SANDIA REPORT

SAND2015-8120
Unlimited Release
Printed September 2015

PANTHER: Trajectory Analysis

Mark D. Rintoul, Andrew T. Wilson, Chris G. Valicka, W. Philip Kegelmeyer,
Timothy M. Shead, Benjamin D. Newton and Kristina R. Czuchlewski

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation,
a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's
National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-8120
Unlimited Release
Printed September 2015

PANTHER: Trajectory Analysis

Mark D. Rintoul *
Sandia National Laboratories, Albuquerque, NM
and
Andrew T. Wilson †
Sandia National Laboratories, Albuquerque, NM
and
Chris G. Valicka ‡
Sandia National Laboratories, Albuquerque, NM
and
W. Philip Kegelmeyer §
Sandia National Laboratories, Livermore, CA
and
Timothy M. Shead ¶
Sandia National Laboratories, Albuquerque, NM
and
Benjamin D. Newton ‖
Universit of North Carolina, Chapel Hill, NC
and
Kristina R. Czuchlewski **
Sandia National Laboratories, Albuquerque, NM

*mdrinto@sandia.gov
†atwilso@sandia.gov
‡cgvalic@sandia.gov
§wpk@sandia.gov
¶tshead@sandia.gov
‖bn@cs.unc.edu
**krczuch@sandia.gov

Abstract

We want to organize a body of trajectories in order to identify, search for, classify and predict behavior among objects such as aircraft and ships. Existing comparison functions such as the Fréchet distance are computationally expensive and yield counterintuitive results in some cases. We propose an approach using feature vectors whose components represent succinctly the salient information in trajectories. These features incorporate basic information such as total distance traveled and distance between start/stop points as well as geometric features related to the properties of the convex hull, trajectory curvature and general distance geometry. Additionally, these features can generally be mapped easily to behaviors of interest to humans that are searching large databases. Most of these geometric features are invariant under rigid transformation. We demonstrate the use of different subsets of these features to identify trajectories similar to an exemplar, cluster a database of several hundred thousand trajectories, predict destination and apply unsupervised machine learning algorithms.

Acknowledgments

We would like to acknowledge Randy Brost, Hyrum Anderson, Eddie Ochoa, Joseph Mitchell and Cindy Phillips for useful initial discussions regarding the problem of classifying trajectories. The authors would like to acknowledge the Sandia LDRD program for their support of this work. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

This page intentionally left blank.

Contents

Appendix

Figures

Tables

This page intentionally left blank.

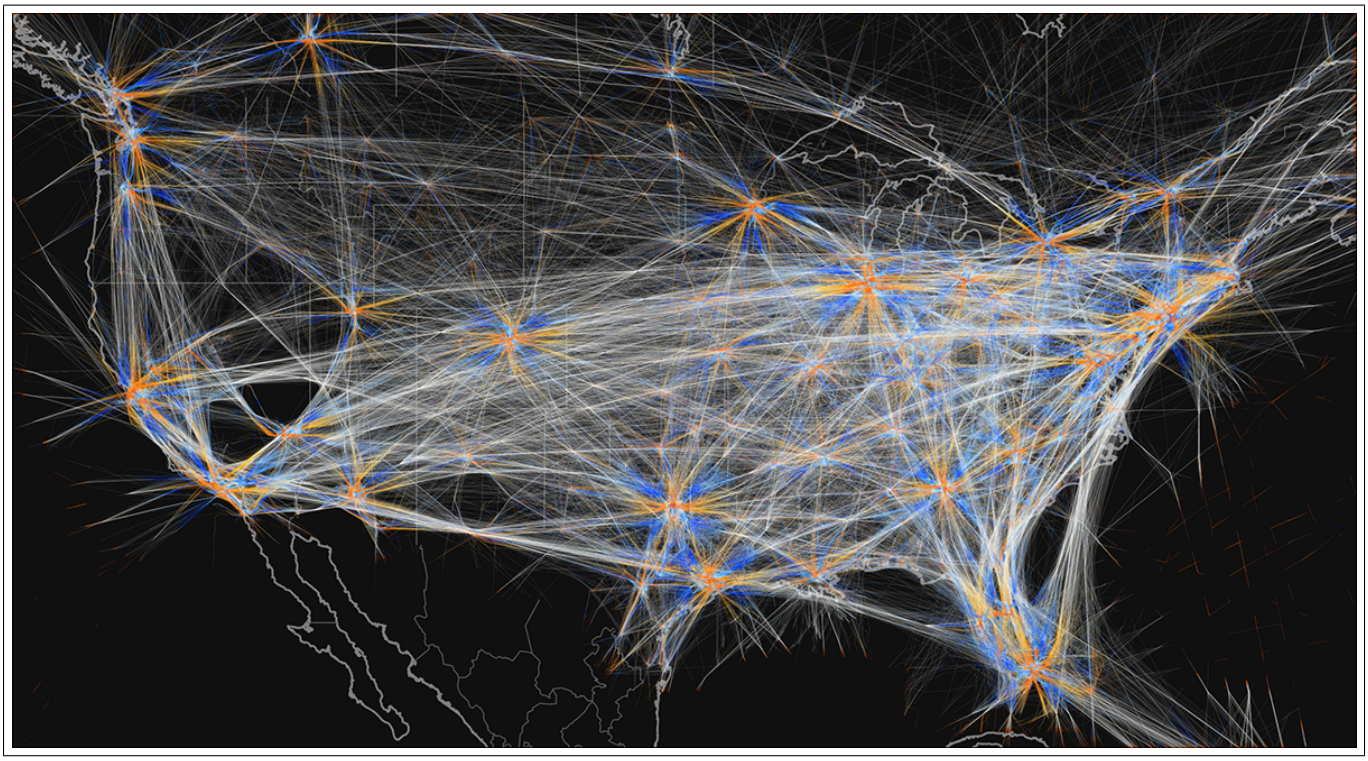


Figure 1. One day of civilian air traffic over the continental United States and Canada. This data set is derived from the Aircraft Situation Display to Industry (ASDI) feed from the US Federal Aviation Administration. An average day's air traffic contains between 40,000 and 50,000 separate flights and over 5 million distinct data points.

1 Introduction

The growth of remote sensing capabilities has resulted in a well-documented explosion of image data[18]. However, interpretation of that data mostly remains a human activity. In recent years we have seen rapid growth not only in image resolution and field of view but also in sampling frequency. This enables an interesting computational analysis problem – trajectory analysis – that is inherently different than the search for large, durable feature changes. Given multiple data captures we can track particular objects, extract their locations, and build up a series of time-stamped positions that compose a trajectory.

Of course, the problem of trajectory analysis is not only of interest in the setting of overhead image analysis. The biology community uses it to examine animal behavior[32]. Molecular dynamics researchers use the trajectories of atoms and molecules to study the behavior and conformations of proteins and polymers[34]. In general, any multidimensional data set that has time-stamped points can be considered a trajectory through phase space.

One difficult but important example that we have chosen to study is the classification of aircraft behavior based on flight trajectories. This problem is important for a number of reasons. First, there are a number of obvious security reasons. It is useful to comb data to search for criminal or terrorist activity. Understanding patterns of both normal and anomalous behavior is critical to optimizing public air traffic resources. Obtaining details of airline performance and analyzing safety issue are also potential applications[5]. Currently, much of the work searching for specific behaviors in trajectories is done manually. If even part of the job of airline trajectory classification could be done in an automated fashion, it could make human analysts much more effective. Figure 1 gives an illustration of how difficult even the manual analysis of a single day's worth of US air traffic would be.

The aircraft trajectory classification problem also has the quality of having a complicated space of input and output. Generally the input consists of time-stamped location and altitude data from which other derived quantities such as speed and heading can be calculated to a certain accuracy. This input is often derived from multiple data sources and has many errors and omissions. The outputs are dependent on the problem of interest. This could include looking for regular patterns, anomalous patterns, patterns that correspond to a specific behavior, clustering into groups, finding a flight similar to an input trajectory or even predicting the destination given the beginning of a trajectory. The outputs described above are not necessarily well-defined and in some cases have a human-defined component to them. The net result of these complexities is a potentially rich set of ways to go about building the model that connects the inputs and outputs.

There have been a number of approaches to the trajectory problem including Fourier descriptors[6], earth mover's distance[11], hidden Markov models[8], Hausdorff-like distances[23], Bayesian models[28] and other approaches. Most of these describe a trajectory in its entirety or compute the distance between two trajectories. We propose an alternative approach based on *trajectory features*. These features have several desirable properties. First, features based on some concise or spatially local property of a trajectory appear to correspond well to how humans envision trajectories. This idea is one of the key drivers behind this work and makes some types of traditional statistical ground truth studies more difficult. Second, most of the descriptors we propose can be pre-calculated *once* for each trajectory, as opposed to proximity measures such as the Hausdorff and Fréchet distances that must be computed *de novo* for every different pair of trajectories. The ability to do precomputation makes our approach suitable for rapid lookup in a database. Finally, for many practical questions of interest that separate flight behaviors, these geometric descriptors correspond fairly closely to one or more quantities that describe the behavior of interest.

In this report we begin by describing some of the related work that has been done in the area of representing and comparing trajectories specifically for aircraft as well as more general work. In Section 3 we describe more carefully the specific problems we are trying to solve by designing geometric measures for aircraft trajectories. We present results and discuss the quality of the different geometric measures in Section 4. We then expand the notion of features and spatial indexing in Section 5 and show how they can be used to predict the destination of a trajectory from a database of historical examples. Examples of using

ensembles of decision trees on the features to do classification are given in Section 6. Finally, we summarize our work and offer suggestions for future work in Section 7.

1.1 Notation

We will use the following conventions when describing trajectories and their features.

- A trajectory $\mathbf{T}$ comprises $n+1$ timestamped points $(\mathbf{x}_0, t_0), (\mathbf{x}_1, t_1), \ldots, (\mathbf{x}_n, t_n)$, where $\mathbf{x}_i$ describes the position of point i.
- Given $\mathbf{T}$, angle θ_i is the turning angle from vector $(\mathbf{x}_i - \mathbf{x}_{i-1})$ to $(\mathbf{x}_{i+1} - \mathbf{x}_i)$. Informally, θ_i is the turn between segments i and $i+1$ in the trajectory. Positive angles indicate counterclockwise turns.
- $|\mathbf{T}|$ is the total length of all the segments of $\mathbf{T}$.
- $||\mathbf{x}_n - \mathbf{x}_0||$ is the *end-to-end distance* of $\mathbf{T}$.
- $\mathcal{C}(\mathbf{T})$ is the convex hull of the points in $\mathbf{T}$. Points $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_m \subseteq \mathbf{x}_0, \ldots, \mathbf{x}_n$ form the vertices of $\mathcal{C}(\mathbf{T})$.
- $\overline{\mathbf{A}}$ indicates the centroid of a polyline $\mathbf{A}$. Thus $\overline{\mathbf{T}}$ is the centroid of a trajectory $\mathbf{T}$ and $\overline{\mathcal{C}(\mathbf{T})}$ is the centroid of the convex hull of $\mathbf{T}$.

Figure 1.1 shows examples of many of the trajectory properties that are used to calculate features.

In the figures where flight trajectories are shown, we use color to indicate the direction of flight. Each trajectory is colored red when it starts and blue when it ends. All distances used in the air traffic examples are great circle distances.

Unless otherwise indicated, the algorithms presented in this paper are applied to the two-dimensional projection of three-dimensional trajectories. In most cases, the extension to higher dimensions is straightforward and in some cases this will be discussed.

1.2 Trajectory Data Set

We tested our algorithms and generated the results shown in this paper using the ASDI (Aircraft Situation Display to Industry) data set. This is an air traffic data set generated by the US FAA (Federal Aviation Administration) that contains most US civilian flights that have flight plans on file. We obtain the data via a subscription through AirNav, LLC, which disseminates the traffic data in XML format along with additional metadata concerning each flight.

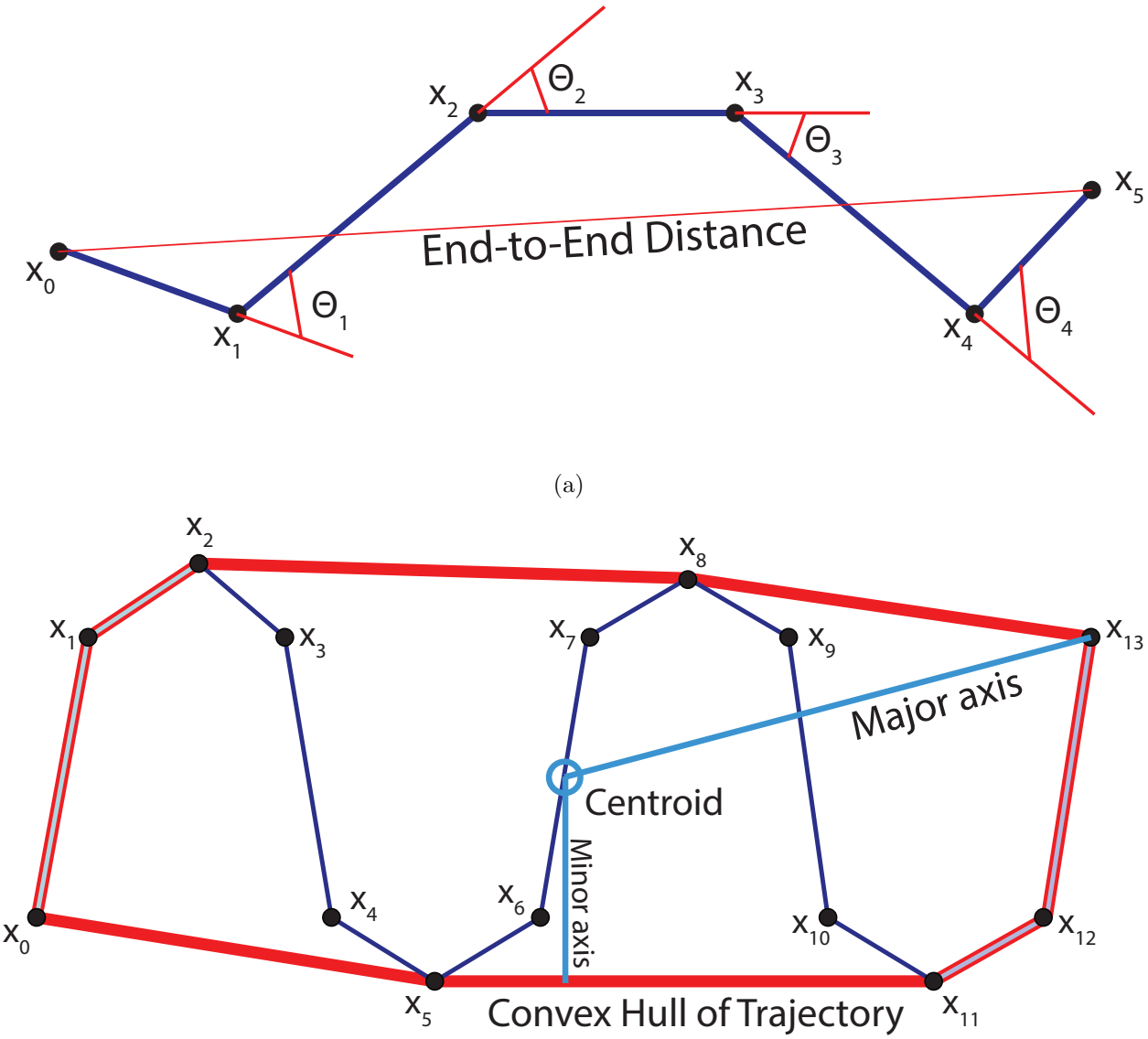


Figure 2. Illustration of the parts and properties of a trajectory that we use to compute features. A trajectory $\mathbf{T}$ comprises $n+1$ points $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_n$. In (a) we see a trajectory $\mathbf{T}$ labeled with its vertices $\mathbf{x}_i$, turning angles $\theta_1 \ldots \theta_{n-1}$ and end-to-end distance $||\mathbf{x}_5 - \mathbf{x}_0||$. In (b) we see another trajectory $\mathbf{U}$ with vertices $\mathbf{x}_0 \ldots \mathbf{x}_{13}$ and convex hull $\mathcal{C}(\mathbf{U})$. We approximate the aspect ratio of $\mathcal{C}(\mathbf{U})$ as the ratio of the lengths of its major and minor axes where the major axis connects the centroid of $\mathcal{C}(\mathbf{U})$ with the most distant point on $\mathcal{C}(\mathbf{U})$ and the minor axis connects the centroid with the nearest point point on $\mathcal{C}(\mathbf{U})$.

