$ split up
2. $min_b \leftarrow \nu_{split}.count$;
3. **for** each node ν' on path from ν_{split} to $b.begin$
4. **do if** ν' is left child of its parent
5. **then** $min_b \leftarrow \min(min_b, \nu'.right.min_{count}, \nu'.count)$;
6. **for** each node ν' on path from ν_{split} to $b.end$
7. **do if** ν' is right child of its parent
8. **then** $min_b \leftarrow \min(min_b, \nu'.left.min_{count}, \nu'.count)$;
9. **return** node where min_b occurs;

The procedure *GetMinimalCount* iterates over all $O(\log n)$ nodes on the search paths to $b.begin$ and $b.end$, while maintaining the node with minimal *count* in between. If a node of set 1 or 2 is encountered the optimum is updated according to that node and according to that nodes child in set 3. We conclude the following running time.

Lemma 16 *The procedure GetMinimalCount finds the candidate segment starting at* $j \in [b.begin, b.end]$ *for which* Opt_j *consists of the smallest number of segments* $O(\log n)$ *time.*

Theorem 17 *Given a trajectory τ and a compressed start-stop matrix $\mathcal{S}$ of a λ-stable criterion, the algorithm ComputeSegmentation computes the optimal segmentation in $O((n+\lambda)\log n)$ time, where n is the number of points on the trajectory.*

Proof. We showed that running time is dominated by the λ calls to *GetMinimalCount*. Combining this with Lemma 16 yields the theorem. □

2.6 Segmentation by movement states

The most natural way to define a criterion for segmentation according to multiple behavioral movement states is a disjunction of subcriteria, of which each subcriterion corresponds to a behavioral state:

$$\text{Behavior } 1 \vee \text{Behavior } 2 \vee \cdots \vee \text{Behavior } m.$$

As was described in Section 2.3.3, to segment according to such a disjunction we take the union of the compressed start-stop matrices. However, taking the union of the compressed start-stop matrices makes us lose valuable information: the segmentation algorithm cannot distinguish between different classes of segments anymore.

The algorithm *ComputeSegmentation* can be extended in such a way that this valuable information remains. In this extended algorithm the input consists of the m compressed start-stop matrices that correspond to the m behavioral states. The loop of lines 9-13 over all blocks of the compressed start-stop matrix is executed once for each of the m start-stop matrices. Given the m individual start-stop matrices, classification of the segments is possible. To store the classification each node in $\mathcal{T}$ has an extra field storing the movement state of the last segment. When the optimal starting point of the last segment is changed on line 13 the movement state corresponding to the current compressed start-stop matrix is assigned to this field. The running time of this segmentation/classification algorithm is $O(mn\log n)$.

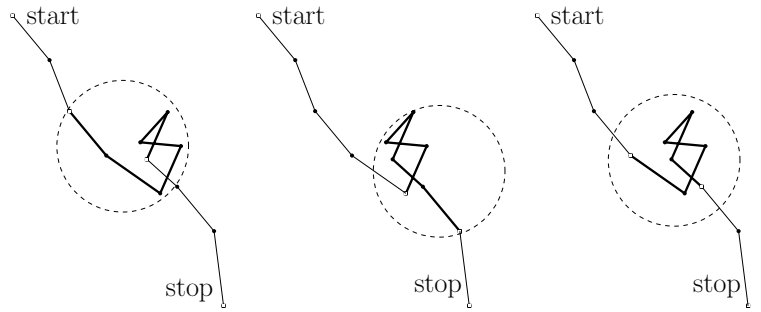
2.6.1 Adding optimization goals in case of ties

Note that a segment is put into the first class, whose criterion it satisfies. Intuitively, the order in which the start-stop matrices are handled corresponds to the order of "preference". However, this order does not yield real guarantees. One could also order all optimal segmentations (equal segment count) by the number of segments of class i that they contain (and in case of ties, on the number of segments of class i', etc). Keeping track of the number of segments of each class and doing a lexicographical comparison instead of the simple comparison of line 11 takes only $O(m)$ additional time, which does not affect the asymptotic running time.

There is another way of breaking ties that is especially practical when segmenting trajectories of animals that show *pausing* or *stopping* behavior (see Section 2.8 for an example). This kind of behavior can usually be described by a disk criterion. Segmenting without defining extra rules for breaking ties, can yield strange results biased towards a certain direction (forward or backward).

For example, consider the trajectory of Figure 2.11. Assume that it is segmented according to a disjunction criterion containing a disk and bounded turning angle criterion ($\leq 60°$). Making the last segment as long as possible would result in the segmentation of Figure 2.11(a). Making the last segment as short as possible results in the situation of Figure 2.11(b). Both are not what we would expect from a good segmentation. They are biased towards respectively the front or the back: the segment that satisfies the disk criterion ends in a strange "limb" at its front- or back-end.

To counteract this biased behavior, rules can be defined to break ties based on the segment classes. In the example of Figure 2.11, one should pick the last segment as short as possible if it satisfies the bounded turning angle criterion, and pick the last segment as long as possible if it satisfies the disk criterion. This strategy results in the segmentation in Figure 2.11(c), which is preferred over the other two segmentations.



(a) Last segment as long as possible (b) Last segment as short as possible (c) Class dependent tie breaking

Figure 2.11: Three optimal segmentations. Pausing segment is depicted thicker.

The tie breaking strategy is easy to incorporate in the algorithm. Simply pick the left-most or right-most valid starting index in the *GetMinimalCount* procedure, depending on the current movement state, instead of an arbitrary one. Hence two variants of *GetMinimalCount* are needed: one to find the left-most and another to find the right-most valid optimal starting index within the block. To enable those two queries on $\mathcal{T}$, each node stores two $argmin_{count}$ fields, one storing the left-most and another one storing the right most-node where min_{count} occurs. Those changes do not affect the asymptotic running time.

2.6.2 Follow relations between movement states

In practice, transitions between movement states are not occurring at random, they are often well-structured. The transitions can be modeled by a connected graph on m vertices, which we call a *transition graph*. The rule "movement state A can be followed by movement state B" corresponds to an edge between vertices A and B in the transition graph. We present a method to enforce those transition relations on a segmentation.

This method computes m segmentation trees $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$, instead of one (as described in Section 2.5). Each tree $\mathcal{T}_s$ stores the optimal segmentation for the subtrajectory $\tau[0, i]$ that ends in movement state s for all $i = 1, \ldots, n$.

Consider the computation of the optimal segmentation for $\tau[0, i+1]$ ending at state s. Assume that s is only preceded by states $p_1, p_2, \ldots, p_d$ according to the transition graph. As before, the start-stop matrix of state s determines which segments (ending at $i + 1$) are valid. Recall that in the unrestricted setting the starting index of the last segment is chosen such that it starts at the index j for which the optimal segmentation of $\tau[0, j]$ has minimal segment count and

for which $\tau[j, i]$ is valid, which is computed efficiently using *GetMinimalCount* queries. In the restricted case this definition of best segment changes: the last segment starts at the index j for which the optimal segmentation of $\tau[0, j]$ ending at p_1, or p_2, or $\ldots$, has minimal segment count and for which $\tau[j, i]$ is valid according to the start-stop matrix of state s. To compute index j we execute the *GetMinimalCount* query on $\mathcal{T}_{p_1}, \mathcal{T}_{p_2}, \ldots, \mathcal{T}_{p_d}$. The rest of the algorithm is similar. The following theorem follows directly.

Theorem 18 *Given a trajectory τ consisting of n points, a movement state transition graph $G(V, E)$ with $|V| = m$ and compressed start-stop matrices $\mathcal{S}_\infty, \mathcal{S}_\infty, \ldots, \mathcal{S}_\Updownarrow$, each of a λ_i-stable criterion with $\lambda_i \leq \lambda$, the algorithm ComputeSegmentation computes a segmentation with minimal number of segments, satisfying the transition graph G of which each segment corresponding to movement state s satisfies the criterion of $\mathcal{S}_\int$ in $O(|E|(n + \lambda) \log n)$ time.*

Enforcing hard follow restrictions can result in situations where no valid segmentation exists. An alternative approach is to model the transitions with a weighted transition graph, which penalizes less likely transitions. In this setting the segmentation goal is to minimize the total penalty of the segmentation. This can be achieved by an algorithm very similar to the one of Theorem 18, but instead of incrementing the segment count by one, when appending a new segment, we increment the penalty by a value depending on the transition.

One could for instance base the penalties on the probabilities of a Markov chain that describes the movement state transitions. Going from one state to another could be penalized by the logarithm of its probability in the Markov chain. This implies that the optimal segmentation has the highest probability of occurring according to the Markov chain.

2.7 Interactive parameter selection

Most criteria that we have discussed involve parameters, such as the upper bound α on the angular range of the heading and the radius r of the covering disk. Tuning these parameters is complex, and in most applications there is no well-defined ground truth. Hence, an interactive setting is needed. This interactive process can be guided by the stability of the parameter values. A value is *unstable* if a small change to its value results in a large change in the segmentation, i.e., in a change of the number of segments. To measure the stability of a parameter value we run the segmentation algorithm multiple times with different parameter values and count the number of segments for each of the resulting segmentations, and sample the function which maps the parameter domain to the number of segments in the corresponding segmentation. The stabler value ranges correspond to the "flat" parts of this function, that is, the parts with the least variation in number of segments (see Section 2.8 for an example).

To compute the stability of parameter values, we need to compute the segmentation of the same trajectory multiple times using the same criterion, but with different parameter values. For decreasing monotone criteria, information can be reused in the different runs. For this purpose we use the double-and-search method described by Buchin *et al.* [6], but instead of testing validity for segments directly, we query a data structure that is constructed once in $O(n \log n)$ and used in all runs with different parameter values. The running time of the double-and-search method is reduced from $O(n \log n)$ to $O(k \log n)$ time, where k is the number of segments.