The ASDI data set comprises approximately 50,000 flights per day. At present we have approximately 6 months of archived data. Each flight consists of a sequence of data points generally spaced 10-120 seconds apart. Each data point contains a flight ID, a timestamp, position data (latitude, longitude, heading) and a large amount of supporting metadata. Flights generally contain anywhere between ten and several hundred data points. Although the majority of the data points in most flights are uniformly spaced every 60 seconds, the data contains occasional dropouts and irregularly spaced samples depending on contact between an aircraft and the ground sensors that communicate with it.

Metadata in the ASDI data set often includes altitude, speed, departure/arrival airport, departure/arrival times and so on. We will sometimes use this metadata to establish ground truth for testing our algorithms. However, much of the focus of this work is to study how geometric features can be used to compare, contrast, identify and classify flights. We will generally demonstrate these techniques using only geometric information derived from position and time data.

2 Background

2.1 Previous Approaches

The fundamental computer science issues related to comparing two trajectories have been studied for many decades in their most general form. If one considers a trajectory $\mathbf{T} = \{(\mathbf{x}_0, t_0), \ldots, (\mathbf{x}_n, t_n)\}$ to simply be a set of points in a $D+1$-dimensional space, where D is the spatial dimension and the additional dimension corresponds to time, there are a significant number of application drivers outside of aircraft trajectory comparison. These include object recognition, handwriting analysis, and many different forms of time-series analysis.

There have been many different distances defined to measure distance or divergence between two trajectories. Perhaps the most straightforward measure of distance between two curves is the Hausdorff metric[36]. For two trajectories $\mathbf{A}$ and $\mathbf{B}$, the Hausdorff distance is defined as greatest distance from any point on $\mathbf{A}$ to the nearest point on $\mathbf{B}$. This gives a rough sense of the distance between two curves but neglects the direction and speed of travel along both trajectories.

One of the most well-known metrics associated with curve similarity that does take the direction into account is the Fréchet distance. The Fréchet distance $F(\mathbf{A}, \mathbf{B})$ is formally defined as

$$F(\mathbf{A}, \mathbf{B}) = \inf_{\alpha,\beta} \max_{t\in[0,1]} d(\mathbf{A}(\alpha(t)), \mathbf{B}(\beta(t))) \tag{1}$$

where $\alpha(t)$ and $\beta(t)$ are continuous, non-decreasing reparameterizations of $\mathbf{A}$ and $\mathbf{B}$, respectively, onto the interval $[0, 1]$. Eiter and Mannila[21] have extended this definition in a straightforward manner to the case where $\mathbf{A}$ and $\mathbf{B}$ are described by discrete points as polygonal curves. Both variations of the Fréchet distance represent the minimum length of a leash required for a man following one curve to walk a dog that is following the other curve.

One problem that both the Hausdorff distance and Fréchet distance have is that they do not allow for translational, rotational or reflectional invariance. That is, they measure the distance between two curves *given some pre-defined position and orientation.* If the curves to be compared are not already arranged as desired, they must be *aligned* before applying either the Fréchet or Hausdorff distances. This is a difficult problem. Typically one would have to do a Procrustes type of analysis to align them[20] or use an alternate method based on dynamic time warping[10] or edit distance[15] that tries to match geometric distance and curvature between points. Additionally, hidden Markov models have also been used[4] to try to compare and classify trajectories.

2.2 Why Something Different?

The measures described above were primarily designed to do one-on-one comparisons between two trajectories, but for very large-scale work in identifying behavior in trajectories ($> O(10^6)$ trajectories), they become difficult to work with. Many of these distance metrics require $O(ab)$ operations to compute where a and b are the number of discrete points in the trajectories being compared. Furthermore, there is little that can be pre-computed for a trajectory in isolation: every comparison must be computed from scratch for every pair of trajectories being compared. At a more abstract level, these measures operate directly on trajectories as objects in a non-normed metric space. This makes clustering an asymptotically more difficult operation since spatial indices such as *r*-trees and *kd*-trees assume a normed vector space. Finally, the aforementioned measures each compare the entirety of two trajectories instead of identifying and addressing features of interest. What would be ideal is a way to measure similarity based on trajectory characteristics that:

- Can be calculated once for each trajectory.
- Can be calculated for each trajectory in a time that is linear in the number of trajectory points.
- Can be used to calculate similarity between two trajectories in constant time.
- Can be used efficiently to cluster trajectories.
- Can have translational, rotational, and potentially scaling and reflection invariance properties.
- Is based on characteristics of the trajectories that can effectively categorize behavior.

Our approach is to use simple scalar measures associated with each trajectory (such as time, total distance, etc.), and combine those values with *geometric* scalar quantities that describe the relevant geometric characteristics of the trajectory. This gives us a feature vector associated with each trajectory that can be used to store information about, and do comparisons between different trajectories. These comparisons between feature vectors can be done through a specifically defined vector product that can be done in a time that is constant with respect to the length of the trajectories themselves. These features can also be used in traditional databases or specially-designed database machines to do lookups very quickly on very large databases.

3 Problem Definition

We define here more precisely what we mean by trajectory comparison. There are a few different types of problems that involve trajectory comparison. Some of the more important ones that we will cover are

- Can we find the trajectories in a database that are most similar to a given trajectory?
- Can we find trajectories that exhibit a behavior of interest without regard to translation, rotation or scale?
- Can we divide trajectories into specific clusters?
- Can we find trajectories that are outliers with respect to a given set of trajectories?
- Can we classify trajectories using unsupervised learning techniques
- Can we predict trajectory destinations given a set of historical training data.

In order to solve these problems using the geometric feature vector approach, we have to define the quantities that will be useful to construct the feature vector. Although there are a very large number of features that one could choose, we focused on features that generally were wholly or mostly insensitive to variations in sampling of points along the trajectory. This allows comparisons between data sets where the points are sampled at different rates or in a non-uniform manner. The features fell into a few different categories that are described below.

3.1 Distance Measures

These measures include many straightforward measure associated with the flight and include:

- End-to-end distance of the flight:

$$d_e(\mathbf{T}) = ||\mathbf{x}_n - \mathbf{x}_0||$$

- Total distance traveled (length of trajectory):

$$d_t(\mathbf{T}) = \sum_{i=0}^{n-1} ||\mathbf{x}_{i+1} - \mathbf{x}_i||$$

- Distance from a given fixed point or set of points

- Centroid of points:

$$\overline{\mathbf{T}} = \frac{1}{n}\sum_{i=0}^{n} \mathbf{x}_i$$

The first two of these measures are simple but important ones for characterizing flights, while the third can be calculated for more specific concerns related to relevant fixed points on the ground. Note that the fourth, along with similar measures defined later, consist of two values defining a position (usually by longitude and latitude) and not just a single value.

3.2 Heading Measures

We can also define measures associated with how straight a flight is such as:

- Total curvature:

$$c_{total}(\mathbf{T}) = \sum_i \theta_i$$

- Total turning:

$$c_{abs}(\mathbf{T}) = \sum_i |\theta_i|$$

These measures turn out to be very useful either by themselves or in conjunction with other measures to separate out different types of flights. In some cases, one would potentially be interested in quantities such as an average curvature associated with a flight. However, care must be taken not to introduce a sampling bias into the quantities by taking the average over the number of points. Instead, an average over total flight distance would be more approriate.

3.3 Geometric Measures

These more sophisticated measures often say more about the shape of the flight than the more basic measures listed above and are key to some of the results later in the paper. These measures include

- Area covered by flight, defined here as the area of the convex hull of the flight points.
- Aspect ratio of the convex hull of the flight. This is defined as the ratio of the shortest to the longest axis of the polygonal convex hull of the points. We approximate the length of the shortest axis as

$$\min_{\mathbf{c}\in\mathcal{C}(\mathbf{T})} ||\overline{\mathcal{C}(\mathbf{T})} - \mathbf{c}||$$

or in words, the distance from the centroid of the convex hull to the nearest point on the convex hull. This includes *any* point on the convex hull, not just the vertices. The length of the longest axis is defined as

$$\max_i ||\overline{\mathcal{C}(\mathbf{T})} - x_i||$$

for x_i on $\mathcal{C}(\mathbf{T})$. In the case of the furthest distance from the centroid, we only need to consider the vertices of the convex hull because the the convexity property of the hull.

- Length of the perimeter of the convex hull.
- Centroid of convex hull $\overline{\mathcal{C}(\mathbf{T})}$.
- Ratio of end-to-end distance traveled to total distance traveled:

$$\frac{d_e(\mathbf{T})}{\overline{d_t(\mathbf{T})}}$$

.

- Radius of gyration of the points:

$$\sqrt{\frac{1}{n}\sum_i (\mathbf{x}_i - \overline{\mathbf{T}})^2}$$

We also believe that the geometric measures described above seem to capture more holistic views of the trajectories and correspond closely to how humans view the trajectories. However, this work will not examine this hypothesis and detailed comparisons to human studies will be left to future work.

We will also use one final geometric measure based on the concept of *distance geometry*[17] to describe complex shapes in more detail. We define it as follows. First, parameterize a trajectory uniformly over the interval $t \in [0, 1]$. Then choose a set of m intervals (t_{m_1}, t_{m_2}) and measure the distance between the corresponding points $\mathbf{x}_{m_1}$ and $\mathbf{x}_{m_2}$. This set of m values then can be used as geometric measures to describe the shape of the trajectory. These m values represent a geometric measure that is invariant to translation, rotation and reflection. Further, if we normalize these m values by the largest value so that that all of the values are between 0 and 1, we obtain a measure that is also *scale invariant*. The use of distance geometry as a feature has the advantage that it is a somewhat universal descriptor of shape and is especially useful when it is not known *a priori* what features would be best. Its primary disadvantage is that it tends to carry less information than more specific features that can be calculated when there is an understanding of what features would best identify the qualities of interest in a trajectory.

3.4 Use of Spatial Indexing

The feature vector representation enables two different approaches to solve the problems listed above. The first is the most straightforward. We can calculate the feature vectors and

then use traditional searching or clustering algorithms using a distance metric defined by the feature vectors.

However, there is another approach that turns out to be faster and more general for some applications. If we choose the feature vector carefully and build a distance metric on those vectors that is expressible as an L^p norm, then we can use a spatial indexing scheme such as an *R-tree*[24] to store feature vector values, search for nearest neighbors, and even do clustering. We will demonstrate instances in search and prediction where an *R-tree* was used to perform indexing that would have otherwise be too computationally expensive.

4 Filtering and Search Results

4.1 Data Cleaning and Trajectory Assembly

The data points in the ASDI feed arrive sorted by timestamp rather than by flight. Our first task was to reorganize this stream into potential trajectories. We first sort by flight ID to create streams belonging to different flight IDs then search each stream for large time breaks between points that indicate multiple stops under a common flight ID. For this section we used a threshold value of 30 minutes to identify these breaks. Values between 20 and 60 minutes did not yield significantly different results.

Once we assembled these candidate trajectories we ran each one through a simple cleaning operation to remove obviously bad data. We looked for and removed data points that were an unreasonably large distance away from their neighbors given the time separation between them. In this case, "unreasonably large distance" required an airspeed 3-10 times faster than a typical airplane. This was sufficient to remove the especially bad points. There are certainly more sophisticated cleaning and filtering operations available. We chose not to use them because we want to test our measures for robustness against data that may contain significant uncertainty or noise in the position fields.

4.2 Simple Geometric Filtering

The first examples we show here are primarily intended to test some of the more straightforward aspects of geometric search and were computed by single passes through data sets looking for specific values of parameters that represent a given type of behavior.

Avoiding Airspace

One possible question that we could ask regarding a collection of flights is, "Is there a section of airspace that flights seem to avoid?" A geometric signature corresponding to such a question could be described in a number of ways. A simple way would be to look at flights that traveled a significant distance (in order to exclude flights that are simply flying circles as part of training), but traveled a distance that was significantly larger than the distance between their take-off and landing points. Furthermore, to exclude flights that simply meander, one could put a constraint on the aspect ratio of the convex hull, requiring the flights to be more "round". These criteria turned up a sizable cluster of flights on July 10, 2013 shown in Figure 3. Upon further research we found out what the flights were avoiding. That day, many flights were rerouted to avoid a large system of thunderstorms that swept eastward through Illinois and Indiana all the way to Ohio and Pennsylvania. In Figure 4 we display the "avoiding" trajectories again along with a weather map from that day.