The data structure is basically a table that stores certain criterion-dependent information for all segments of length $1, 2, 4, 8, \ldots$. For instance, for the range criterion on speed, the maximal and minimal speed is stored, and for the (approximate) disk criterion the extreme points in all k directions are stored. The data structure can efficiently (in $O(\log n)$ time) test the validity of any segment given a certain parameter value. In case of the range criterion it can check whether

a certain segment is valid by retrieving the maximum and minimum value from the tables and testing whether their difference is small enough. A naive use of the retrieve query, i.e., in each step of the double-and-search method, yields a running time of $O(k \log^2 n)$. However, by reusing information during the search, we can reduce this running time to $O(k \log n)$.

2.8 Case study

We assess the performance of our framework by analyzing two trajectories of migrating white-fronted geese [18] during their spring migration. The goal is to segment this data set into migration flight and stopovers (including wintering, breeding, and moult). Figure 2.12 shows the segmentation as computed by our framework.

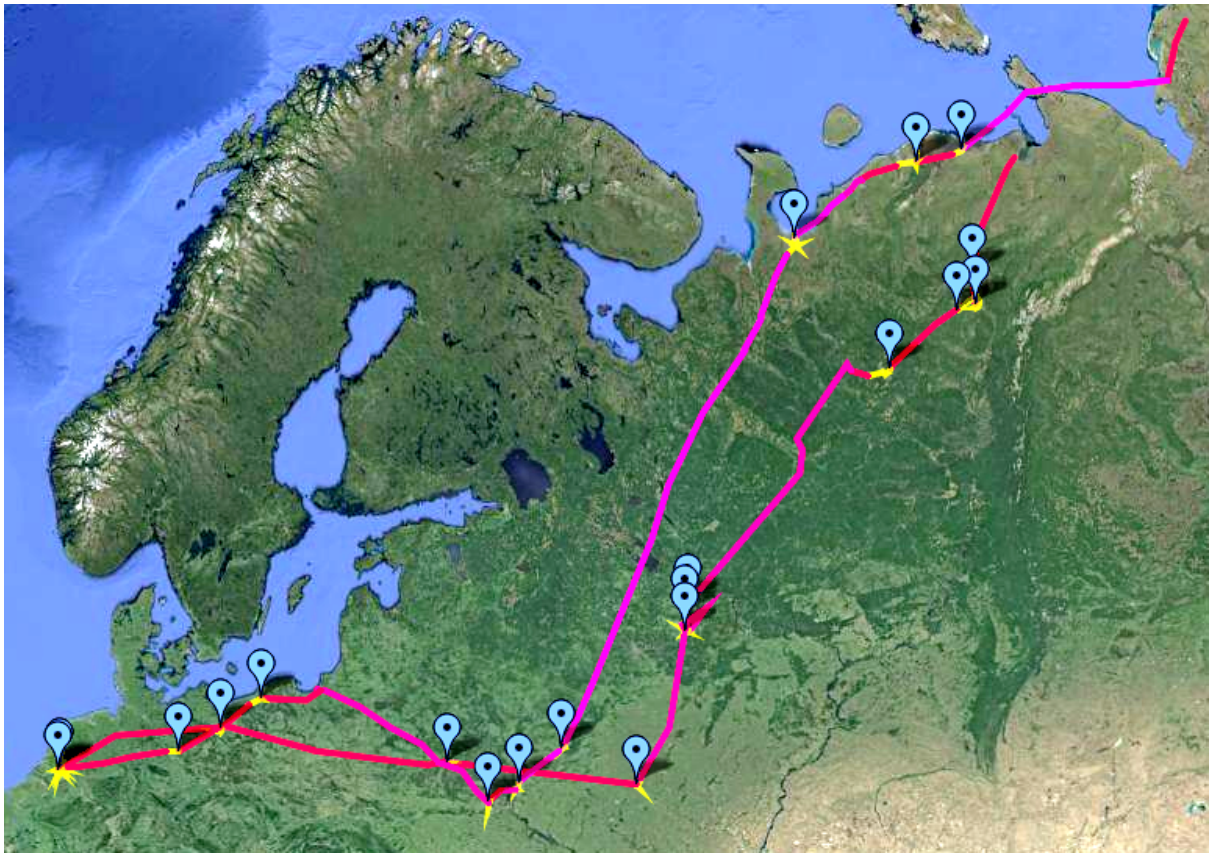


Figure 2.12: Trajectories of two migrating geese. Red/pink segments are flight, yellow segments are stopovers. Blue markers indicate end of a stopover. The red to pink color map is used to indicate the absolute number of outliers.

The data was previously analyzed and manually segmented by domain experts [18]. A stopover segment is characterized by its limited variation in location. Therefore, the disk criterion is used for stopovers. Note that a stopover is not simply a stop, but a resting place, where a goose rests, flies (short stretches) and feeds for at least 48 hours [18] before moving on. During flight, geese maintain the same heading for long stretches. This clearly differs from the seemingly undirected motion that can be observed at stopovers. Hence we use the angular range criterion for heading for flight. On this data set, a disk with radius 30 km and an angular range of 1.7 radians yield the best segmentation result, that is, the segmentation that is most similar to the domain-expert's segmentation by hand.

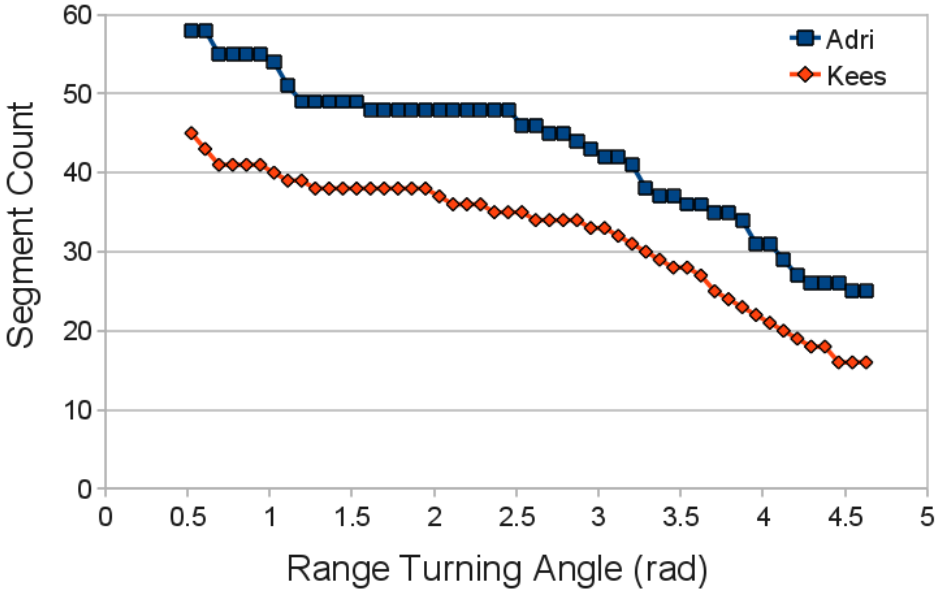


Figure 2.13: Stability diagram of the turning angle for two geese data sets: Adri and Kees.

Finding suitable parameters was an interactive process. Our search was aided by stability diagrams (see Section 2.7). The stability diagram of Figure 2.13 was very helpful in our search for a good angular range bound. The diagram clearly shows a stable region between 1 and 2.5 radians. We picked three different values in this stable region: 1, 1.7 and 2.5 and segmented according to those parameter settings. There is a tradeoff between large stopovers that can cover parts of flight segments and small stopovers that leave some of the stop points uncovered which are either incorrectly labeled as flight, or which form extra stopovers. The middle value 1.7 proved to be a suitable compromise. A typical example motivating our choice is shown in Figure 2.14.

(a) $\alpha = 1.0$ rad

(b) $\alpha = 1.7$ rad

(c) $\alpha = 2.5$ rad

Figure 2.14: Segmentations for varying heading range bound α.

A similar analysis was done to find a suitable disk radius. The stability diagram indicated several stable regions. We tried one value from each of the most stable regions: 10 km, 30km and 40 km. We preferred 30km, because it resulted in stopovers of size similar to those in the manual segmentation. A radius of 40km mislabeled numerous migration flight segments as stopovers. A radius of 10km was good at detecting stops, but not at finding stopovers. For this 10km radius the manually labeled stopover segments were split in stops (labeled stopover) and non-migration flight (labeled migration flight) (see Figure 2.15).

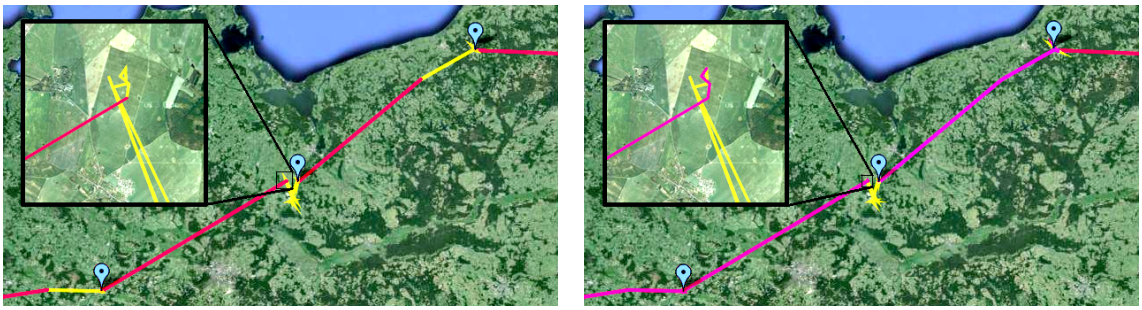
(a) r =30 km (b) r =10 km

Figure 2.15: Segmentations for varying radius r.

The criteria as discussed above are successfully detecting stopovers, but are mislabeling some short stops ($<$ 48h) as stopover. We therefore placed a minimal duration criterion (48h) in conjunction with the disk criterion as suggested by domain knowledge [18]. This resulted in the short stops being labeled migration flight. The geese do not maintain their heading on all data points in the migration flight (according to manual segmentation). Geese can fly in a completely different direction for a very small period of time after which they change back to the previous heading (zigzag). This leads to a segmentation with numerous artificial splits in the stretches of migration flight.

Allowing a constant number of outliers per segment can improve the segmentation [7], since flight segments are then merged. However, a constant number of outliers causes small flight segments to cover a significant part of the stopovers, so many even that complete stopovers can be missed by the algorithm. Instead, we allow a number of outliers proportional to the number of points in the segment (using the method described in Section 2.4.5). In this specific case we have chosen an outlier percentage of 20%. Allowing this percentage of outliers effectively reduces the number of consecutive flight segments. Most of them are merged, which is preferred.

Just optimizing with respect to the number of segments does not uniquely define the splitting points of the segmentation. Thus, we add the rule that flight segments must be as short as possible and stopovers as long as possible using the method described in Section 2.6.1. This tie breaking method performs very well, especially compared to the alternative: making flight as long as possible and stopovers as short as possible. This is illustrated by Figure 2.16.



(a) Long stops, short flight (b) Short stops, long flight

Figure 2.16: Segmentations with different tie breaking rules.

We conclude that our segmentation framework proved to be useful in practice. In contrast to previous approaches [7], our framework offers mechanisms for enforcing a minimum duration, optimizing the splitting points, choosing parameter values, and handling outliers in a more consistent way, which are effective in practice. The resulting segmentations are very close to the manual segmentation. Our labeling agrees with the manual labeling on respectively 96.3 and 92.6% of the total points.

Chapter 3

Segmentation based on the dynamic Brownian Bridge Movement Model

An important issue regarding the criterion-based segmentation framework of Chapter 2 is choosing the right parametrized criteria. That is why we have developed a novel segmentation method that does not rely on specific criteria, but on a statistical movement model: the dynamic Brownian Bridge Movement Model (dBBMM).

First we discuss the dBBMM in Section 3.1. An important task regarding the dBBMM is estimating the diffusion coefficient (a model parameter) along the trajectory. We introduce a new way to describe the diffusion coefficient and relate this to the segmentation problem in Section 3.2. Furthermore, we discuss information criteria and how they are used in our method to formalize the tradeoff between homogeneous segments (measured in likelihood of assigned diffusion coefficients) and number of segments.

In Section 3.3 we present our algorithm, which finds the optimal segmentation with respect to a given information criterion. We also show a way to speed up the algorithm on realistic data using a technique that we call table compression, and provide means to compare different information criteria settings.

Finally, in Section 3.4 we show that our dBBMM-based segmentation method can be combined with the criterion-based methods from Chapter 2.

3.1 Brownian bridge movement model

In this section we discuss the model and its most important properties in the context of segmentation. For a more detailed description of the model see [15].

3.1.1 Brownian bridges

The BBMM describes the movement of an object as a random process. It assumes that the moving object has a Brownian motion. Recall that a Brownian motion can be characterized by a starting location $\boldsymbol{x}$ and a scale parameter σ_m, which is called the *diffusion coefficient* (and σ_m^2 is called the *Brownian motion variance*). If $\boldsymbol{X}_t$ is a Brownian motion with parameters $\boldsymbol{x}$ and σ_m, then

$$\boldsymbol{X}_t = \boldsymbol{x} + \sigma_m \boldsymbol{B}_t,$$

where $\boldsymbol{B}_t$ is a standard Brownian motion. This implies that $\boldsymbol{X}_t \sim \mathcal{N}(\boldsymbol{x}, t\sigma_m)$, with $\mathcal{N}(\mu, \sigma)$ denoting the circular normal distribution with mean μ and covariance matrix $\sigma^2\mathbf{I}$.

Tracking a moving object is done by recording multiple timestamped locations. The BBMM can be used to interpolate the location of the object at a moment in time between the recorded timestamps. More formally, we condition Brownian motion on the location $\boldsymbol{a}$ at time 0, and on the location $\boldsymbol{b}$ at some time T. Such a conditioned Brownian motion is called a *Brownian bridge*. Using the Markov property of Brownian motion, one can show that for $0 \leq t \leq T$,

$$(\boldsymbol{X}_t \mid \boldsymbol{X}_0 = \boldsymbol{a} \wedge \boldsymbol{X}_T = \boldsymbol{b}) \sim \mathcal{N}(\mu(t), \sigma^2(t)),$$

where $\mu(t) = \boldsymbol{a} + \alpha(\boldsymbol{b} - \boldsymbol{a})$ and $\sigma^2(t) = T\alpha(1-\alpha)\sigma_m^2$, with $\alpha = t/T$. Figure 3.1 shows two location distributions at different times.

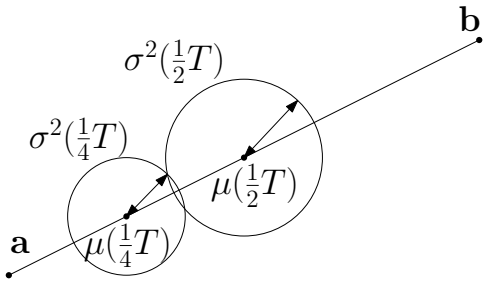


Figure 3.1: A Brownian bridge and the location distributions at times $T/4$ and $T/2$. The distributions at times 0 and T have $\sigma^2 = 0$ and respectively $\mu = \boldsymbol{a}$ and $\boldsymbol{b}$

The BBMM can also handle uncertainties in the location of points $\boldsymbol{a}$ and $\boldsymbol{b}$. If we assume that two points are normally distributed with variances δ_a^2 and δ_b^2 respectively, then we can add $(1-\alpha)^2\delta_a^2 + \alpha^2\delta_b^2$ to $\sigma^2(t)$ to correct for this uncertainty.

Aggregating all the normal probability distributions over time yields the probability density for the fraction of time spent at each location. An example of such a density function is shown in Figure 3.2.

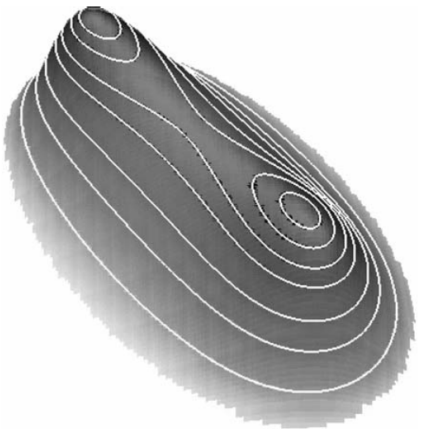


Figure 3.2: Probability density for the fraction of time spent in different regions between two sample points which correspond two the two peaks [15].

3.1.2 Estimating the diffusion coefficient

Horne *et al.* [15] presented a method for estimating the diffusion coefficient using a maximum likelihood method. In this approach, the recorded data points are partitioned in two sets. The even-numbered points are used to define Brownian bridges, and the odd-numbered points are used to estimate the diffusion coefficient and are assumed to be independent. To avoid confusion we renumber and denote the bridge points by $\tau(0), \tau(1), \ldots, \tau(n)$ and their timestamps to $t_0, t_1, \ldots, t_n$. The odd-numbered points are denoted by $\tau^b(0), \tau^b(1), \ldots, \tau^b(n-1)$ and their timestamps by $t_0^b, t_1^b, \ldots, \tau^b(n-1)$. We will see later on that our segmentation algorithm runs on the trajectory consisting of points $\tau(0), \tau(1), \tau(2), \ldots, \tau(n)$, and treats the points $\tau^b(0), \tau^b(1), \ldots, \tau^b(n-1)$ as a trajectory attribute. An example of a trajectory and its Brownian bridges is shown in Figure 3.3.

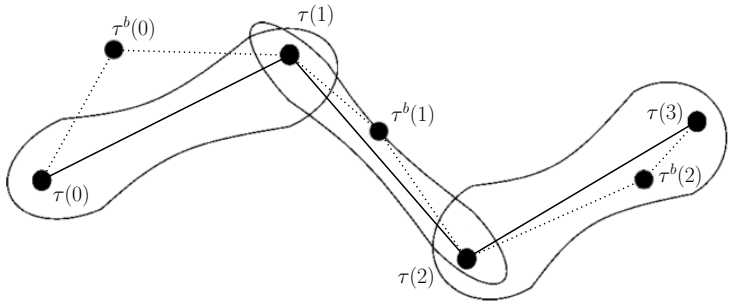


Figure 3.3: A trajectory τ and its Brownian bridges (adapted from [15]).

The likelihood of a candidate diffusion coefficient σ_m^2 given a certain Brownian bridge $\tau[i, i+1]$ (including the point $\tau^b(i)$) is:

$$L(\sigma_m^2 \mid \tau[i, i+1]) = \frac{1}{2\pi\sigma_m^2(i)} \exp\left(\frac{-\|\tau^b(i) - \mu(i)\|}{2\sigma_m^2(i)}\right), \tag{3.1}$$

where $\mu(i) = \tau(i) + \alpha(\tau(i+1) - \tau(i))$ and $\sigma^2(i) = (t_{i+1} - t_i)\alpha(1-\alpha)\sigma_m^2$, with $\alpha = \frac{t_{i+1} - t_i^b}{t_{i+1} - t_i}$.