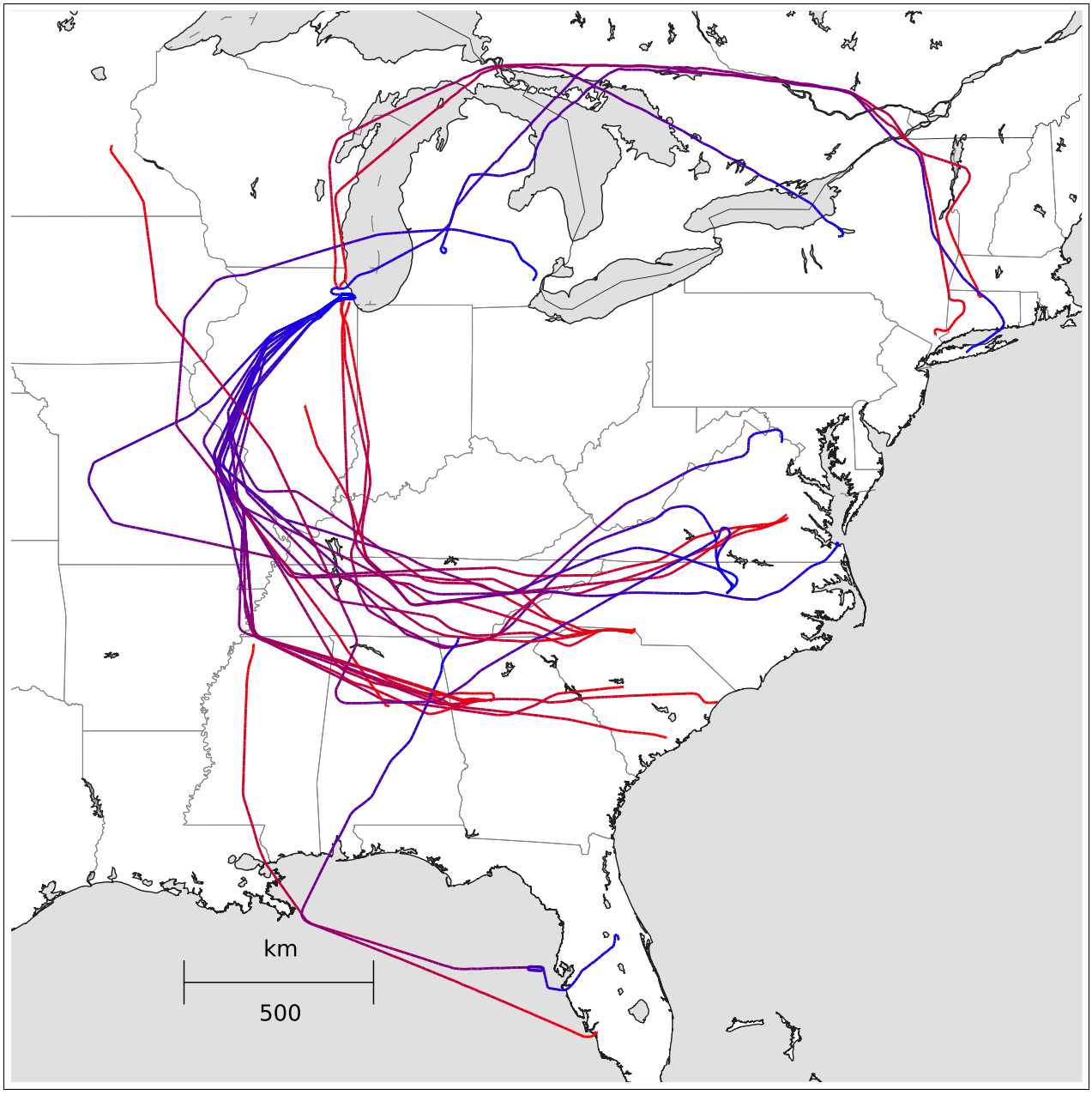


Figure 3. Examples of flights found for the "avoiding" specification. In this case, we required the end points of the flight to be at least 1000 kilometers apart, the ratio of the end-to-end distance of the flight to the total flight distance to be less than 0.7, and the aspect ratio of the convex hull to be at least $\frac{1}{3}$.

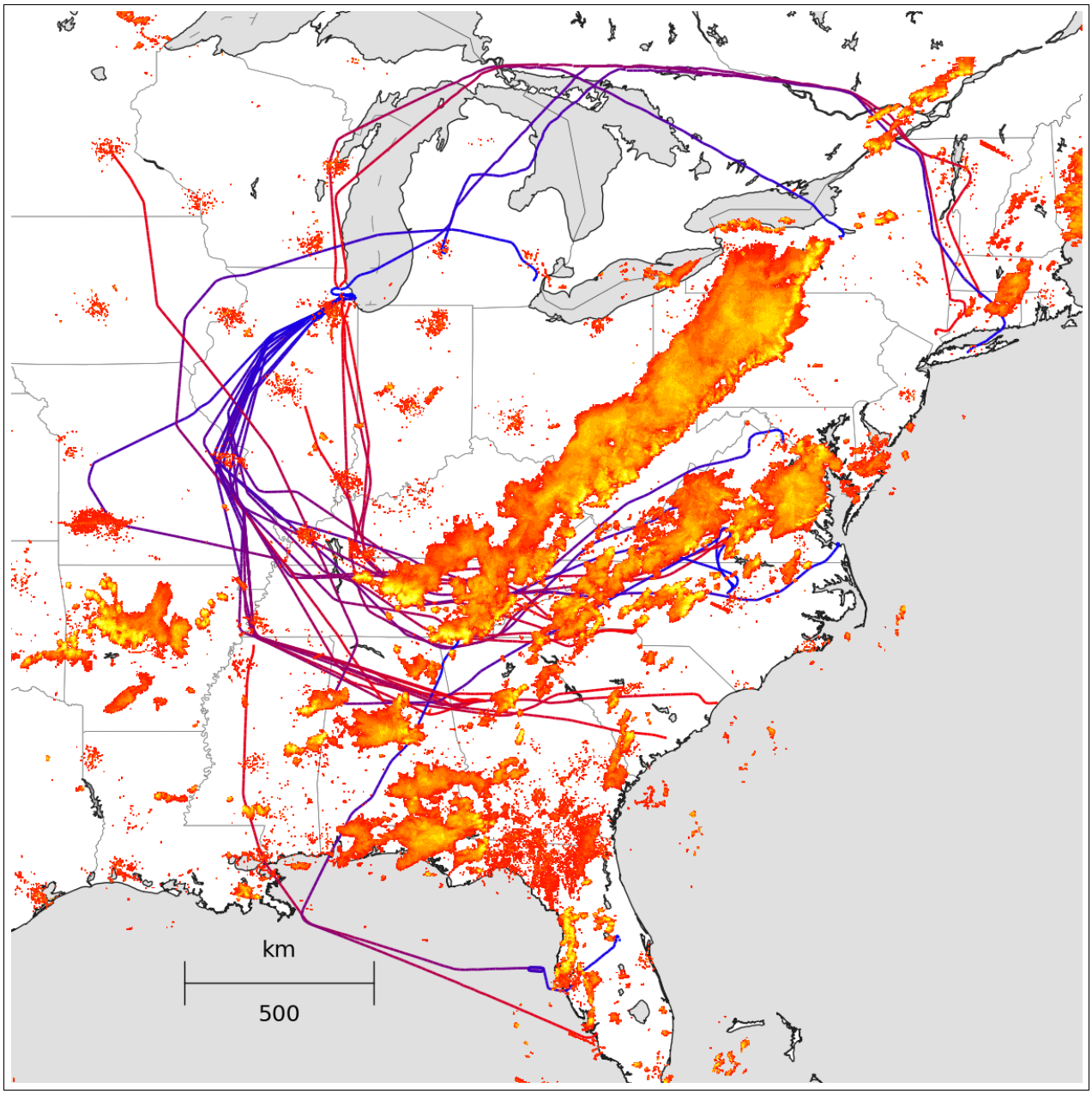


Figure 4. Most of the trajectories identified with the "avoiding" specification were responding to this event: a severe weather system crossing most of the midwestern United States from Illinois to New York. This weather map was captured at 8:30PM Eastern Daylight Time (UTC-5) on July 10, 2014.

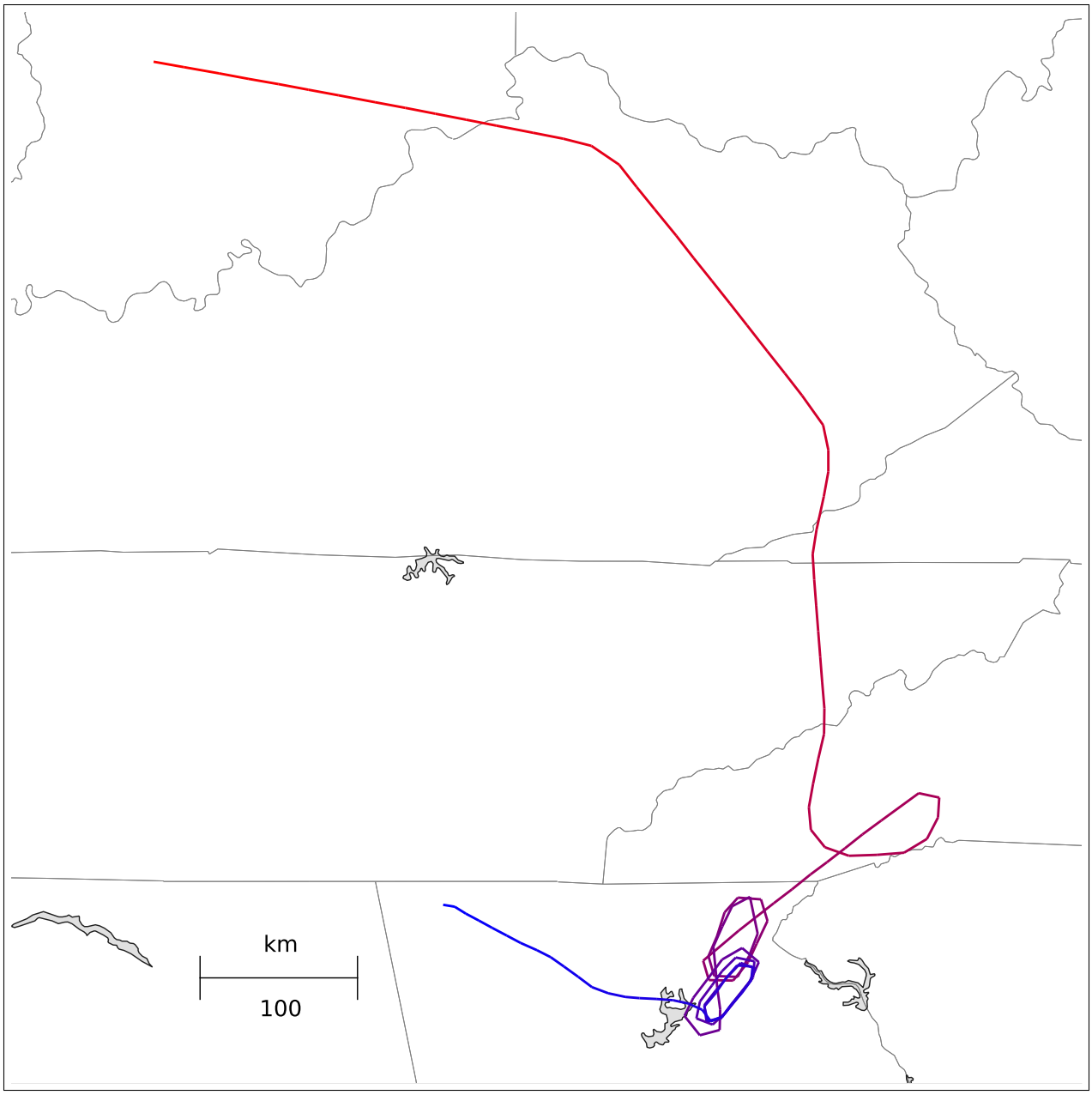


Figure 5. Examples of a flight found for the "holding and diverted" specification. In this case, we required the end points of the flight to be at least 200 kilometers apart, the total amount of turning to be at least 20π radians, and the aspect ratio of the convex hull to be at least $\frac{1}{10}$.

Holding Pattern

Another distinctive pattern of interest in flight trajectories is a holding pattern. We define this as a flight that flies for some distance and then enters a circling pattern due to some sort of landing delay. We translated this into two geometric constraints. First, the flight had to have at least moderate length (200km) and a significant total curvature that would be unusual for a point-to-point flight (at least 20π).

This search returned many flights that had clearly been instructed to circle while awaiting permission to land. We decided to extend it for a more difficult test of our approach to search for flights that entered holding patterns and were ultimately diverted to different airports. To accomplish this, we added the constraint that the aspect ratio of the convex hull of the trajectory must have an aspect ratio of at least 0.1. This value was chosen to elimate what were essentially straight flights that paused on approach to the original airport. This search turned up one flight in our test data set (see Figure 5). When we examined the original metadata for this flight we found that it was indeed inbound to Atlanta when it entered a holding pattern and was finally diverted to Chattanooga in early June of 2013.

Mapping Flights

Given the advances in imaging technology and the burgeoning business in on-line map services, there are a significant number of planes flying in a back-and-forth scanning, or boustrophedon, pattern. This type of flight will have a significant length, but will be enclosed by a fairly compact shape. For this search, we require a reasonably long total distance, but a more compact shape than a straight flight. An example of these flights is shown in Figure 6. Because the distinctive pattern of these flights was relatively easy for the human eye to pick out, they were used to create ground truth data for a sensitivity calculation that is described in detail in subsection 4.4.

4.3 Distance Geometry

The distance geometry approach described in Section 3 deserves a more in-depth discussion. Most of the features that have been used so far are those that are obviously relevant for the problem of interest. This is a useful approach when there is an understanding of the specific pattern of interest. However, there are many cases where the problem of interest might involve finding shape similarities that are difficult to explicitly describe. In this case, using a generic set of intra-trajectory distances to describe the shapes turns out to be a powerful means of letting the computer determine similarity without the user defining it.