The likelihood of a candidate diffusion coefficient σ_m^2 given multiple Brownian bridges $\tau[i, i']$ is simply the product of the individual likelihoods:

$$L(\sigma_m^2 \mid \tau[i', i]) = \prod_{j=i'}^{i-1} L(\sigma_m^2 \mid \tau[j, j+1]).$$

To estimate the diffusion coefficient for a whole trajectory Horne *et al.* choose the one that maximizes the likelihood given $\tau[0, n]$. This is equivalent to maximizing the log-likelihood of $\tau[0, n]$. The log-likelihood given a subtrajectory $\tau[i, i']$ is given by:

$$\log\left(L(\sigma_m^2 \mid \tau[i, i'])\right) = \sum_{j=i}^{i'-1} \log\left(L(\sigma_m^2 \mid \tau[j, j+1])\right).$$

Substituting Equation 3.1 yields:

$$\log\left(L(\sigma_m^2 \mid \tau[i, i'])\right) = -\sum_{j=i}^{i'-1} \log\left(2\pi\sigma_m^2(j)\right) + \frac{\|\tau^b(j) - \mu(j)\|}{2\sigma_m^2(j)}. \tag{3.2}$$

3.1.3 Dynamic Brownian Bridge Movement Model

The BBMM assumes that the diffusion coefficient, and therefore the movement behavior is homogeneous along the whole trajectory. This is often not the case in practice. Kranstauber *et al.* [16] introduced the dynamic Brownian Bridge Movement Mmodel (dBBMM) in which the diffusion coefficient may vary along the trajectory to allow for changes in movement behavior.

They presented a sliding window technique which estimates the diffusion coefficient at each of the n bridges. For each window position the bridges are fitted with one σ_m^2 or the window is split in two parts, of which each part is fitted with a separate σ_m^2. After all the window positions are processed, each individual bridge is assigned the average σ_m^2 over all relevant σ_m^2's.

3.2 Using the dBBMM to characterize segments

In Sections 3.1.2 and 3.1.3 we discussed two methods to model the diffusion coefficients along a recorded trajectory. We can model a constant diffusion coefficient, or a diffusion coefficient that can change at each bridge. Allowing a diffusion coefficient that *can* change at each bridge does not necessarily mean that it *has* to change. However, the windowing method by Kranstauber *et al.* [16] results in n different diffusion coefficients on realistic trajectories.

For most trajectories, far fewer diffusion coefficients are needed to model the movement appropriately. Many trajectories consist of a small number of segments, each corresponding to a certain movement state. Hence we can model them using one diffusion coefficient per segment.

We have developed a novel method which, given a trajectory, computes a segmentation labeled with diffusion coefficients. Throughout the next sections the diffusion coefficient of the i-th segment is denoted by $\sigma_{m,i}$. An example is shown in Figure 3.4.

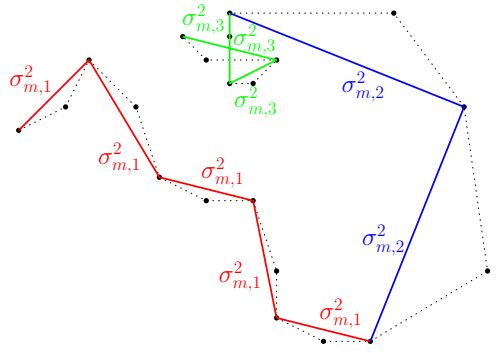


Figure 3.4: Correspondence between estimated diffusion coefficients and the segmentation.

The homogeneity of segments is measured by the likelihood of the estimated diffusion coefficient given the segment. In other words, the homogeneity of the i-th segment (recall that this is $\tau[s_i, s_{i+1}]$) is measured by $L(\sigma_i^2 \mid \tau[s_i, s_{i+1}])$. The homogeneity of the whole segmentation is measured by the products of these likelihoods. Maximizing the overall homogeneity is hence equivalent to maximizing the sum of all the individual loglikelihoods per segment:

$$\log(L) = -\sum_{i=0}^{k-1} \sum_{j=s_i}^{s_{i+1}-1} \log\left(2\pi\sigma_{m,i}^2(j)\right) + \frac{\|\tau^b(j) - \mu(j)\|}{2\sigma_{m,i}^2(j)}. \tag{3.3}$$

We can fix the number of segments and compute the "best" segmentation; that is, the segmentation which maximizes the function of Equation 3.3. This is explained in Section 3.3.1. However, choosing the correct number of segments is not trivial. We can formalize the trade-off between the homogeneity of segments and the number of segments using an information criterion.

3.2.1 Information criterion

An *information criterion* (IC) is a measure which evaluates the quality of a model instance. We allow for information criteria of the following form:

$$-2\log(L) + kp, \tag{3.4}$$

where L is the likelihood of the model instance, k is the number of variables of the model instance, which correspond in our context to respectively the likelihood of the diffusion coefficients given in Equation 3.3, and the number of segments. The number p is a penalty factor that counteracts overfitting.

We have developed an algorithm that finds the optimal segmentation with respect to a given information criterion (of the form of Equation 3.4); that is, it finds the segmentation with minimal information according to the given information criterion.

There are several ways to choose a good information criterion. One could pick one of the two widely used information criteria, the Bayesian information criterion (BIC) or the Akaike information criterion (AIC). The BIC has a penalty $p = \ln(n)$, where n is the number of recorded data points for which the model is constructed. The AIC has a constant penalty $p = 2$. The BIC has been used previously in the windowing technique that obtained diffusion coefficients for the dBBMM [16].

On the other hand, custom penalty factors provide more flexibility. There is one important observation regarding the penalty factor.

Observation 2 *Given a trajectory τ and an information criterion IC with penalty factor p, let S be the optimal segmentation of τ with respect to IC, let IC' be another information criterion with penalty factor $p' > p$, and let S' be the optimal segmentation of τ with respect to IC'. The number of segments of S' is at most the number of segments of S.*

Intuitively, increasing the penalty factor p "decreases" the number of segments of the optimal segmentation. Picking the right p can be a complex interactive process. We have developed means to make this easier. This is described in Section 3.3.3.

3.3 Segmentation algorithm

In this section we present our novel segmentation algorithm which finds the segmentation and diffusion coefficient assignment that is optimal with respect to a given information criterion IC.

3.3.1 Dynamic programming approach

Our algorithm is basically a Dynamic Program (DP). Let Opt_i denote the optimal segmentation of $\tau[0, i]$ with respect to IC. We compute all Opt_i in increasing order of i. In order to compute Opt_i we also maintain a two-dimensional table $OptFixedLast$. Each entry $OptFixedLast_{i,v}$ stores the best segmentation (minimizing the IC) of subtrajectory $\tau[0, i]$ that ends with a segment with diffusion coefficient $\sigma_m^2 = v$.

For this table $OptFixedLast_{i,v}$, we need to sample the set of $\sigma_{m,i}^2$ values. Let $V = \{v_1, v_2, \ldots, v_V\}$ denote this sample set. A regularly sampled set of variances starting at $\varepsilon > 0$ and ending with v_{max} yields good results in practice. We note that v_{max} can be determined by an exponential search that ends when no diffusion coefficients are fitted with $\sqrt{v_{max}}$.

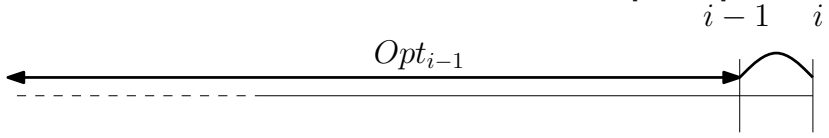
Given $OptFixedLast_{i,v}$ for all v, it is straightforward to compute Opt_i. We assume that the last segment of the optimal segmentation ends either with v_1 or v_2 etc, hence

$$Opt_i = \underset{S \in \{OptFixedLast_{i,v} \mid v \in V\}}{\arg\min} IC(S),$$

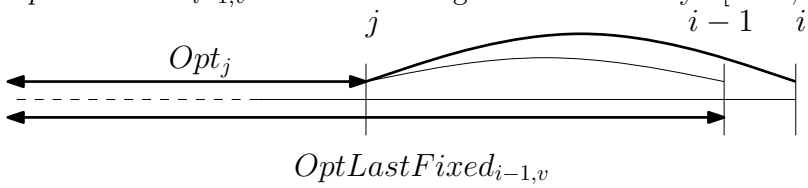
where $IC(S)$ denotes the information of segmentation S according to information criterion IC. The table $OptFixedLast$ is computed using the greedy property stated in Lemma 19.

Lemma 19 *$OptFixedLast_{i,v}$ is equal to one of the following options:*

Append: Opt_{i-1} appended with the one-bridge segment $\tau[i-1, i]$.



Extend: $OptLastFixed_{i-1,v}$ with the last segment extended by $\tau[i-1, i]$.



Proof. The last segment of $OptFixedLast_{i,v}$ consists of one Brownian bridge $\tau[i-1, i]$, or it consists of strictly more than one Brownian bridge.

Case 1: Last segment consists of one Brownian bridge. The last segment is in this case a one-bridge segment $\tau[i-1, i]$ with $\sigma_m^2 = v$. We are left with the part $\tau[0, i-1]$. Consider the loglikelihood function of Equation 3.3. Each bridge corresponds to its own independent term. Hence the part $\tau[0, i-1]$ should be segmented optimally independent of the rest of the trajectory. By definition, this results in segmentation Opt_{i-1}, which proves the append option.

Case 2: Last segment consists of strictly more than one Brownian bridge. In this case $OptFixedLast_{i,v}$ is set to $OptFixedLast_{i-1,v}$, with the last segment extended by one bridge. We prove that this greedy choice is correct.

Let S_1 denote the segmentation that is equal to $OptFixedLast_{i-1,v}$, with the last segment extended by one bridge. Let S_2 denote the segmentation of subtrajectory $\tau_{1,i-1}$ according to $OptFixedLast_{i,v}$ (with the last segment one bridge shorter). Since $OptFixedLast_{i,v}$ minimizes the IC we have

$$IC(OptFixedLast_{i,v}) \leq IC(S_1).$$

Substitution of both values yields:

$$IC(S_2) - 2L_i \leq IC(OptFixedLast_{i-1,v}) - 2L_i,$$

where L_i is short for $L(v\bar{\tau}[i, i+1])$. This implies that

$$IC(S_2) \leq IC(OptFixedLast_{i-1,v}).$$

By definition, segmentation $IC(OptFixedLast_{i-1,v})$ minimizes the information over all possible values of S_2. Hence $IC(S_2) = IC(OptFixedLast_{i-1,v})$. Greedily choosing $S_2 = OptFixedLast_{i-1,v}$ is hence correct. This proves the extend option. □

Lemma 19 implies that we can compute $OptLastFixed_{i,v}$ in a dynamic programming fashion, looping over i and v. We compute each new entry $OptLastFixed_{i,v}$ using a comparison between two already computed table entries:

$$IC(OptLastFixed_{i,v}) = \min\left(IC(OptLastFixed_{i-1,v}), IC(Opt_{i-1}) + p\right) + \log\left(L(v \mid \tau[i-1, i])\right)$$

If $IC(Opt_{i-1}) + p$ is smaller, $OptLastFixed_{i,v}$ is set according to the append option. Otherwise it is set according to the extend option.

In our algorithm we store segmentations in the same way as the algorithm of Section 2.5: we only store the length of the last segment. Furthermore, we store for each segmentation the information according to our IC. For the Opt_i segmentations the diffusion coefficient of the last segment is stored as well. Using this storage format computing $OptLastFixed_{i,v}$ takes only constant time. The actual segmentation can be retrieved from the tables in $O(n)$.

Theorem 20 *The optimal segmentation of a trajectory τ with respect to an information criterion IC can be computed in $O(n|V|)$ time, and $O(n + |V|)$ space, where $|V|$ is the size of the set of sampled Brownian variances and n the number of points on the trajectory.*

Proof. Our algorithm computes the table $OptLastFixed_{i,v}$, which is size $n \times |V|$. Using Lemma 19, this takes constant time per entry, hence $O(n|V|)$ in total. Computing the table with the overall optimal segmentations Opt_i takes $O(|V|)$ per entry, hence also $O(n|V|)$ in total. To compute an entry $OptLastFixed_{i,v}$ in loop i, only $OptLastFixed_{i-1,v}$ is needed. No access is needed to any entry $OptLastFixed_{i',v}$, with $i' < i - 1$. Hence the space of the $OptLastFixed$ table can be reused. The algorithm thus requires only $O(n + |V|)$ space. □

We have performed several small test with our algorithm. An example is shown in Figure 3.5. The segmentation is optimal with respect to the BIC. The trajectory data is available at Movebank. It contains the movement of a fisher [16].

Fixed number of segments

It is straightforward to change the algorithm in such a way that it computes the segmentation with a fixed number k of segments and maximal likelihood. All tables get an extra dimension:

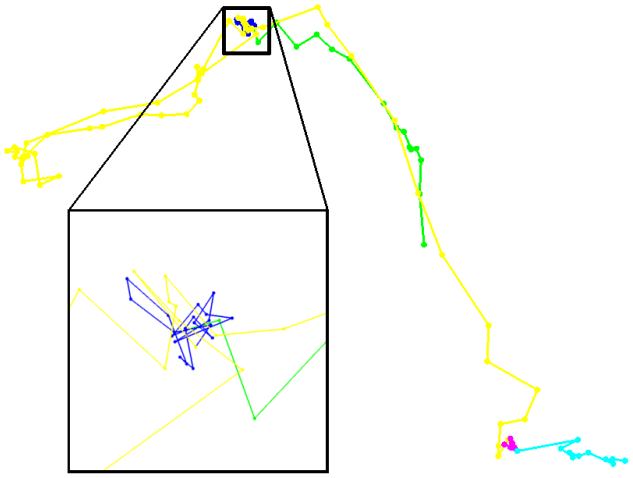


Figure 3.5: Segmentation of a real world trajectory. Optimal with respect to BIC. Segments are shown in different colors.

the number of segments. Those tables can be computed using a greedy property that is similar to Lemma 19.

To compute $OptLastFixed_{i,v,m}$ (with m the number of segments) we choose between either appending a one-bridge segment to $Opt_{i-1,m-1}$ and extending the last segment of $OptLastFixed_{i-1,v,m}$ by one bridge. Each table entry is hence computed in constant time, which results in a $O(n|V|k)$ running time and $O(n + |V|k)$ required space. Note that this algorithm does not only give the best segmentation with k segments, but also the best segmentations with $k-1, k-2, \ldots, 1$ segments. Furthermore, the algorithm can be run incrementally, increasing k one by one.

3.3.2 Table compression

We can significantly speed up the basic segmentation algorithm from Section 3.3.1, by using the following observation in the DP table. Let $S.last$ denote the starting index of the last segment of segmentation S. In all data sets we inspected, the variation of $OptFixedLast_{i,v}.last$ for a fixed i over all v was very limited. Given a set of ordered Brownian variances $v_1 < v_2 < \cdots < v_V$ there seems to be only a constant (with respect to V and n) number of variances v_j for which $OptFixedLast_{i,v_j}.last \neq OptFixedLast_{i,v_{j+1}}.last$. An example of this repetitive behavior on a typical real world data set is shown in Table 3.1. The table consists of 5000 rows, but there are only 4 changes of the *last* field.

Table 3.1: DP table, with repetitive *last* field.

	$OptLastFixed_{110,v}$	
v	Start of last segment	Information
1.00	109	-711.2
...	109	...
12.28	109	-681.6
12.29	96	-681.6
...	96	...
33.13	96	-679.0
33.14	99	-679.1
...	99	...
37.05	99	-679.3
37.06	105	-679.3
50.00	105	...

Let's assume throughout this section that for each i there are at most w variances at which the startindex $OptFixedLast_{i,v}.last$ changes. We improve the original algorithm by storing the DP table in a compressed way. Multiple entries of $OptFixedLast_{i,v}$ with equal *last* field can be compressed into one entry $OptFixedLast_{i,[v_s,v_f]}$. The proposed compressed table is shown in Table 3.2.

Table 3.2: Compressed DP table corresponding to the uncompressed Table 3.1, all consecutive entries with equal start field are compressed.

	$OptLastFixed_{110,v}$	
v	Start of last segment	Information
$[1.00, 12.28]$	109	$f_1(\sigma_m^2)$
$[12.29, 33.13]$	96	$f_2(\sigma_m^2)$
$[33.14, 37.05]$	99	$f_3(\sigma_m^2)$
$[37.06, 50.00]$	105	$f_3(\sigma_m^2)$

Note that the Information field is not constant for all variances that are stored in the same compressed table entry. The information of a segmentation $OptFixedLast_{i,\sigma_m^2 \in [v_s,v_f]}$ consists of two parts: a variable part for the last segment, which depends on the exact value of $\sigma_m^2 \in [v_s, v_e]$, and a constant part for the rest of the segmentation, which equals the information of the optimal segmentation of $\tau[0, OptFixedLast_{i,[v_s,v_f]}.last]$. The constant part can be stored without causing problems to the compression.

The variable part is a function which maps σ_m^2 to $\log\big(L(\sigma_m^2 \mid \tau[s,i])\big)$, with $s = OptFixedLast_{i,[v_s,v_f]}.last$. This function can be computed using Equation 3.2. Figure 3.6 shows an example of such a function. In many practical situations this equation solves to function with a constant number of parameters. For instance, if no error in the recorded points (all $\delta = 0$) is assumed the variable part can be stored by a function of the form:

$$c_1 + c_2 \log(\sigma_m^2) + \frac{c_3}{\sigma_m^2}, \tag{3.5}$$

with c_1, c_2 and c_3 constants.

If the measurements were done at a regular interval (hence $\alpha = 1/2$) and the errors in the recorded points are equal, then the variable part has the form:

$$c_1 \log(c_2\sigma_m^2 + c_3) + \frac{c_4}{\sigma_m^2 + c_5}, \tag{3.6}$$

with c_1, c_2, c_3, c_4 and c_5 constants.

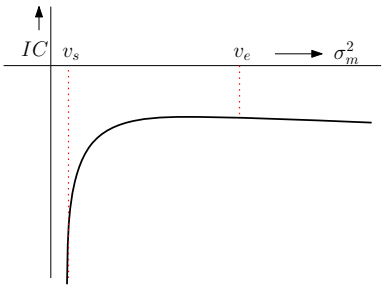


Figure 3.6: Function that maps σ_m^2 to $\big(L(\sigma_m^2 \mid \tau[last, i])\big)$.

The change in our DP table representation causes the need for several changes in the DP algorithm. First of all, we do not loop over a sampled set V of variance. Instead, we loop over all variance intervals that together cover the whole variance space.

In the original algorithm we compared the information of the append and extend option. This was a simple comparison of two floating point numbers. In our compressed setting the comparison between append and extend yields comparing two functions of σ_m^2 as is depicted in Figure 3.7. Note that the function corresponding to the append option is a constant function. Computing this the minimum of two functions takes constant time given a suitable representation of the extend function, such as the ones of equations 3.5 and 3.6.