To demonstrate the distance geometry technique, we will use one of the flights that was found above in the avoiding airspace example above (Figure 3). While the goal in that example was achieved by describing the features, distance geometry enables an even simpler solution. We begin with the flight shown in Fig 7(a), measure distances at various points

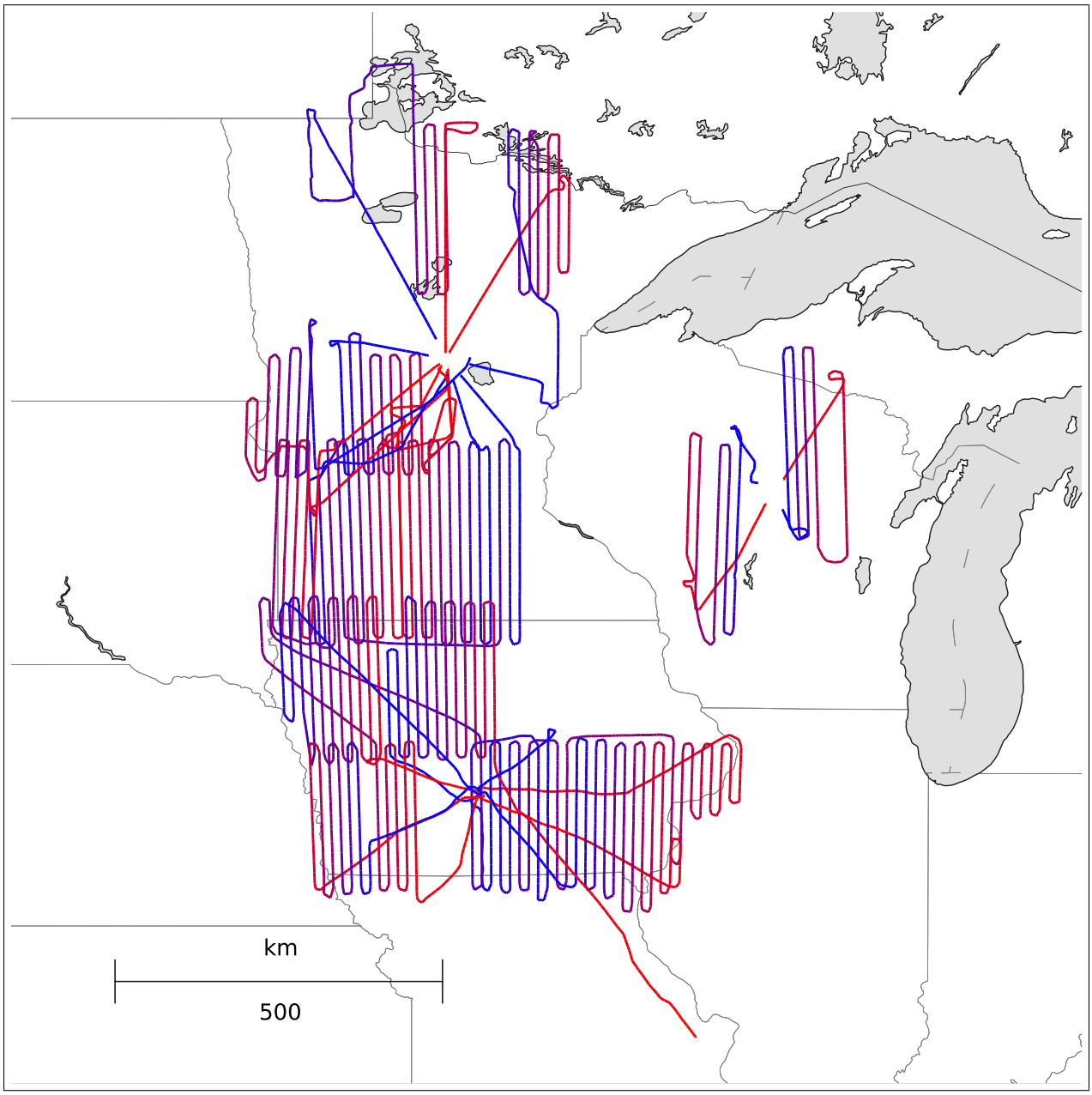
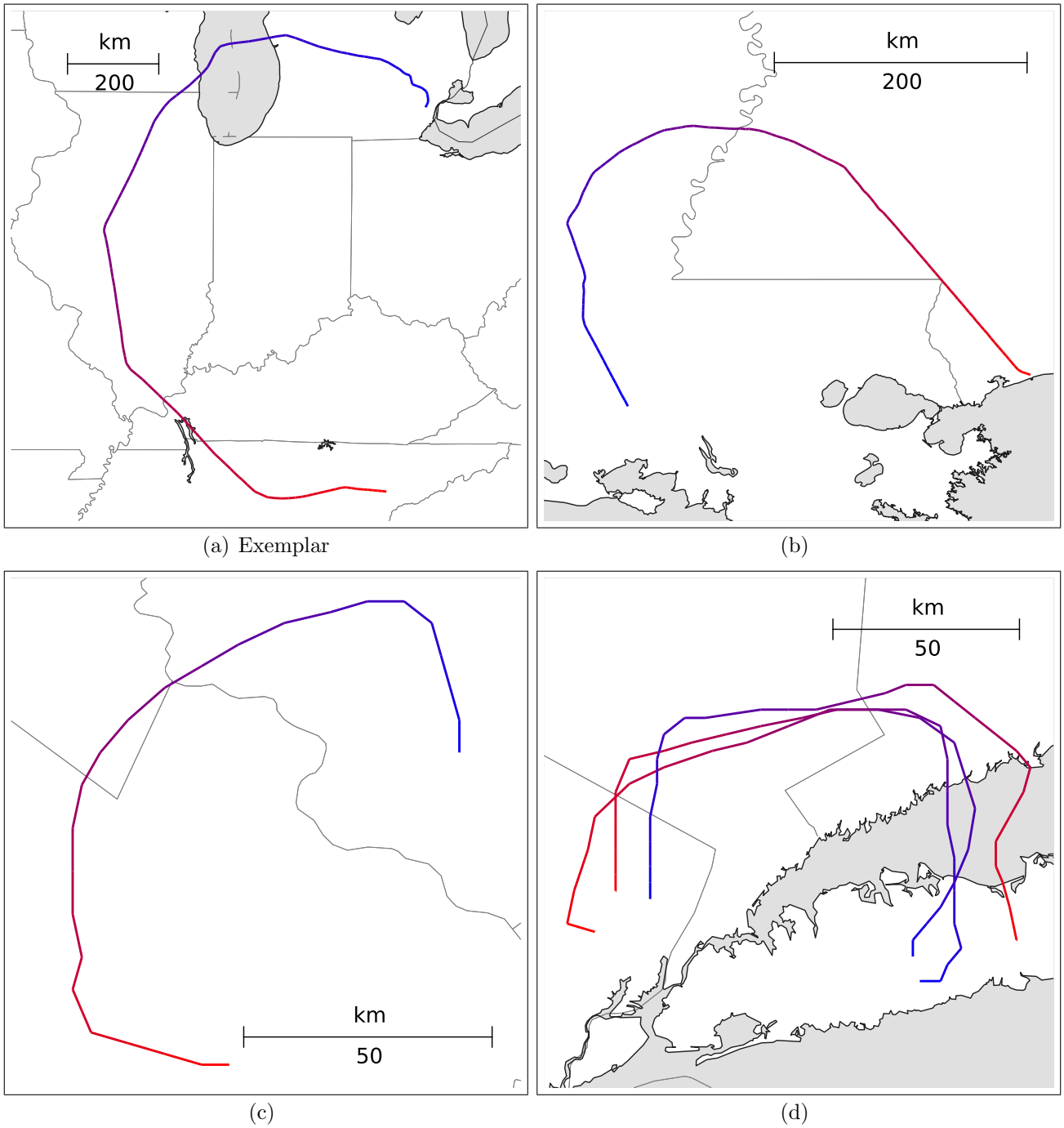


Figure 6. Examples of flights found for the mapping criteria. We require the flights to be longer than 255 miles, have a convex hull aspect ratio greater than 1/20, and require at least 4 180 degree turns. A selected sample of the results are shown in the figure.



(a) Exemplar (b)

(c) (d)

Figure 7. Examples of curve matching using the distance geometry algorithm. The curve to be matched is shown in (a). Two examples of matched curves are shown in (b) and (c), although at very different scales than (a). The curve in (b) flies around the southern Louisiana area, while the curve in (c) flies around Washington, DC. Finally, in (d), we see examples of flights diverting around New York City, in both directions.

along the flight, and build a feature vector with the distances normalized to fall between 0 and 1. This gives us a feature vector based solely on the relative distances between different points in the trajectory. We then compare this feature vector to those from other flights in the database using the L^2 norm to find flights with a similar shape.

In our example, we chose 10 different distances to use as the intratrajectory distances. Let $\mathbf{T}(t)(t \in [0,1])$ be the entire trajectory parameterized by t. Let $d : \mathbb{R}^2 \times \mathbb{R}^2 -> \mathbb{R}$ be a distance function between points (here the familiar Euclidean distance). We then define the following distances as our features:

- End-to-end distance: $d(\mathbf{T}(0), \mathbf{T}(1))$
- Distances from midpoint to beginning and end: $d(\mathbf{T}(0), \mathbf{T}(\frac{1}{2}))$ and $d(\mathbf{T}(\frac{1}{2}), \mathbf{T}(1))$
- Thirds: $d(\mathbf{T}(0), \mathbf{T}(\frac{1}{3}))$, $d(\mathbf{T}(\frac{1}{3}), \mathbf{T}(\frac{2}{3}))$, $d(\mathbf{T}(\frac{2}{3}), \mathbf{T}(1))$
- Quarters: $d(\mathbf{T}(0), \mathbf{T}(\frac{1}{4}))$, $d(\mathbf{T}(\frac{1}{4}), \mathbf{T}(\frac{1}{2}))$, $d(\mathbf{T}(\frac{1}{2}), \mathbf{T}(\frac{3}{4}))$, $d(\mathbf{T}(\frac{3}{4}), \mathbf{T}(1))$

While we could have estimated the distances at the precise time points through interpolation between the nearest discrete points, we simply chose the points closest to the interval boundary under the assumption that the points were roughly equally spaced. This made the lookup very fast and did not significantly change the outcome compared to precise interpolation between points. However, we have implemented an approach that allows easy interpolation when it would significantly affect the results.

The results for that the distance geometry search are shown in Figure 7. There were a wide variety of results, all with similar fundamental shapes but with a wide variety of sizes and orientations. We had also originally attempted these comparisons with curve alignment algorithms that were based on dynamic programming techniques. Those approaches took much longer due to their increased computational complexity and failed to match the global shape of the curves due to their focus on aligning local structures.

It is important to note that while the distance geometry approach was effective in some types of search and more mathematically satisfying than many of the ad hoc features, it actually performed worse than using ad hoc features when the ad hoc features were able to be chosen to more directly capture the behavior of interest. For many of the practical problems we have studied, the use of more specific features was much more effective as they contained much more information about the classication of a trajectory into a specific category.

4.4 Effectiveness of Feature Space Approach

Generally, the testing the effectiveness of an new approach to classifying data involves applying the approach on a data set where the ground truth is well known. This is especially

difficult with the technique we have proposed and the data set that we are using. First, this technique is fundamentally based around finding similar shapes, and there is a continuum of potentally similar shapes and ways to measure similarity. Second, no ground truth is available *a priori*. Moreover, the data are not labeled in a way that useful shapes or behavior can be extracted automatically. We must find our ground truth by hand. Given the sizes of the inital data set of observed points (approximately 1-10 GB each), this is a very labor intensive process.

We performed one sensitivity analysis on the task of finding the mapping flights that were disucssed in subsection 4.2. The process for establishing ground truth was as follows:

1. We began by setting broad search parameters that we believed would be inclusive of almost every mapping flight in the data of interest.
2. We then studied those results and extracted by hand what we believed to be flights with a behavior of interest ("mapping"). Note that this was not a very clearly defined behavior and in some cases the acceptance or rejection of a flight involved a somewhat arbitrary analyst decision.
3. We then took the aircraft IDs of the flights that had been previously identified as mapping flights, and searched through the entire database for additional flights from those aircraft that had not been previously found as mapping flights.
4. All of the flights from any plane that had been identified as a mapping aircraft from that data set were then considered mapping flights. This was then considered our ground truth data set.

We note that the procedure above has its own false positives. For example, if aircraft A flies a mapping trajectory in one part of the country, our approach of including all of A's traffic in ground truth will necessarily include the segment where A travels from one part of the country to another in order to map some other territory. Our data set was too large to manually remove those. In the absence of an infallible oracle, we chose to draw the line as described above.

We chose as our data set all of the flights in the July 2013 ASDI data set that were non-scheduled commercial flights and that met the criteria for having enough data to construct a reasonable trajectory. This data set consisted of 330,492 flights, with 819 being labeled as true positives for classification of mapping flights.

To establish parameters for what we would consider mapping flights, we studied one day's worth of mapping flights, as this was a good surrogate for how such a methodology might be used in practice. We chose 3 features as identifying the mapping flights. These were

- A flight length of at least 0.04 radians (approximately 255 km)
- A convex hull aspect ratio of at least 0.05 (remove straight flights)

		Ground Truth	
		True	False
Test outcome	Positive	331	1120
	Negative	488	328553

Table 1. Confusion matrix for mapping flight search.