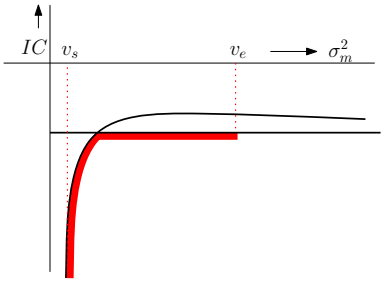


Figure 3.7: Determining for which σ_m^2's to append and for which to extend by taking the minimum of two functions. This minimum is shown in red.

When this minimum of two functions is not completely equal to one of the two compared functions, the variance interval is split in a number of parts: parts of the append function and parts of the extend function. If the information can be described by a function that consists of an increasing part followed by a decreasing part (such as the functions of equations 3.5 and 3.6), then this number of parts is at most three.

Extending the last segment or appending a new segment with a different variance takes constant time given suitable representation of the information. If we extend the last segment we change the constant number of likelihood function parameters according to the added Brownian bridge. If we append a new segment we initialize the constant number of likelihood function parameters according to the new segments single Brownian bridge. After looping over all intervals we merge neighboring intervals that have equal start field. This guarantees that each of the i loops in the computation of the compressed $OptLastFixed$ table takes $O(w)$ time.

The segmentation Opt_i is equal to the $OptLastFixed_{i,v}$ with minimal IC. In the original algorithm this is computed by simply taking the minimum over all $v \in V$. In the compressed table variant, we find the best $v \in [v_s, v_e]$ for every variance interval; that is, the v for which the information criterion is minimal, and take the best of those segmentations. Computing the best variance in a variance interval takes constant time given a suitable representation of the extend function. The following theorem summarizes our running time analysis.

Theorem 21 *Given a trajectory τ consisting of n points, the optimal segmentation with respect to an information criterion IC can be computed in $O(nw)$ time, where w is maximal the number of indices at which the startindex $OptFixedLast_{i,v}.last$ changes along v.*

Note that the numerical operations on the variance functions take constant time with respect to n and w, but their running times do depend on their precision and the exact numerical/analytical methods that are used.

3.3.3 Choosing the penalty factor

Given a trajectory, the result of the dBBMM-based segmentation algorithm only depends on the information criterion IC, which is completely defined by one parameter: the penalty factor p. Recall from Section 3.2.1 that increasing the penalty factor p decreases the number of segments of the optimal segmentation. Hence this parameter controls the conceptual "scale" of the segmentation.

We have experimented with different values of p on various data sets and noticed that the most suitable choices for p correspond to the most stable values. The concept of stability of segmentation parameters was already introduced in Section 2.7. A parameter value is *unstable* if a small change to its value results in a large change in the segmentation, i.e., in a change of the number of segments. The stability of a parameter can be visualized in a stability diagram, which is basically a plot of the function which maps a value in the parameter space to the number of segments in the segmentation corresponding to the parameter value.

In Section 2.7 we sample the parameter space and perform multiple segmentation computations. In this particular case however, we do not have to sample the penalty space. An exact method exists and is based on a simple observation.

Observation 3 *If a segmentation S is optimal with respect to a certain information criterion IC, and it consists of k segments, then segmentation S maximizes the loglikelihood of the diffusion coefficients over all segmentations with exactly k segments.*

Our method to compute the stability diagram consists of two steps. First we compute the information of the optimal segmentations with $1, 2, \ldots, n$ segments. We will denote these segmentations by respectively $S_1, S_2, \ldots, S_n$. They can be computed in $O(n^2)$ time by the algorithm described in Section 3.3.1. All other segmentations can be ignored due to Observation 3.

In the second step the stability diagram is computed from these segmentations. Note that for large enough p the segmentation S_1 maximizes the IC. The information of this segmentation for any information criterion is described by the line ℓ_1 with equation $(x, y) = (p, IC(p))$, where $IC(p) = L_{S_1} + p$, with L_{S_1} a constant depending on S_1. More general, each segmentation S_i corresponds to a line ℓ_i with equation $(p, IC(p) = L_{S_i} + i \cdot p)$. An example is shown in Figure 3.8.

Segmentation S_1 is optimal for all $p > p_1$, where p_1 is the rightmost intersection of ℓ_1 and another line ℓ_j. It should be clear that the stability diagram increases (viewed from right to left) from 1 to j at p_1. In a similar way, for $p_2 < p \leq p_1$ the segmentation S_j is optimal, where p_2 is the rightmost intersection point of ℓ_j and another line $\ell_{j'}$ left of p_1. At p_2, the stability diagram increases from j to j'. This argument can be continued until the whole range of $p > 0$ is covered.

Computing the points $p_1, p_2, \ldots$ given the lines $\ell_1, \ell_2, \ldots, \ell_n$ is a well known problem in Computation Geometry. The points $p_1, p_2, \ldots$ define the lower envelope of lines $\ell_1, \ell_2, \ldots, \ell_n$. There is a close connection between the lower envelope of a line set and the upper convex hull of a point set.

Consider the dual points $\ell_1^*, \ell_2^*, \ldots, \ell_n^*$ of the lines $\ell_1, \ell_2, \ldots, \ell_n$, defined by the standard dual transform, which defines the dual transform of point $p = (a, b) \in \mathbb{R}^2$ to be line $p^* := (a \cdot x - b = y)$. The upper convex hull of the points $\ell_1^*, \ell^*2, \ldots, \ell_n^*$ is the dual of the lower envelop of the $\ell_1, \ell_2, \ldots, \ell_n$ [10]. Recall that the upper convex hull of a pointset can be computed in $O(n)$ by Graham's scan [10], given a sorted input. The points $\ell_1^*, \ell_2^*, \ldots, \ell_n^*$ are already sorted on their x-coordinate, since the lines $\ell_1, \ell_2, \ldots, \ell_n$ are computed in ascending order of slope. The following theorem summarizes our result.

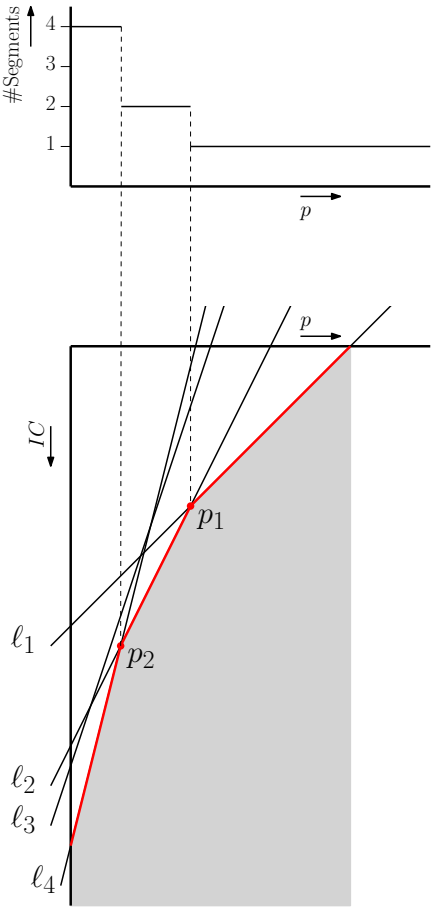
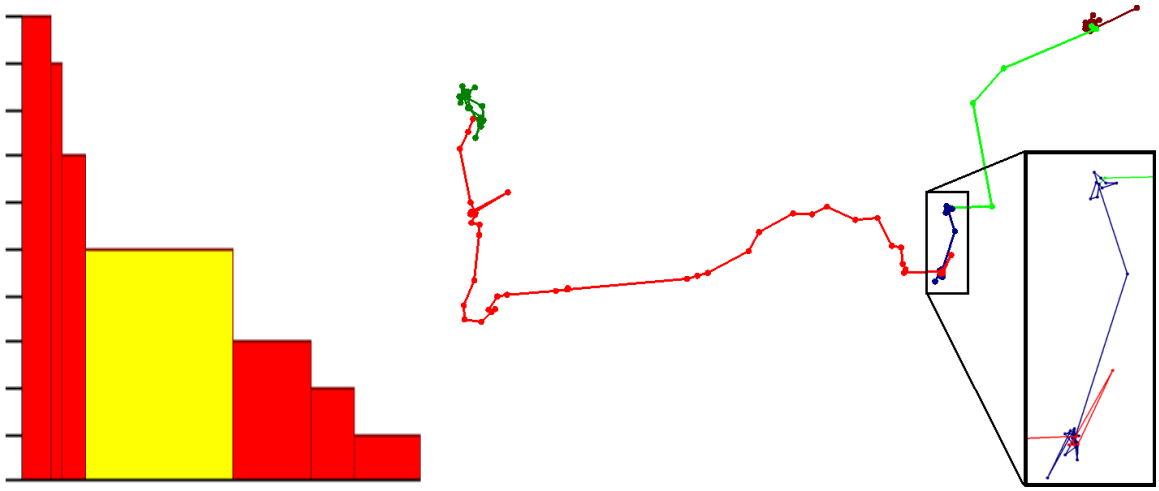


Figure 3.8: Relation between lines $\ell_1, \ell_2, \ldots, \ell_n$ and the stability diagram.

Theorem 22 *Given the information of the optimal segmentations with* $1, 2, \ldots, n$ *segments the stability diagram for the penalty factor can be computed in* $O(n)$ *time.*

Unfortunately, the computation of the lines $\ell_1, \ell_2, \ldots, \ell_n$ is the bottleneck in terms of running time in our computation of the stability diagram. Computing the n lines takes $O(n^2)$ time in total.

However, if we put an upper bound k_{max} on the number of segments, then it is sufficient to compute only lines $\ell_1, \ell_2, \ldots, \ell_{k_{max}}$, which takes $O(k_{max}n)$. Computing the stability diagram takes $O(k_{max})$. An example on a real world trajectory (fisher set) is given in Figure 3.9. The stability diagram clearly indicates that choosing 5 segments is the best option.



(a) Stability diagram (b) Segmentation consisting of 5 segments

Figure 3.9: A trajectory and its stability diagram ($k_{max} = 10$). The segmentation corresponds to the broadest step in the stability diagram.

3.4 Adding criteria

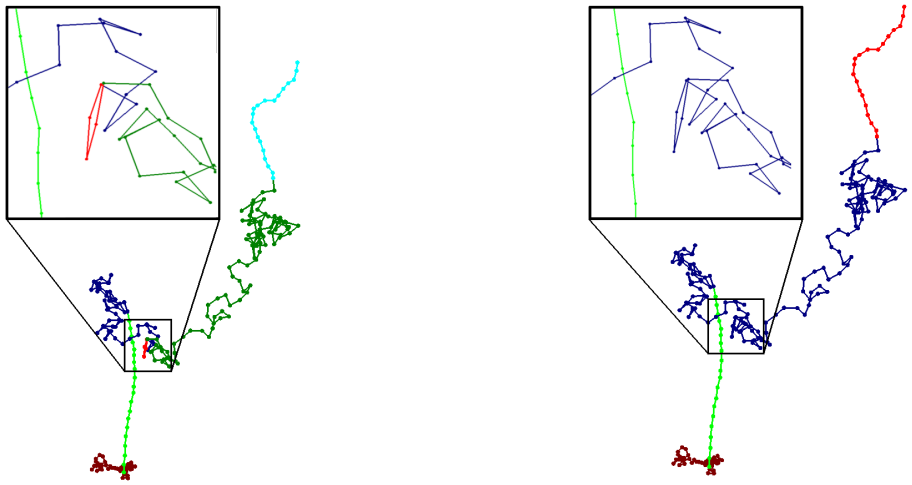
In this section we combine our BBMM-based segmentation method with criterion based methods. Given a trajectory, an information criterion and a criterion, the combination method optimizes a segmentation with respect to an information criterion like before, but this time the optimization is subject to the constraint that each segment satisfies the criterion.

In Section 3.4.1 we show how to extend our method from Section 3.3.1 in such a way that it handles an increasing monotone criterion (those criteria were discussed in Section 2.2). In Section 3.4.2 a method is presented which handles stable criteria. This segmentation method is basically an adaptation of the framework described in Chapter 2.

3.4.1 Increasing monotone criteria

The algorithm described in Section 3.3.1 did not put any restrictions on the segments of the output. The only goal was to minimize the information criterion. On several data sets that we tested, we noticed that the likelihood maximization method was biased towards small diffusion coefficients ($\sigma_m^2 \approx 0$). The loglikelihood function has an asymptote at $\sigma_m^2 = 0$, and hence diffusion coefficients of approximately 0 get an extremely low score by the information criterion. So low even, that it can compensate the penalty for adding segments completely. We experienced that this bias in the loglikelihood method frequently leads to very short segments with an diffusion coefficient of almost zero. A typical example is shown in Figure 3.10(a). This issue can be resolved by putting a lower bound on the number of points on a segments as is shown in Figure 3.10(b).

Furthermore, consider an animal that is in behavioral state A for a long time. Assume that it switches to state B and quickly back to state A. There are two ways to segment this data, either by three or by one segment. To forbid the three segment option we could put a lower bound on the duration of segments. Note that this is essentially a form of outlier handling.



(a) Without criteria

(b) Lower bound on number of points

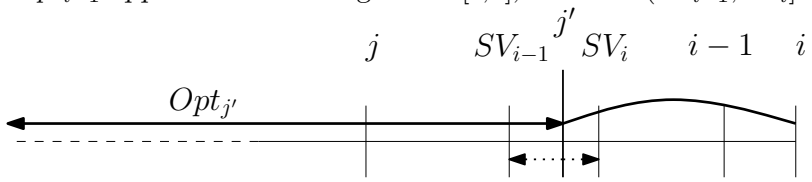
Figure 3.10: Adding a lower bound on the number of points per segments counteracts the bias in the likelihood method. Each segments is shown in a different color.

The above considerations motivated us to extend the algorithm from Section 3.3.1. The input of the extended versions is not just a trajectory τ and an information criterion IC, but also an increasing monotone criterion C. Note that the class of increasing monotone criteria contains both the minimum duration and the minimum point count criteria that were suggested above. Our extended algorithm finds the segmentation and diffusion coefficient assignment which minimizes the information criterion IC, subject to the constraint that every segment satisfies C.

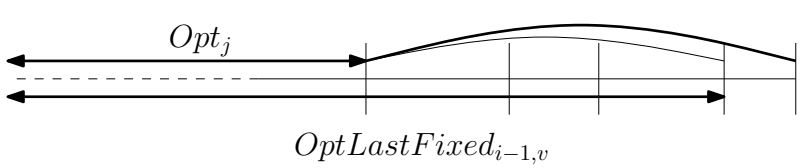
First we compute for every trajectory index j the largest index i for which $\tau[i, j]$ satisfies C. This is stored in SV_j. In Section 2.4 was described how to compute this array efficiently for various criteria. We can formulate a more general variant of Lemma 19, which states the options for $OptFixedLast_{i,v}$ in terms of array SV:

Lemma 23 *$OptFixedLast_{i,v}$ is equal to one of the following options:*

Append: Opt_{i-1} appended with a segment $\tau[\ell, i]$, with $\ell \in (SV_{i-1}, SV_i]$.



Extend: $OptLastFixed_{i-1,v}$ with the last segment extended by $\tau[i-1, i]$.



$OptLastFixed_{i-1,v}$

Proof. Let j' be the starting index of the last segment of $OptFixedLast_{i,v}$. By definition $j' \leq SV_i$. We make a distinction between two cases: either $j' > SV_{i-1}$ or $j' \leq SV_{i-1}$.

Case 1: $j' > SV_{i-1}$**.** The last segment is thus $\tau[j', i]$ with $\sigma_m^2 = v$. As in case 1 of Lemma 19, the part $\tau[0, j']$ should be segmented optimally according to $Opt_{j'}$. This proves the append option.

Case 2: $j' \leq SV_{i-1}$ Let j be the starting index of the last segment of $OptFixedLast_{i-1,v}$. Note that the segment $\tau[j', i-1]$ is valid, because $j' \leq SV_{i-1}$. We can show that $j' = j$. The proof is completely analogous to case 2 of Lemma 19. □

Lemma 23 can be used to formulate our segmentation algorithm as a dynamic program, just like Lemma 19 in Section 3.3.1. The main difference is that in the new extended setting we compare $SV_i - SV_{i-1} + 1$ options instead of two in the computation of $OptFixedLast_{i,v}$. It is easy to see that the number $SV_i - SV_{i-1} + 1$ could be $O(n)$. However, summing this number over all $i = 1, 2, \ldots, n$ the following telescoping behavior is observed:

$$\sum_{i=0}^{n} (SV_i - SV_{i-1} + 1) = n + \sum_{i=0}^{n} (SV_i - SV_{i-1}) = n + SV_n - SV_0 \leq 2n.$$

Hence the algorithm takes $O(n|V|)$ time. We can no longer reuse table entries. Hence the space required by the algorithm is $O(n|V|)$. Table compression gets a little bit more complicated, because the appended segments might be longer than one bridge, but it can still be done within the same (asymptotic) running time with some straightforward additional bookkeeping and numerical procedures. The following theorem follows immediately.

Theorem 24 *The optimal segmentation of a trajectory τ with respect to an information criterion IC subject to the constraint that each segment satisfies an increasing monotone criterion C can be computed in $O(n|V|)$ time, where $|V|$ is the size of the set of sampled Brownian variances and n the number of points on the trajectory.*

3.4.2 Stable criteria

In Chapter 2 we discussed how to segment based on stable criteria. In this section we combine this framework for stable criteria with our dBBMM-based segmentation method. The resulting framework is basically an adaptation of the framework of Chapter 2.

Our approach is also similar to the approaches from Sections 3.3.1 and 3.4.1. For increasing $i = 0, 1, \ldots, n$ we compute the optimal segmentation of $\tau[0, i]$ (denoted by Opt_i). This optimum is computed as the minimum of $OptLastFixed_{i,v}$ over all v, where $OptLastFixed_{i,v}$ denotes the optimal segmentation of $\tau[0, i]$ that ends on a segment with $\sigma_m^2 = v$.

Unlike the problems of Sections 3.3.1 and 3.4.1, we are not able to pick the optimal segmentations greedily. We take a different approach that is more similar to the framework of Chapter 2. Our algorithm relies on the following property (compare this to the property described in Observation 1).

Observation 4 *The segmentation $OptLastFixed_{i,v}$ (if it exists) either consists of just one segment, or it is equal to Opt_j appended with a segment $\tau[j, i]$, where j is an index such $\tau[j, i]$ is valid.*

To enable efficient testing of segment validity we compute the compressed start-stop diagram $\mathcal{S}$ of the criterion C using the methods described in Sections 2.3 and 2.4.

The main part of the algorithm is *ComputeSegmentationBBMM*. It consists of n outer loops. Within each outer loop we loop over all $v \in V$. Each of those inner loops is similar to a loop in the algorithm *ComputeSegmentation*. We maintain one tree $\mathcal{T}_v$ for each $v \in V$. In iteration i a $\mathcal{T}_v$ stores all segmentations that consist of Opt_j appended with a segment $\tau[j, n]$ (with $\sigma_m^2 = v$) for all $j = 0, 1, \ldots, i$. A segmentation is stored in node by a *last* field, which indicates the starting index of the last segment, and an $info$ field, which stores the information of the whole segmentation. The segmentations/nodes are ordered on the *last* field.