- 4 examples roughly "turning around" behavior, defined by an approximately 180 degree change in heading over a relatively short distance

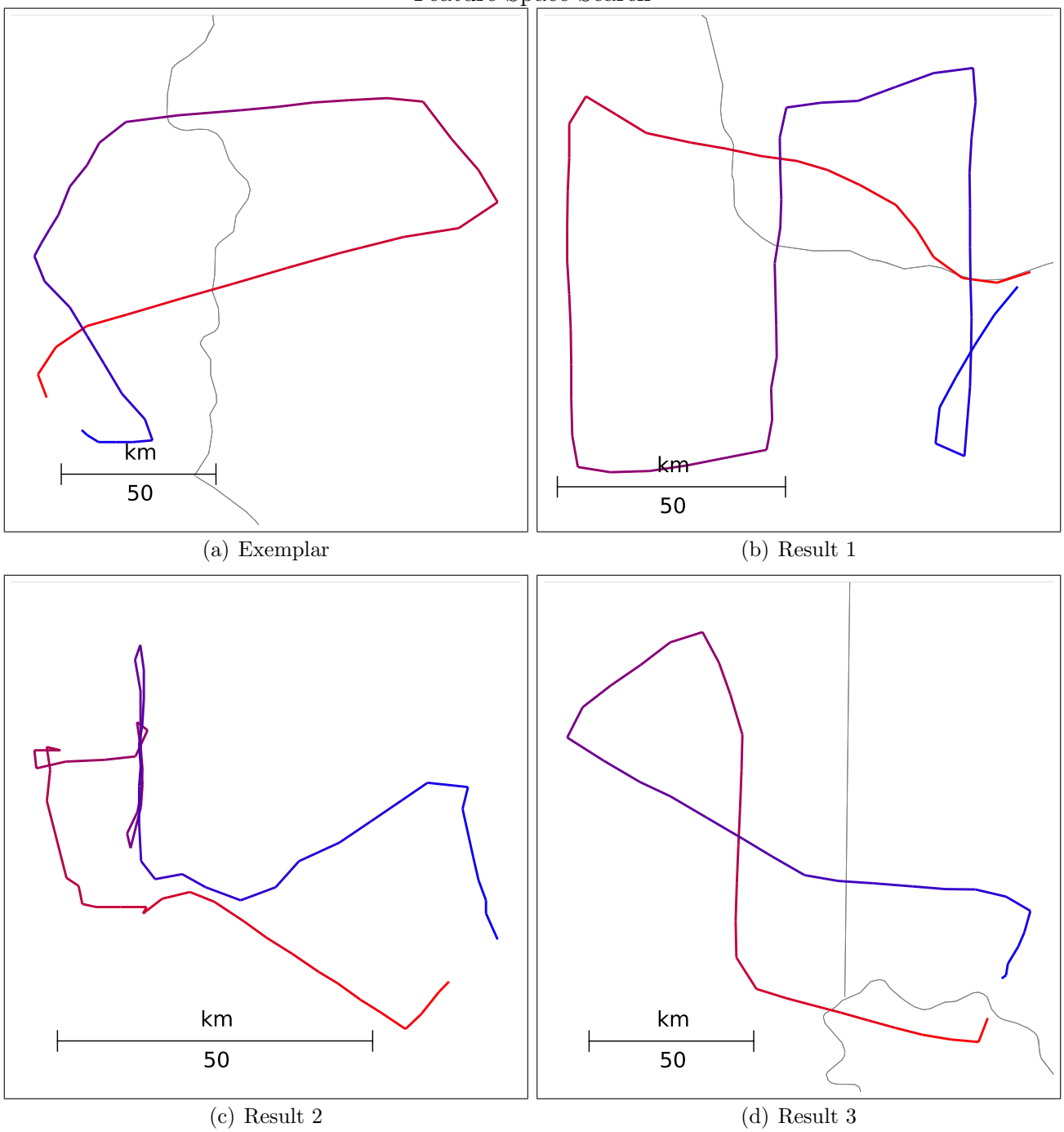
The results are shown in Table 1. The algorithm identified 331 of the true positives and had a total of 1120 false positives. This corresponds to a sensitivity of 40% and a specificity of 99.66%. Given known errors in the ground truth set and a somewhat subjective definition of flights that counted as true positives, this demonstrates an effective way of finding a significant fraction of the flights of interest using the feature vector technique.

4.5 Other Advantages of Feature Space Approach

We have just demonstrated finding trajectories with a certain shape by calculating a feature vector for an exemplar and then comparing that exemplar to all trajectories in the database one by one - an $O(n)$ search with respect to the size of the database. This becomes expensive as the database grows to millions or billions of trajectories, especially since each search has to be computed from scratch. An approach that would allow us to re-use calculations is to create a spatial index within multidimensional feature space that will allow us to search quickly for nearby flights.

There are many types of data structures for this type of spatial indexing, including *k-d trees*[9] and *R-trees*[24]. These hierarchical structure allow in most cases for logarithmic time search and insertion. If the specific characteristics required for comparison are known a priori, a multidimensional space of those geometric features can be populated with the database of flights and finding "similar flights" becomes a neighbor search that is simple to do on the tree structure.

As an example of this, we demonstrate a somewhat more sophisticated search. We start with the flight shown in Figure 8(a), a roughly figure-eight shape which is somewhat unusual among the flights in our database. It is more difficult to write a feature descriptor for this flight. Instead of writing the descriptor directly, we define the different dimensions of the feature space to be features that we guess will be relevant. For this test we chose three features: the total distance, the ratio of the end-to-end distance to the total distance, and the aspect ratio of the convex hull. We built an index over approximately 50,000 flights (about 1 day's worth) and asked for the 10 closest points in feature space. The flights corresponding to three of the closest points are shown. Given the small dimension of the feature space, some



(a) Exemplar (b) Result 1

(c) Result 2 (d) Result 3

Figure 8. Examples of curve matching using feature space search. The curve to be matched is shown in (a). The dimensions in the feature space here represent total distance, the ratio of total distance to end-to-end distance, and the aspect ratio of the convex hull. The 10 nearest-neighbor points in the feature space were searched for, and 3 of the results are shown.

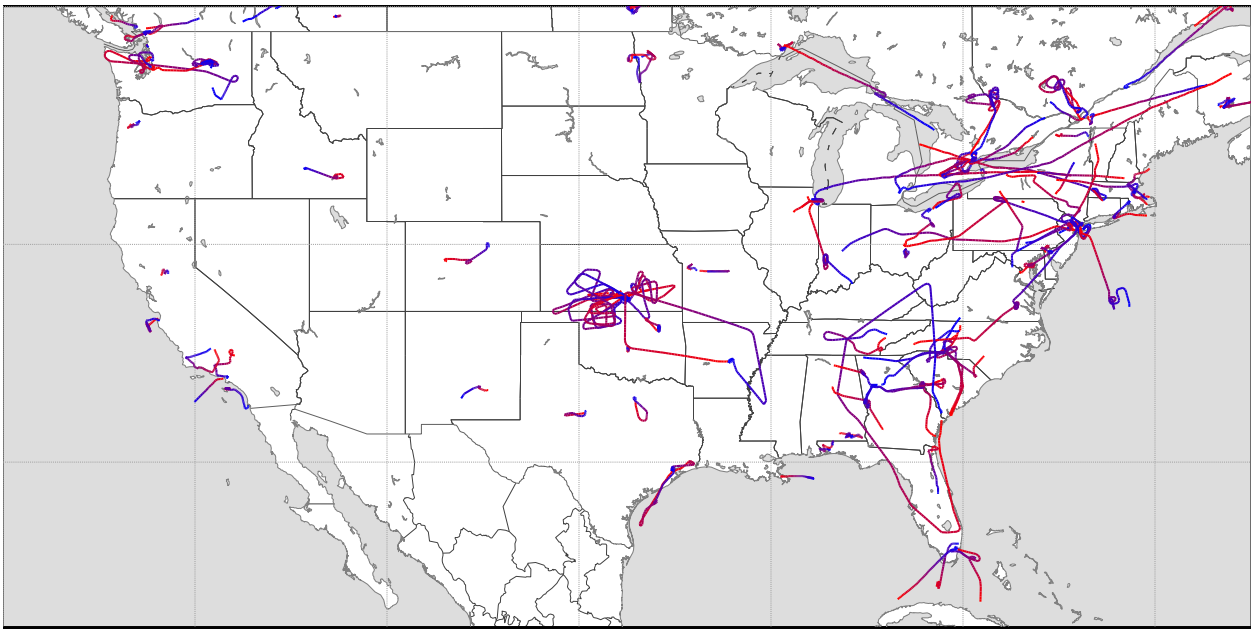


Figure 9. A sample of 100 outlier trajectories discovered using DBSCAN clustering within feature space. Since DBSCAN defines outliers as "points that are not part of any cluster" this set was a natural consequence of the same clustering operation used earlier.

of the other neighbors did not resemble the figure-eight shape as closely. On an interesting note, we can also search for the flights that are "furthest" away from the test flight above. In this case, the 10 flights furthest away were all long, straight trans-Atlantic flights.

Representing the data as feature vectors in a normed vector space also allows clustering to be done in a number of different ways. There are a variety of traditional dimensionality reduction techniques that project data down from a high dimensional space to a two-dimensional space so that clusters can be found through visual inspection or by existing algorithms.

Finally, the feature space embedding enables an elegant solution to a difficult problem: finding trajectories that are outliers with respect to a set of other trajectories. Through the feature space embedding method, one can search for individual trajectories or small clusters of trajectories that do not have many nearby neighbors. This gives a quantitative definition of the notion of an outlier or outliers with respect to a set of trajectories and their respective features. Figure 9 shows a collection of examples.

5 Prediction Results

Given the feature space representation of the trajectories, we can also develop algorithms capable of efficiently predicting the eventual destination location of an unfinished trajectory given a database of historical trajectories. We make no assumption about potential constraints on the routes, such as a network of roads, and we desire to be able to quickly obtain results given a data set of millions or even hundreds of millions of trajectories.

Predicting destinations is applicable to many problem domains. For example, with accurate destination prediction a user's smart phone could determine where the user is likely traveling, and inform them of their estimated time of arrival and/or alert the user of potential issues (i.e. dynamic traffic alerts) or opportunities along the route. In addition, if a hybrid vehicle were able to accurately predict its expected route and destination, its energy usage could be optimized by, for example, delaying a battery recharge until regenerative braking could be used on an upcoming hill [19]. One could even imagine trajectories being extracted eye-tracking data (time-indexed positions of where a user is looking on a screen), and predicting in real-time to which position on a screen a user's eyes were moving.

Our approach exploits two characteristics exhibited by many types of trajectory data. First, a very similar route is generally taken from a specific origin to a specific destination. Second, most new trajectories are very similar to a previously seen trajectory, given enough historical data [22]. Given these facts, to predict an expected destination we first generate multiple feature vectors in a high-dimensional space for each historical trajectory. Then, nearest neighbors are found, in the feature-space, for the unfinished trajectory whose destination we wish to predict. Confidence values are assigned to each of the most similar trajectories, and finally the probable destination location is determined.

We use a large database of aircraft trajectory data to demonstrate the efficacy and efficiency of this method for predicting trajectory destination locations. Although our method is general enough to work on other types of trajectories, analyzing air traffic allows us to experiment with a large-scale real-world database, where origin and destination locations are generally well-defined.

The remainder of the section is organized as follows. In the next subsection we summarize several works related to trajectory destination prediction. Next, in Subsection 5.2 the problem we desire to solve is formally defined. Subsection 5.3, then, details each stage of our method, and in Subsection 5.4 various results of predicting destinations with our method are discussed. Finally in Subsection 5.5 we describe some opportunities for future work.

5.1 Related Work

Trajectory destination prediction has been studied in several different contexts, and using several different methods. Some of the most relevant research is summarized below.

Froehlich and Krumm describe their route prediction system in [22]. They use a version of the Hausdroff distance to compare and Dendogram clustering to merge cleaned trip trajectories into common routes. Using similar methods, partial trajectories were then compared with the routes to obtain a predicted destination. The method was tested on a database of automobile trajectories collected from 250 drivers. The results show that half way through a trip, the future route can be correctly predicted 20% of the time, and the correct route is in the top 10 matches 40% of the time. Considering only repeat trips, the same values jump to 40% and 97% respectively. Because this method must compare every pair of trajectories in the input (n^2 comparisons), it will likely not scale well to large data sets, such as the hundreds of thousands of aircraft trajectories we desire to analyze.

In [29] Krumm and Horvitz detail Predestination, a system for inferring destinations from partial trajectories. They utilize Bayesian inference to produce a probabilistic map of potential destinations, which results in a median error of 3 to 5 km in predicting the destination at the half-way point of a trip. Patterson et al. [31] applied machine learning and a particle filter to GPS traces to predict a person's destination, future route, and even mode of transportation. This method is able to make good predictions for a few blocks into the future in an urban environment, but unfortunately does not make accurate long-term predictions. Simmons et al. [39] rely on information about a road network, in their task of predicting driver routes and destinations. They generate a Hidden Markov Model (HMM) of the routes and destinations visited by a driver, and then use the model to predict future behavior. Our desire, however, is a more general method which does not rely on information about an underlying network of roads or airways.

Researchers at Motorola [40] have also developed a system for learning routes from a users travel data. Learned routes are stored in a database which can then be queried to predict the destination of a current route. Based upon the predicted destination and future route, appropriate traffic advisories can be sent to the user.

Chen et al. [16] developed a system for predicting future routes and destinations, but rather than focus on vehicular trajectories, they collected real personal movement data from a small set of participants, and mined this data to enable accurate predictions. Movement data was first mined to extract a set of significant places from which destination and origin locations could be predicted. Next, the system extracted abstract movement patterns from the training data. Finally, in a separate module the movement patterns were used to construct a pattern tree, that is traversed to find matches to the live movement data. One focus of their work was to predict both the future route and the destination in a unified manner, unlike previous approaches for which the predicted route may not match well with the predicted destination. The researchers showed their method could predict the destination of a user with 79.6% accuracy. Our method, however, is more general and does not seek to handle or utilize the unique qualities of personal movement data.

5.2 Prediction Problem Definition

The problem we desire to solve is defined as follows. Given a large-scale data set of historical trajectories, and without explicitly assuming anything about the geographical limits placed on those trajectories (such as a road network), how can the eventual destination of a new unfinished trajectory be accurately and efficiently predicted? One key aspect of our problem is the scale. We desire to be able to train using a database of at least 1 million trajectories each made up of 100 points, on average. Our training should be efficient, and not require direct comparisons between every trajectory. Training should only require hours, and predicting a single destination should take a fraction of a second. Finally the prediction results should be relatively accurate.

5.3 Our Approach

Our method predicts destinations by advancing through a series of stages, including data cleaning, feature vector creation, R-tree building, similar trajectory finding, and analysis. Each of these stages are described in detail below.

Cleaning The Prediction Data

As described in Section 1, the ASDI data is not organized or divided into individual flights. We first group the ASDI data into position reports with the same call sign, but further refinement is necessary, because the same call sign may be used for a similar flight each day, or even different flights during the same day. A heuristic is used to separate position reports for a given call sign into individual flights. The separation criteria consists of three components: the maximum separation time between consecutive position reports, the maximum separation distance between consecutive position reports (unused for the analysis below), and the minimum number of position reports for a flight. A time-sorted list of position reports for a given call sign is split into separate flights where a time gap larger than the maximum separation time, or a distance gap larger than the maximum separation distance is found. Further, if this splitting yields a set of position reports whose size is less than the given minimum number, the set of position reports is discarded. The result is about 45,000 separate trajectories for each day of ASDI data (in 2015).

Our data is now separated into a set of flight trajectories, with each flight trajectory containing a set of position reports. Because of the noisiness of the data, we perform another filtering step, using a heuristic to identify suspicious position reports in each flight trajectory. A position report is considered suspicious if, (1) it occurs abnormally soon after a previous position report, (2) its reported position is far away from the previously reported position, (3) its reported altitude is far above or below the previously reported altitude, (4) its reported altitude is below some threshold value (for example 0). All suspicious position reports are removed from the flight trajectories.

In a final filtering step, we also discard any flight which now contains fewer than the minimum number of position reports for a flight, any flight whose call sign indicates that it is a general aviation flight, and any flight whose origin airport or intended destination airport is invalid. Each position report includes the origin airport and the intended destination airport. An airport is considered invalid if, (1) it is not reported, (2) it changes between the first and last position reports, or (3) the actual trajectory start or end is not within 50 km of the reported origin or intended destination. Removing flights with invalid airports allows us to have unambiguous ground-truth data. There are many trajectories in our data set, for example, which start or terminate abruptly due to technical difficulties, or at the edge of the tracked space. Including these trajectories, which do not terminate at the intended airports, yields degraded results. Similarly, removing general aviation traffic allows us to concentrate on commercial and cargo air traffic. The result of this pre-processing is a database of trajectories (about 11,000 for each day of 2015 data) which we can now use to assist in destination prediction.

The output of this step is a set of trajectories where each trajectory is comprised of $n+1$ time-stamped points $(x_0, t_0), (x_1, t_1), ..., (x_n, t_n)$, where x_i is the position of point i. In our example application of predicting aircraft destinations, the positions are latitude, longitude pairs, however the our method also works for other coordinate systems.

Creating Feature Vectors

We desire to a way to succinctly summarize a trajectory (or fraction of a trajectory). Combining a small set of normalized trajectory characteristics into an N-dimensional feature vector will enable us to find similar trajectories in and N-dimensional space, using traditional techniques.

Two issues must be overcome for this approach to work. First, an appropriate set of trajectory characteristics to use in feature vector creation must be determined. Second, since our goal is to predict the destinations of flights for which only a fraction of the trajectory is given, the feature vectors created must allow for matching partial trajectories.

A wide variety of characteristics have been used to summarize trajectories in trajectory analysis [35] applications such as finding similarly shaped trajectories, or identifying anomalous trajectories. Often, characteristics for these applications are chosen such that they are translation invariant. For the application of finding destinations, however, translation invariance is not desirable. Instead, trajectories which basically follow the same path, without translation, must be found. To accomplish this, we choose a set of N sample points equally spaced in time along the trajectory. The positions are determined by linear interpolation (along the great circle path), when they are not aligned with an existing position report. Each sample point contributes to the feature vector two values, the latitude and longitude components of the position. These sample points provide a succinct representation of the trajectory. The feature vector may also include additional values representative of the entire trajectory, such as total flight duration. As an example, setting N to 4, and using a single

additional value of duration, yields a 9-dimensional feature vector.

To be able to match partial trajectories we simply create feature vectors for progressively larger portions of each original trajectory. For example, feature vectors could be generated for the first 1/4, the first half, and the first 3/4 of an original trajectory, generating 3 feature vectors from one trajectory. Creating feature vectors in this manner, enables better destination prediction for trajectories of in-progress flights. Users may select a lower bound, an upper bound, and a number of feature vectors per input trajectory as parameters to control how feature vectors are generated. For even better results when large training data sets are utilized, we select random-length portions of the original trajectory.

Building an R-tree

Next, the generated feature vectors are stored in a data structure known as an R-tree [25]. An R-tree is a data structure which enables efficient searching for nearest neighbors in multi-dimensional data. Our implementation builds upon boost's implementation of an R-tree, adding specialized find methods. For this stage of the prediction process, we simply insert the feature vectors created for the input training data set of trajectories into an instance of the R-tree data structure.

Finding Similar Trajectories

Now, given a new trajectory for which we desire to predict a destination, we need only create a feature vector the trajectory (using the same method described above), and find its K nearest neighbors in the R-tree. For each similar trajectory i and j, a confidence value c_{ij} is generated to measures how well the trajectories match. The confidence value is essentially the inverse of the distance between the points in feature-space squared, and is computed using the following formula:

$$c_{ij} = \frac{1.0}{0.01 + {d_{ij}}^2} \tag{2}$$

where d_{ij} is the distance between the feature vectors i and j.

Performing Analysis

The similar trajectories and associated confidence values can now be analyzed to predict destinations. We describe two different methods of analysis depending on the desired output, though many others could be utilized. For the first method, we associate each trajectory in the data set with a destination airport code (i.e. LAX, for Los Angeles International Airport). The confidence values of those trajectories (from the set of similar trajectories found) with matching destination airport codes are summed. The result is a weighted list of potential destination airport codes. From this list the trajectory with the highest confidence

can be reported as the expected destination. If multiple guesses are allowed, we can proceed down the list until a threshold weight or number of guesses are reached.

For applications where a set of potential destination locations (such as airports) is not available, the actual expected destination position can be computed. This is accomplished by taking the weighted spherical linear interpolation (Slerp) [38] of the destination positions of the similar trajectories found, using their associated confidence values as weights. The result will be a point which is the weighted "geographic" mean of the candidate trajectory destinations.

5.4 Results

We tested our destination prediction method with the Tracktable Trajectory Analysis library [37]. Tracktable is an open source library which contains a core set of functionality for ingesting, processing, plotting, and analyzing trajectories. We have subsequently incorporated or prediction method into the library. This section describes the parameters used, and the results obtained using our method to predict destinations of real aircraft trajectories.

Parameters

Our data cleaning and destination prediction rely on several parameters. Table 2 lists these parameters, and the values used to obtain the results described below. The first 5 parameters are used to filter out bad points and bad trajectories. The next four parameters adjust how feature vectors are created for a trajectory, and how multiple feature vectors are generated for different fractions of an input trajectory. For this analysis we use 4 position samples per trajectory, and an additional value, representing the duration of the partial trajectory. Finally, the last parameter is used to determine the number of nearest neighbors to utilize while finding similar trajectories.

Matching Trajectories

First, we show an example of our method successfully finding several similar flights with a common destination. We select as a test trajectory the path taken on an inter-island flight in Hawaii from Kona to Honolulu. Using only features from the first half of this trajectory, and the duration of the first half of the flight, we employ our method to find matches and predict the destination given a training set of only 8,636 trajectories spread across the United States. In Figure 10 the trajectories of the test trajectory and 9 matching trajectories are plotted on a map of the Hawaiian Islands. As with other figures in this work, the trajectories are plotted with lines whose color slowly transitions from red, at the origin, to blue, at the destination. The matching trajectories all depart from the Kona International Airport (KOA) and arrive at the Honolulu International Airport (HNL), giving us a very

Trajectory Splitting and Cleaning	min trajectory length	20 samples
	max separation time between consecutive points	10 minutes
	min separation time between consecutive points	30 seconds
	max distance between consecutive points	60 nautical miles
	max altitude change between consecutive points	75,000 feet
	min altitude	0 feet
	max distance between airport and trajectory end	50 km
Feature Vector Creation	N (samples per trajectory)	4
	additional value in each feature vector	duration
	low	0.1, 0.2 for table data
	high	1.0, 0.8 for table data
	feature vectors per trajectory	10, 7 for table data
Prediction	K (num nearest neighbors)	10

Table 2. Parameter values for cleaning data and predicting destinations

high confidence that the intended destination of this flight is the Honolulu airport. While this is a straight-forward example, since there are few other likely destinations along this path in the vast ocean, it allows us to examine some aspects of the method. First, notice that our method finds trajectories which follow the same general path but can exhibit some variability. Also note that because our test trajectory is a partial trajectory, utilizing only feature points along the first half of its total path, there is generally more variation in the trajectories after the midway point. One flight, in particular juts out to the north east just after the midway point. This trajectory would not have matched as strongly had the test trajectory been slightly longer. Figure 11 similarly shows a successful prediction for a flight from Seattle Washington to Portland Oregon. Notice how perfectly the trajectories in the first half of the flight are aligned.

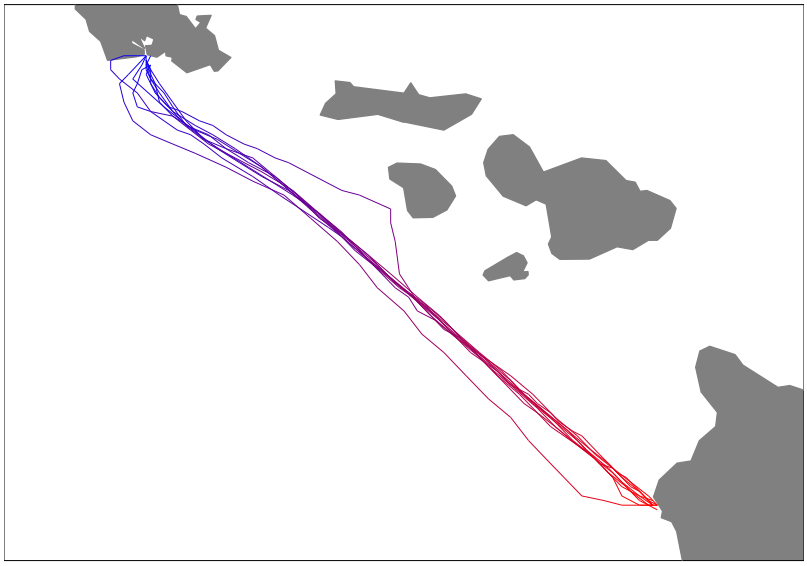


Figure 10. An example of 9 flights used to accurately predict the destination of a test trajectory, the first half of a flight from Kona to Honolulu Hawaii. (red=origin, blue=destination)

Mismatched Trajectories

As could be expected, matching a test trajectory is not always so easy. Figure 12 shows a region of the western United States, with Wyoming in the center, Utah on the left, and South Dakota in the upper right. Plotted are a test trajectory for Sky West flight 125F departing Rapid City Regional Airport (RAP) in South Dakota, and arriving at Salt Lake International Airport (SLC) in Utah, and the top 5 similar trajectories found using our method when using a training data set containing 21,569 trajectories of flights across the United States all on

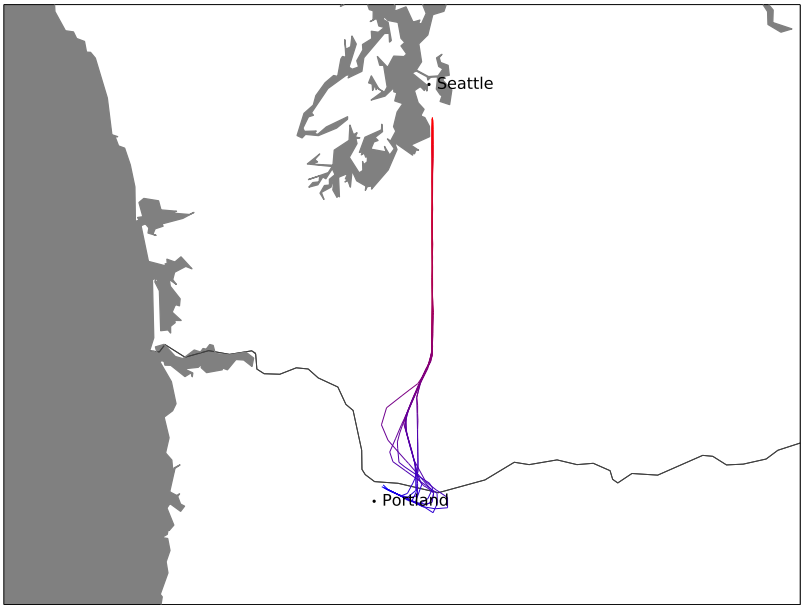


Figure 11. An example of 10 flights used to accurately predict the destination of a test trajectory, the first half of a flight from Seattle, Washington to Portland, Oregon. (red=origin, blue=destination)

the same day, July 10, 2013. As you can expect, the destination prediction for this case is a dismal failure, as none of the matching trajectories even share the same destination. Notice that there are even matching trajectories that fly the opposite direction of the test trajectory, from south to north. The failure of our method in this case is likely due to insufficient training data. This appears to be an uncommon route, perhaps flown only once a day, or less. If predictions are to be accurately made on uncommon flights, such as this, our method requires more training data, perhaps weeks or months of data.

Prediction Accuracy

We evaluate the accuracy of our method using one month of historical trajectory data from July 2015. Cleaning the large data set results in 323,605 trajectories. We employ a leave-one-out approach for this analysis. One trajectory is left out of the training data set, and then the destination for a random fraction of that trajectory is predicted. This is repeated for each input trajectory, such that each is "left out" in turn. By a random fraction of a trajectory we mean a partial trajectory starting at the origin and extending to a point, a random temporal fraction of the way towards the destination. Using only a fraction of the trajectory enables realistic testing of predicting destinations of unfinished trajectories. The random fraction for this analysis is selected anywhere within the same lower and upper

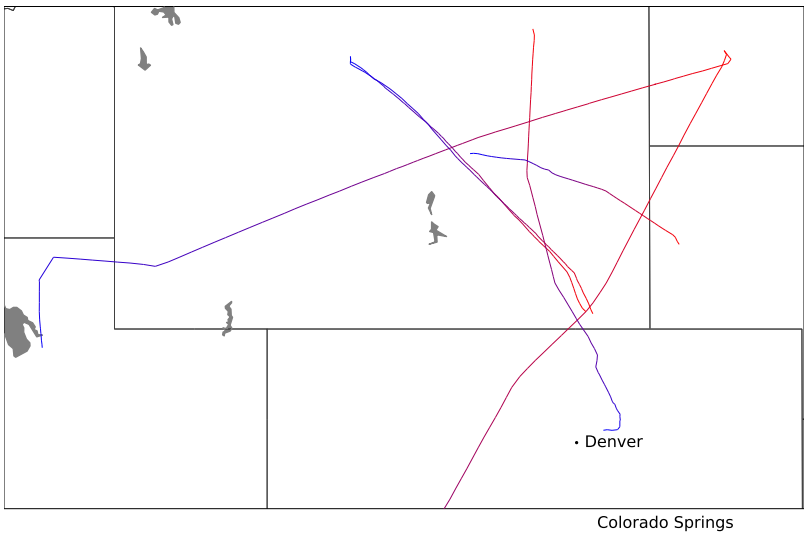


Figure 12. An example where our prediction method fails. A test trajectory from Rapid City to Salt Lake City is plotted, as well as the top 5 matching, which show little similarity due to a lack of training data. (red=origin, blue=destination)

bounds used for creating feature vectors (see Table 2 and note we use 0.2 and 0.8 for the lower and upper values here). We find the 10 nearest neighbors, and extract the destination airport codes from those matches. We then merge results for identical airport codes, to obtain the top potential matches. Table 3 shows the results of this analysis.

The results indicate that our method is able to obtain relatively good accuracy given a large training data set. While for about 14% of the flights the destination was not able to be predicted, over 64% of our top matches were correct. An additional 12% (76.7%) of the second matches were correct, and nearly 86% of the destination were able to be correctly predicted given up to 5 guesses.

A more detailed analysis is obtained by generating feature vectors for 10 random portions of each trajectory in the data set, where each random portion extends from the origin to 10 to 100% of the way to the destination (low=0.1, high=1.0). We again employing a leave-one-out analysis, and predict the destinations of 10 random portions of each trajectory also extending from the origin to 10 to 100% of the way towards the destination. Figure 13 shows the results of this analysis where each data point is the number of correct predictions for all the random samples of a given percentage of flight length, for various numbers of top matches.