Algorithm *ComputeSegmentationBBMM*$(\tau, \mathcal{S})$
1. **for** each $v \in V$
2. **do** Initialize empty $\mathcal{T}_v$;
3. Create new node ν_0;
4. $\nu_0.last \leftarrow 0$; $\nu_0.info \leftarrow 0$;
5. $\mathcal{T}_v.Insert(\nu_0)$
6. **for** $i \leftarrow 1$ **to** n
7. **do for** each $v \in V$
8. **do** Create new node ν;
9. $\nu.info \leftarrow \infty$;
10. **for** each block b at row i of $\mathcal{S}$
11. **do** $\nu' \leftarrow \mathcal{T}.GetMinimalInfo(b)$;
12. **if** $\nu'.info < \nu.info$
13. **then** $\nu \leftarrow \nu'$;
14. $OptLastFixed_{i,v} \leftarrow \nu$ with last segment shortened to i;
15. $Opt_i \leftarrow \arg\min_{S \in \{OptFixedLast_{i,v} \mid v \in V\}} S.info$;
16. **for** each $v \in V$
17. **do** Create new node ν;
18. $\nu \leftarrow Opt_i$ appended with segment $\tau[i, n]$ $\mathcal{T}.Insert(\nu)$;

Note that we do not store the segmentation $OptLastFixed_{i,v}$ explicitly. However, we can find it efficiently. Let $OptLastFixedExt_{i,v}$ be equal to $OptLastFixed_{i,v}$, except for the last segment, which is extended to index n instead of i. Observation 4 implies that $OptLastFixedExt_{i,v}$ is in $\mathcal{T}_v$ at the start of iteration i and that $\tau[last, i]$ satisfies the criterion. Hence we find it by taking the segmentation with minimal information over all segmentations in $\mathcal{T}_v$ for which $\tau[last, i]$ is valid. Note that we can ignore the part of the segmentations beyond index i in this comparison, since it has equal information for all segmentations, and hence makes no difference.

We can find this segmentation/node efficiently by calling the *GetMinimalInfo* procedure for every block in compressed start-stop matrix $\mathcal{S}$ on row i. The procedure *GetMinimalInfo* is similar to the *GetMinimalCount* procedure that was described in Section 2.5. It finds the node with minimal $info$ (instead of *count*). Hence $\mathcal{T}_v$ is augmented with the fields $\nu.min_{info}$ and $\nu.argmin_{info}$.

When $OptLastFixedExt_{i,v}$ is found in $\mathcal{T}_v$ we compute $OptLastFixed_{i,v}$ by removing the part $\tau[i, n]$ from the last segment. This takes only constant time if we have precomputed $\sum_{j=0}^{i} L_j$ for all i. Given $OptLastFixed_{i,v}$ for all $v \in V$ we can compute Opt_i. Segmentation Opt_i is equal to the segmentation $OptLastFixed_{i,v}$ with minimal $info$ over all v.

Finally we need to update the $\mathcal{T}_v$ structures. We add a new node to every tree $\mathcal{T}_v$ that corresponds to the segmentation Opt_i appended by the segment $\tau[i, n]$ with $\sigma_m^2 = v$. Such a node can be computed in constant time, again using precomputed values for $\sum_{j=0}^{i} L_j$. The running time analysis is summarized in the next theorem.

Theorem 25 *The optimal segmentation of a trajectory τ with respect to an information criterion IC subject to the constraint that each segment satisfies a λ-stable criterion can be computed in $O(|V|(n+\lambda)\log n)$ time, where $|V|$ is the size of the set of sampled Brownian variances and n the number of points on the trajectory.*

Chapter 4

Conclusions and future work

In this thesis we have introduced a framework for criterion-based segmentation that can efficiently handle a broader and more powerful class of criteria than previous algorithms. It allows for segmentation by movement states and in contrast to previous methods we can handle a broad range of state-based rules governing state transitions and additional optimization goals to fine tune the exact point of transitions. We have also introduced interactive parameter selection guided by segmentation stability.

Our segmentation framework proved to be useful in practice and yielded segmentations similar to manual ones. Hence they can substitute these, where a manual segmentation is not possible, e.g., due to the size of data to be analyzed. In this context the interactive approach has a large potential. Incorporating advanced statistical methods into interactive parameter selection could guide the user even more.

Furthermore, we have developed a novel segmentation method based on the dBBMM. As a model fitting algorithm, our method is better than the previous window based model fitting algorithm in terms of the resulting model complexity. The method is parametrized by only one parameter, which can be selected automatically or interactively guided by a stability diagram. We have also combined the novel dBBMM-based method with the criterion-based methods.

On the algorithmic side, future work could focus on the more advanced tie breaking and state transition rules, especially targeting outlier handling. In this thesis we discussed the problem of allowing a fraction of outliers per segment, but in practice it makes sense to define even more complex criteria that take the location of those outliers into account. For instance, having outliers on the start or end of a segment is undesirable. Also the relative location of outliers could be of importance: a cluster of outliers is worse than evenly distributed outliers.

Furthermore, we noticed that our dBBMM-based method is practically always (on real world data) segmenting "hierarchically" for increasing penalty factor; that is, given a segmentation which is optimal for a certain penalty factor, increasing the penalty factor results in the splitting of exactly one segment and *no* other changes. Further increase of the penalty factor results again in only one extra split, and so on. This suggests that a different approach could be taken for the computation of the optimal segmentations with fixed number of segments, which are used in the computation of the stability diagram.

Our experiments regarding the dBBMM-based segmentation algorithm were limited and did not comprise an in-depth ecological evaluation. We suggest a thorough evaluation of our dBBMM segmentation in the form of case studies. Furthermore, comparing the dBBMM-based method to the criterion-based method could provide inside to how the two methods could benefit from each other. This knowledge could help formulating effective criteria for our combined framework.

Bibliography

[1] H. Alt and M. Godau. Computing Fréchet distance. *International Journal of Computational Geometry & Applications*, 5:75–91, 1995.

[2] B. Aronov, A. Driemel, M. J. van Kreveld, M. Löffler, and F. Staals. Segmentation of trajectories for non-monotone criteria. In *Proc. 24th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1897–1911, 2013.

[3] M. Benkert, B. Djordjevic, J. Gudmundsson, and T. Wolle. Finding popular places. *Int. J. Comput. Geometry Appl.*, 20(1):19–42, 2010.

[4] K. Buchin, M. Buchin, J. Gudmundsson, M. Löffler, and J. Luo. Detecting commuting patterns by clustering subtrajectories. *Int. J. Comput. Geometry Appl.*, 21(3):253–282, 2011.

[5] K. Buchin, S. Sijben, T. J. M. Arseneau, and E. P. Willems. Detecting movement patterns using brownian bridges. In *SIGSPATIAL/GIS*, pages 119–128, 2012.

[6] M. Buchin, A. Driemel, M. van Kreveld, and V. Sacristan. Segmenting trajectories: A framework and algorithms using spatiotemporal criteria. *Journal of Spatial Information Science*, 3:33–63, 2011.

[7] M. Buchin, H. Kruckenberg, and A. Kölzsch. Segmenting trajectories based on movement states. In *Proc. 15th Internat. Sympos. Spatial Data Handling (SDH)*, pages 15–25. Springer-Verlag, 2012.

[8] T. M. Chan. A dynamic data structure for 3-d convex hulls and 2-d nearest neighbor queries. In *Proc. 17th Annu. ACM-SIAM Sympos. Discrete Algorithms (SODA)*, pages 1196–1202. ACM, 2006.

[9] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, 3rd edition, 2009.

[10] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications.* Springer-Verlag, Santa Clara, CA, USA, 3rd ed. edition, 2008.

[11] S. Dodge, R. Weibel, and E. Forootan. Revealing the physics of movement: comparing the similarity of movement characteristics of different types of moving objects. *Computers, Environment and Urban Systems*, 33(6):419–434, November 2009.

[12] R. C. Gonzalez and R. E. Woods. *Digital Image Processing (3rd Edition).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006.

[13] J. Gudmundsson, P. Laube, and T. Wolle. Computational movement analysis. In W. Kresse and D. M. Danko, editors, *Springer Handbook of Geographic Information*, pages 423–438. Springer Berlin Heidelberg, 2012.

[14] J. Harguess and J. K. Aggarwal. Semantic labeling of track events using time series segmentation and shape analysis. In *Proceedings of the 16th IEEE international conference on Image processing*, pages 4261–4264. IEEE, 2009.

[15] J. S. Horne, E. O. Garton, S. M. Krone, and J. S. Lewis. Analyzing animal movements using Brownian bridges. *Ecology*, 88(9):2354–63, September 2007.

[16] B. Kranstauber, R. Kays, S. D. Lapoint, M. Wikelski, and K. Safi. A dynamic Brownian bridge movement model to estimate utilization distributions for heterogeneous animal movement. *The Journal of animal ecology*, 81(4):738–46, July 2012.

[17] P. Laube and R. S. Purves. How fast is a cow ? Cross-Scale Analysis of Movement Data. 15(3):401–418, 2011.

[18] R. E. van Wijk, A. Kölzsch, H. Kruckenberg, B. S. Ebbinge, G. J. D. M. Müskens, and B. A. Nolet. Individually tracked geese follow peaks of temperature acceleration during spring migration. *Oikos*, 121(5):655–664, 2012.