The figure shows that half way through a flight the correct destination can be predicted

	Number	Accuracy
Not Matched	44,390	13.7%
Top Match	208,439	64.4%
Within Top 2 Matches	248,316	76.7%
Within Top 5 Matches	276,756	85.5%
Within Top 10 Matches	279,215	86.2%

Table 3. Accuracy given 1 months worth of historical aircraft trajectories, for a first random fraction for each left out trajectory.

as the top match 63% of the time, and in the top 5 matches 85% of the time. There is also little difference between using the top 5 and top 10 matches, suggesting that fewer than 10 matches could be computed, with minimal impact. The rise at the beginning of the plot is likely a boundary effect, since there are no feature vectors generated for the first 10% of the flights.

5.5 Summary of Prediction Work

We have described our novel method for predicting destinations of incomplete trajectories by finding similar trajectories in a a high-dimensional space populated with feature vectors derived from historical trajectories. Our method is efficient, in that it does not need to directly compare the new trajectory to all historical trajectories, and it is relatively accurate, given a large data set.

Several aspects of our method could be improved with further effort. Accuracy could likely be improved by utilizing the Hausdroff Distance Algorithm or other similarity algorithms to further refine predictions once a small set of candidates are identified. Also, our method is highly parallelizable, and we are working towards efficiently using multiple processors to speed-up the processing stages, and R-tree look-up.

Data scientists deal with large data sets of trajectories, where similar trajectories are often traveled from a set of common origin and destination points, they now have a new tool to efficiently predict probable destinations of unfinished trajectories. We hope that this new method will enable further progress in analyzing and predicting behaviors in various data sets.

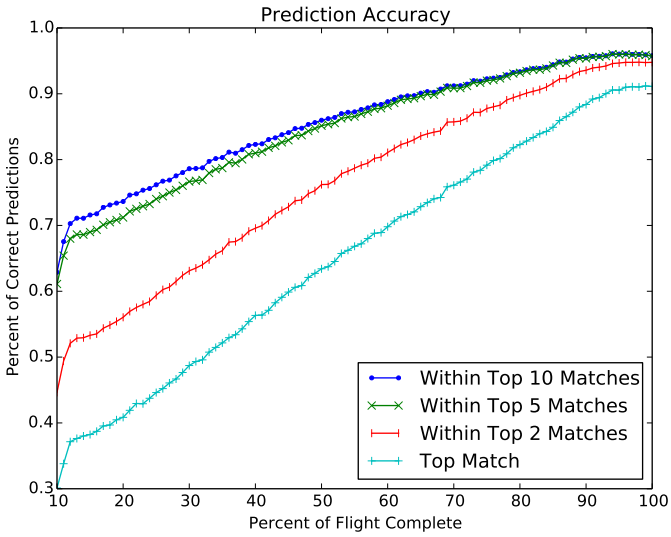


Figure 13. Accuracy obtained using various numbers of top matches, employing the leave-one-out method.

6 Machine Learning

Another use of having a feature-based description of the trajectories is the ability to provide numerical values to machine learning algorithms. We describe here a novel approach to doing classification via ensembles of decision trees.

6.1 Machine Learning Data

Trajectories

The raw data for our machine learning experiments was extracted from the raw ASDI data described in Section 1. Our goal was to work with data for a contiguous range of dates covering at least one full year, so that seasonal and other cyclical processes could be observed. Unfortunately, issues with the data feed from the vendor led to unavoidable gaps in the database, so that in the end our experiments were run using data covering the period from April 1st, 2013 through February 28th, 2014. Although this time range covered 334 days, there were 48 days with no data and 44 days with reduced or corrupted data[1].

The raw ASDI data that we were working with comprised one delimited text file per month, containing the set of observed aircraft positions over the United States for that month. Each row in the file contained one observation of a specific aircraft at a particular time, plus a sparse set of additional features. Typically, the data contained one observation per aircraft per minute. Key raw data features that were used in our analysis were as follows:

aircraft_id A unique aircraft identifier. Either the flight number for scheduled commercial flights, or an aircraft tail number for unscheduled commercial and private flights.

update_time Timestamp when the aircraft was observed.

longitude Aircraft longitude in degrees. Negative is West, positive is East.

latitude Aircraft latitude in degrees. Negative is South, positive is North.

speed Aircraft speed.

heading Aircraft heading.

altitude Aircraft altitude.

scheduled_arrival_time Time at which the aircraft was scheduled to arrive at its destination.

scheduled_departure_time Time at which the aircraft was scheduled to depart its origin.

[1] And this was not atypical; we looked, but there were no other year-long ranges in the ASDI data that were better behaved.

origin ICAO code for the departure airport.

destination ICAO code for the destination airport.

There were many additional features in the raw ASDI data that we did not use for our analysis.

Airports

We supplemented the raw ASDI data with the OpenFlights Airport Database [2] which contained, in addition to other fields, the ICAO code, latitude, and longitude for 6977 airports worldwide.

6.2 Machine Learning Trajectories

Since our goal was to apply supervised machine learning to aircraft trajectories, the first step in our analysis was to transform the raw data from a collection of individually-timestamped aircraft positions into a collection of *trajectories*, where one trajectory would encapsulate the entirety of a single flight.

Software

The software described in this subsection was written using the Python [3] programming language, and relied heavily on the numpy [41] and scipy [27] scientific computing libraries for Python.

Preprocessing

The first step in the trajectory-generation process was to split the raw ASDI data files - each of which contained a month of aircraft observations - into one day of observations per file. Processing 24 hours of data at-a-time reduced the memory footprint for our workflow and facilitated operating on the data in parallel.

Trajectory Grouping

To convert the raw ASDI data into trajectories for machine learning, we created the `trajectory2avatar` command-line tool, which functioned as-follows:

First, one-or-more delimited text ASDI files was loaded and parsed, and the tool iterated over the loaded observations. In this case each observation represented the position of an aircraft at a specific time. The observations were grouped together in a hashed dictionary, using the aircraft id (the scheduled flight number or aircraft tail number) as the dictionary key. This provided us with a collection of per-aircraft observations over time; however, the grouped observations were not the same as *trajectories*, since a given aircraft might make many flights within a single data file.

To split the grouped observations into individual trajectories, we used two parameters: a maximum time gap between observations, and a minimum number of observations in a trajectory. Using these parameters, the process to group per-aircraft observations into trajectories was as follows:

1. Identify every pair in the set of per-aircraft points where the gap between timestamps exceeded the maximum time gap parameter, which was 10 minutes in our experiments. Because the raw data typically contained one observation per minute for each aircraft, a ten minute gap between observations was a strong signal that one flight had ended and another had begun. Each of these gaps represented a *possible* boundary between flights (trajectories).

2. For every group of points between boundaries, if the number of points exceeded the minimum (20 points for our experiments), we created a new trajectory with the given points. Otherwise, we discarded the points. Again, given one observation per minute, we found that very few flights were less than 20 minutes in duration, so this approach filtered out potentially noisy data.

Having grouped together the raw data observations into individual trajectories, our real feature-extraction could begin.

Trajectory Feature Extraction

The following process was applied iteratively to each of the trajectories identified in the preceding subsection.

First, we filtered trajectories whose first timestamp fell outside a range specified on the command-line. Since our raw data could only be accessed in 24 hour chunks, this allowed us to perform finer-grained analyses, extracting trajectories from individual hours during the day, and extracting trajectories for every flight that originated during a given day, even if they continued into the following day.

An important goal for our research was to determine not only whether we could make meaningful predictions on entire trajectories "post-mortem", but whether we could make similarly meaningful predictions on subsets of trajectories, e.g. could we make predictions

on a trajectory while the flight was still in progress? To facilitate this analysis, we provided command-line options to specify the range of trajectory points to be analyzed. With these options we could analyze just a fixed percentage of points at the beginning of each trajectory (e.g. analyze just the first 10% of each trajectory); a fixed percentage at the end (e.g. analyze just the last 20% percent of each trajectory); a fixed percentage in the middle (e.g. analyze just the middle 50% of each trajectory); or a fixed percentage with a random offset (e.g. analyze a random 15% of each trajectory). Thus the first step in analyzing each individual trajectory was to identify a subset of the trajectory observations to be analyzed. Some of the features in the discussion to follow were based on this analyzed subset, while some features were extracted from the full set of trajectory points. Where appropriate we will note which set of points was used to extract each trajectory feature.

Because a single aircraft could make many flights in a single data file, the first feature created for each trajectory was a `trajectory_id`, which was generated by concatenating the aircraft_id with the timestamp of the first and last observations for the full trajectory. This provided a unique identifier that could be used to access individual trajectories, as we will see in Section 6.3.

Next, we extracted `day_of_year` and `time_of_day` features, based on the timestamp from the first observation for the full trajectory. Converting a monotonically-increasing timestamp into cyclical features in this way provided features that our machine learning algorithms could use to tease-out correlations based on daily or seasonal effects.

The originating airport ICAO code was extracted from the first observation of the full trajectory as an `origin` feature, along with `origin_latitude` and `origin_longitude` features, identified by looking-up the origin ICAO code in our OpenFlights Airport Database.

Similarly, we extracted `scheduled_destination`, `scheduled_destination_latitude`, and `scheduled_destination_longitude` features, using the destination airport ICAO code from the *first* observation in the full trajectory.

Because the destination airport code could change over the duration of a flight (i.e. if the flight was diverted), we also extracted an `actual_destination` feature, along with the actual destination latitude and longitude, using the destination airport code from the *last* observation in the full trajectory.

Using the originally scheduled departure and arrival times from the first observation in the full trajectory, we created a `scheduled_duration` feature which represented the expected flight duration at the time of departure.

With altitude information from the *analyzed subset* portion of the trajectory observations, we extracted two features:

- `altitude_variance_80` was computed as the variance of the middle 80% of the analyzed altitudes. This feature was intended to capture the prevalence of altitude changes during the course of a flight (excluding the initial climb and final descent), on the as-

sumption that wide changes in altitude might be correlated with poor weather or disruptive changes in traffic.

- `altitude_mode` was computed as the mode (most common value) of the analyzed altitudes, as a way to capture the most representative altitude for the flight as a whole.

Next, we computed the distance from the origin airport to the *first* observation in the full trajectory, and the distance from the *last* observation in the full trajectory to the actual destination airport. If either distance exceeded a threshold (80 kilometers in our experiments), we marked the trajectory as `clipped` and removed it from further analysis. This allowed us to filter-out trajectories for which only partial data was available in the raw data. Typically, this occurred for international flights to and from the United States, because the ASDI data only contained information describing that portion of a flight that occurred within the airspace of the United States and its territories.

Next, we extracted a set of features based entirely on the latitudes and longitudes in the analyzed subset of a trajectory, as outlined in Section 3:

The `travel_distance` feature was the sum of the distances between each observed position, and thus an estimate of the total distance travelled for the (subset of points in the) trajectory.

The `start_end_distance` feature was the distance between the first and last points in the analyzed subset. Note that this distance might differ significantly from `travel_distance` since it represented the distance "as the crow flies".

`scheduled_start_end_distance` was the distance between the origin airport and the originally-scheduled destination airport.

`actual_start_end_distance` was the distance between the origin airport and the actual airport where the flight ended. This value could differ greatly from `scheduled_start_end_distance` if a flight was diverted, or took a roundabout path to its destination.

We computed a set of "signed turns", which were the change in azimuth from one observation to the next in the analyzed subset of the trajectory. The signed turn values were summed to produce a single `winding` feature for the trajectory, while the absolute values of the signed turns were summed to produce a `turning` feature. In addition, the `loiter_ratio` feature was computed as the ratio between `travel_distance`, and the total distance around a convex hull containing the analyzed subset of the trajectory. Collectively, these features also acted as indicators of a "complex" or roundabout path from origin to destination.

We then used the latitudes and longitudes to extract a set of distance geometry features. Note that when computing these features, we used linear interpolation between trajectory position vectors for simplicity, and we interpolated over the trajectory coordinates, not time. We computed descriptors for $M = [1, 2, 3, 4]$, which we named `ld11`, `ld12`, `ld22`, `ld13`, `ld23`, `ld33`, `ld14`, `ld24`, `ld34`, and `ld44`. In this case the feature name `ldNM` represented the Nth

segment of the distance geometry with M segments. We also created a second set of distance geometry features, normalizing each feature by the `travel_distance` to create `ldn11`, `ldn12`, `ldn22`, `ldn13`, `ldn23`, `ldn33`, `ldn14`, `ldn24`, `ldn34`, and `ldn44`.

In all cases, we computed great-circle distances between latitude / longitude coordinates assuming a spherical model of the Earth with a radius of 6371 kilometers, which was a compromise between the equatorial radius and the polar radius.

Finally, we generated a `random` feature by drawing from a uniform distribution between zero and one. The purpose of this feature will be described in Section 6.4.

To provide ground-truth labels for our supervised machine learning task, we implemented two algorithms to label whether a flight had been diverted from its original destination or not. "Type 1 diversion" was defined as a flight where there was more than one destination over the course of the flight. "Type 2" diversion was defined as a flight where the `actual_destination` differed from the `origin_destination`. The difference between the two types of diversion was a subtle one - if, over the course of a flight, an aircraft was diverted to a new destination, then diverted back to the original destination, the Type 1 definition would consider this a diverted flight, while the Type 2 definition would not.

Trajectory Generation

With the `trajectory2avatar` tool written we could begin converting our raw ASDI data to inputs suitable for use with our supervised machine learning tools. A driver script was used to run multiple instances of `trajectory2avatar` in parallel, to take advantage of available compute resources. Over the course of the project, we ended-up generating the following collections of trajectory datasets:

trajectories-0-100 For our initial experiments, we generated trajectory data using 100% of the available raw data.

trajectories-0-90, -0-75, -0-50, -0-25, -0-10, -0-5 Once we had baseline performance using 100% of the available data, we ran experiments using the first 90%, 75%, 50%, 25%, 10%, and 5% of each trajectory.

trajectories-25-75, -37.5-62.5, -45-55, -47.5-52.5 Based on the surprising robustness of our predictions on partial trajectories, we generated datasets containing the middle 50%, 25%, 10%, and 5% of each trajectory.

trajectories-random-50, -random-25, -random-10, -random-5, -random-1 Finally, we created datasets was based on a random 50%, 25%, 10%, 5%, and 1% of each trajectory.

Each of these datasets contained a separate trajectory data file for every date in the range April 1st, 2013 through February 28th, 2014. Each file contained observations with

the features extracted by `trajectory2avatar` for each trajectory that originated on that date. To simplify downstream processing, we included empty data files for dates even if there was no raw data available.

Further, we generated a separate trajectory data file for every hour of every day for the same range of dates, including empty hours. As before, each file contained the trajectories that originated during that hour.

Dividing the data in this fashion gave us good granularity for parallel processing, kept the memory requirements manageable, and gave us great flexibility for our downstream processing, since individual trajectory data files could be concatenated together when needed to analyze arbitrary ranges of time.

The storage required by the 1.3 billion raw ASDI data points was 303 Gbytes. For each of our trajectory datasets, those points cooked down to 9.5 million trajectories (in roughly three hours of heavily parallel processing), and the resulting feature files consumed roughly 4.8 Gbytes per dataset.

6.3 Panther Font

An an exercise to demonstrate, in a human-consumable way, the utility of the distance geometry features described in Section 4.3 for performing search tasks, we conceived the idea of using those features to extract a set of alphabet-like trajectories from the raw ASDI data to produce the "Panther Font" - a font where each letter was represented by an actual aircraft trajectory from the ASDI data. Although time constraints eventually limited us to extracting just the letters in "PANTHER", the results of the exercise still proved useful, dramatically demonstrating the value of the distance geometry features as well as the surprising variety of trajectories in our data.

To begin, we hand-encoded a set of crude block letters as trajectories using latitude-longitude coordinates (Figure 14).

Figure 14. Block letter probe trajectories generated by hand.

We then generated the normalized distance geometry features for each letter using $M = [1, 2, 3, 4]$, which corresponded with the `ldn11`, `ldn12`, `ldn22`, `ldn13`, `ldn23`, `ldn33`, `ldn14`, `ldn24`, `ldn34`, and `ldn44` features described in Section 6.2.

We then scanned the trajectory data file for one day's data, comparing each letter to each

of the trajectories using a Euclidean distance metric and the distance geometry features. For each letter, we recorded the trajectory identifiers of the closest 40 matches. Using those identifiers, we extracted the raw latitude / longitude data for each matching trajectory, and generated a KML file containing all 40 matches, which was loaded into Google Earth [1]. Using visual inspection, we chose one "best" match from the 40, rotated and zoomed the view in Google Earth to center the match, and captured an image of the matching trajectory. After repeating this process for each letter, the images were assembled into a surprisingly legible PANTHER logo (Figure 15).

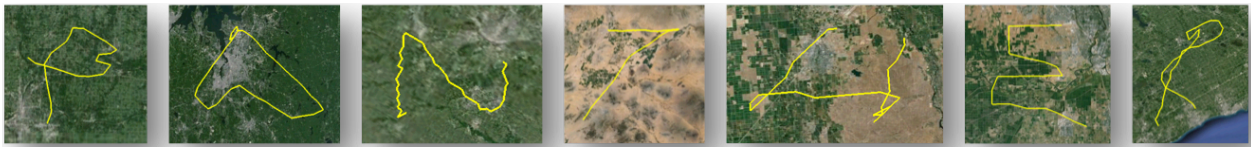


Figure 15. Matching trajectories extracted from one day's flights over the continental United States

We then repeated the process using our 334 day "year" of flight data, producing markedly improved matches (Figure 16). Note that the position, orientation, and scale of the matched trajectories could vary widely, since the choice of normalized distance geometry features ensured that only the *shape* of the trajectories was used for matching. Of course, we could have chosen to take additional features into account for matching, but this exercise, was an engaging demonstration of how the distance geometry features can be used to compress a complex trajectory into a remarkably compact feature vector for effective search and retrieval.

Figure 16. Matching trajectories extracted from one "year" of flights over the continental United States

6.4 Machine Learning Trajectory Analysis

Introduction

We describe our attempts to predict whether a flight will be "diverted" when working from the ASDI data described in Section 6.1. Diversion serves as an interesting prediction task both for its innate interest (as it is a good bet that everyone writing or reading this report

has been affected by a diverted flight) and as a sensible surrogate for more sensitive "predict change of state" problems. Further, it seems *a priori* likely that accurate diversion prediction might depend on an interesting mix of trajectory features *and* meta-data such as point of departure, time of year, and so on.

After establishing some idea of what baseline performance to expect using all available features and the full length of the trajectory, we will then investigate more realistic and increasingly constrained scenarios where we lack knowledge of the full feature set, the full extent of the trajectory, or both.

Methodology

We will use a supervised machine learning (SML) method for diversion prediction. Specifying an SML approach requires specifying the

- base elements of investigation,
- the features used to describe them,
- the labels we wish to predict,
- and the specific SML algorithm.

Since we primarily intend to assess performance under various circumstances, we must also specify our performance metric and how we apply it.

Trajectories as the Base Elements Unsurprisingly, we will use trajectories (or truncated portions of trajectories) as our base elements. The conversion of raw ASDI data into trajectories was described in Section 6.2. As outlined earlier, the specific data we investigated in these analyses covered dates from April 1, 2013 to February 28, 2014, with numerous empty dates. This will be reflected by drop-outs in the plots to follow.

Trajectory and Metadata Properties as Features The specific features extracted from the ASDI data, and the various derived statistical and trajectory features, were described in detail in Section 6.1 and Section 6.2. A summary is in Table 4.

The one unexpected oddity here might be the inclusion of the "random" feature, one uncorrelated with the diversion label. Such features turn out to be useful in the feature analysis to follow. Part of what we consider is the relative importance of each feature for making predictions, but it can be hard to tell when that ordering is dominated by noise rather than signal. Including a random feature is useful in that any feature assessed to be less important than random is clearly, actually irrelevant.

day_of_year	
time_of_day	
origin_latitude	
origin_longitude	
scheduled_destination_latitude	
scheduled_destination_longitude	
scheduled_duration	delta between original scheduled departure / arrival times.
altitude_variance_80	variance of altitude for middle 80% of trajectory.
altitude_mode	mode of altitude for entire trajectory.
travel_distance	actual path length.
scheduled_start_end_distance	distance from origin airport to scheduled destination airport.
turning	sum of the absolute values of the trajectory's turns.
winding	sum of the signed values of the trajectory's turns.
loiter_ratio	travel_distance divided by convex hull circumference.
ldNM	segment N of M from an M-ary partition of the trajectory.
ldnNM	ldNM divided by the actual path length.
commercial	0 for private, 1 for commercial.
random	a random, irrelevant feature.

Table 4. A summary of the trajectory features

Further, the SML method we will use is not affected by the inclusion of random features[2], so there's no harm in including this calibration feature.

"Diversion" as the Prediction Task As described earlier in Section 6.2, we identified two subtly-different ways to label whether a trajectory was diverted or not. For these experiments, we ended-up using the "Type 2" definition of diversion, where we labelled each trajectory as "diverted" if **actual_destination** differed from **origin_destination**.

That seems straightforward enough, but there are various small nuances to be aware of. For one, these labels are not perfectly accurate. The ASDI **destination** field is updated by hand, not by some automatic process, and sometimes that update did not occur. There are cases where the flight was clearly diverted (as indicated by the lat/long of the destination), but the **destination** metadata field was not updated. This is part of what will drive our selection of ensembles of decision trees (EDT) as our machine learning method, as EDT are robust to "groundtruth" that lies to you.

As another nuance, if a flight veers off, but returns to destination airport (that is, if over the course of the trajectory its raw points have more than one **destination** value, but the **destination** values at start and stop are the same), we call that undiverted.

[2]That is, accuracy is not diminished, though it is the case that computational load is slightly increased.

Finally, it is worth noting that on average, over the course of the year, the percentage of diverted flights is only 7.2%. (On a day to day basis, the percentage of diverted flights ranged from 4.3% to 38.6%.) This is a small enough portion of the total to be considered a "skew" machine learning problem, and so require special handling, as will be discussed in Section 6.5.

Ensembles Of Decision Trees as the Machine Learning Method Ensembles [30], and in particular ensembles of decision trees (EDT) [12], are a robust, reliably near-optimal method for practical, robust, accurate supervised machine learning[7]. They also effectively lower design stress in building a machine learning model, as they have the attractive property that they are unaffected by junk features and are invariant to monotonic transformation of the feature axes. Practically, this means there is no need to attempt to normalize features, to determine what their respective weight should be relative to each other: the decision tree ensemble can work this out on its own.

As a result, EDTs are very popular and widely used. We accordingly focused on EDTs as the primary supervised machine learning method investigated in diversion prediction. The specific EDT implementation we used is "Avatar", a set of tools developed in earlier Sandia projects. The Avatar Tools embody all the algorithms and practices that have been gleaned from a decade's research into how to best make supervised machine learning work with huge, messy, noisy, unbalanced data.

Features that distinguish Avatar Tools from other "ensembles for decision trees" codes include:

- Native implementation of out of bag (OOB) validation, in addition to standard methods for cross-validation such as 10-fold and 5x2 validation.
- The use of OOB validation to automatically determine optimal ensemble size.
- Implementation of a wide range of skew-correction mechanisms, SMOTE[14] in particular.
- Implementation of a wide range of skew-sensitive performance metrics.
- A post-processing analysis tool, "tree_stats", which provides a robust assessment of the relative importance of the features[13] used to build the EDT model. These assessments play a role in feature investigations in Section 6.5.

Evaluation Method Note that since the data is skew, that is, the average percentage of diverted flights is 7.2%, it could be deceptive to use overall accuracy as our metric of performance. After all, if we generated a trivial machine learning solution that labeled everything as "undiverted", it would be on average 92.8% accurate.

Accordingly, we will instead report "class averaged accuracy" (CAA), which is the average of the percentage of the diverted flights correctly labeled and the percentage of the undiverted flights correctly labeled. CAA has the attractive property that trivial classifiers that labeled everything "diverted", or everything "undiverted", or assigned labels uniformly randomly will all have an accuracy of 50%.

Our assessment scheme will be to evaluate each day individually, independent from the rest, and to assess each day with 10-fold cross validation. That is, the trajectories for a given day are partitioned randomly into ten disjoint partitions. Then one partition is set aside while the other nine are combined into a single training set and used to build an EDT model. The set-aside partition is used as a test set for that model, and a CAA value is determined. That process is repeated nine more times, using each partition as a test set, resulting in ten CAA values which are averaged to generate an overall CAA value for that date.

6.5 Diversion Prediction

BaselineResults

With the preliminaries of the prior subsection in place, see Figure 17 for a sense of the baseline diversion prediction accuracy. This depicts the daily CAA over the course of the year. The tic marks on the x-axis span a week each, and indicate Mondays. The y-axis indicates CAA; note that it only spans the range $[0.5, 1.0]$, as CAA less than 0.5 would indicate worse than random performance. Also note that the plot key (here and in the following plots) indicates the average CAA over all the values in the plot.

There are a number of notable properties of this plot. One is the fact that there are so many plot points missing; as explained in Section 6.4, this is due to the fact that the underlying ASDI data was missing or corrupted on those days, and so analysis was not possible.

Feature Importance Varies Over Time

Note also that the CAA values have a wide swinging range around their mean, and there even seems to be a certain cyclical structure. That cyclical structure seems to be roughly weekly in period, which motivated the choice of week-spanning tic marks on the x-axis.

For another view of the periodicity, consider Figure 18, which contains two views of a river chart depicting how much each feature mattered[3] when predicting diversion on that particular day. In this figure, each row captures the spread of feature importance for a single date, and each column depicts a single feature. Each feature has a unique color (and families of related features have similar colors), and the width of that feature's ribbon indicates the

[3]The feature importance measure is the PATH metric[13].

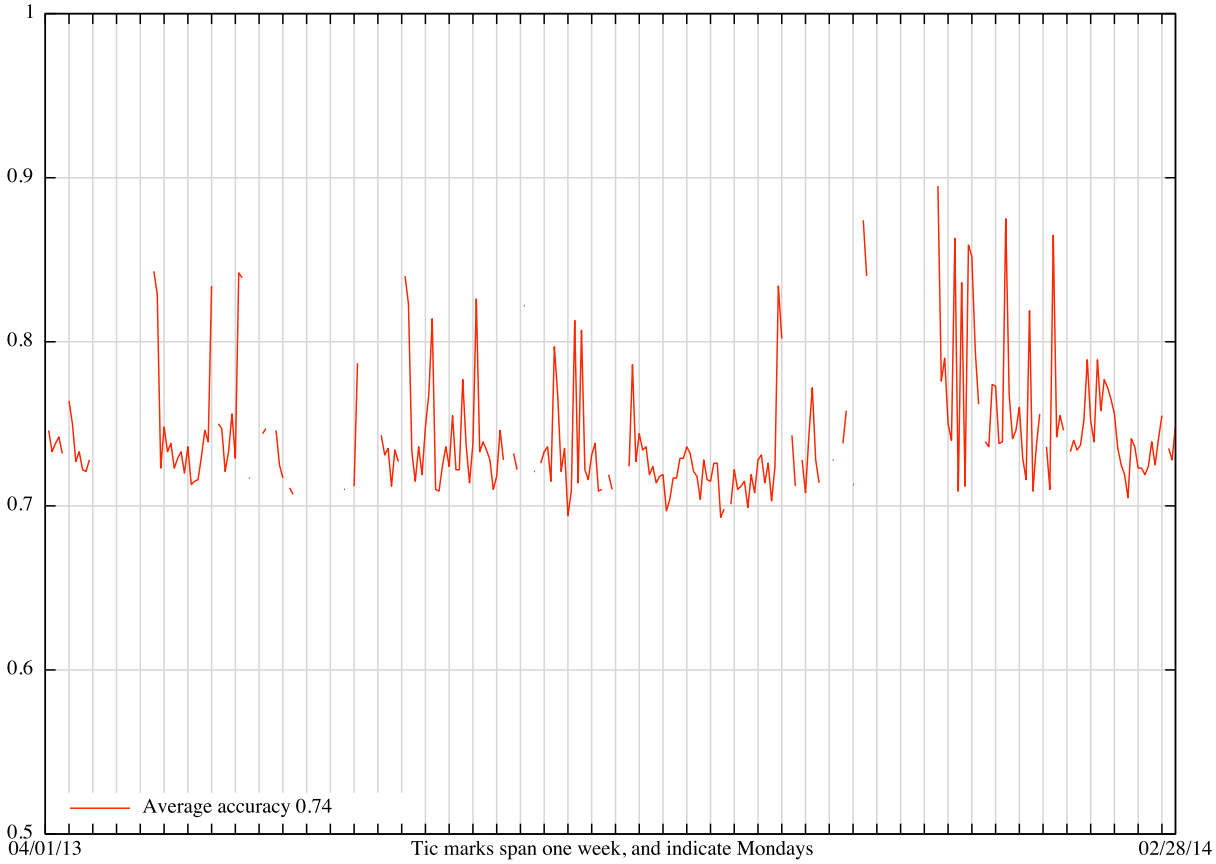


Figure 17. Class-averaged accuracy derived from day-by-day 10-fold cross-validation

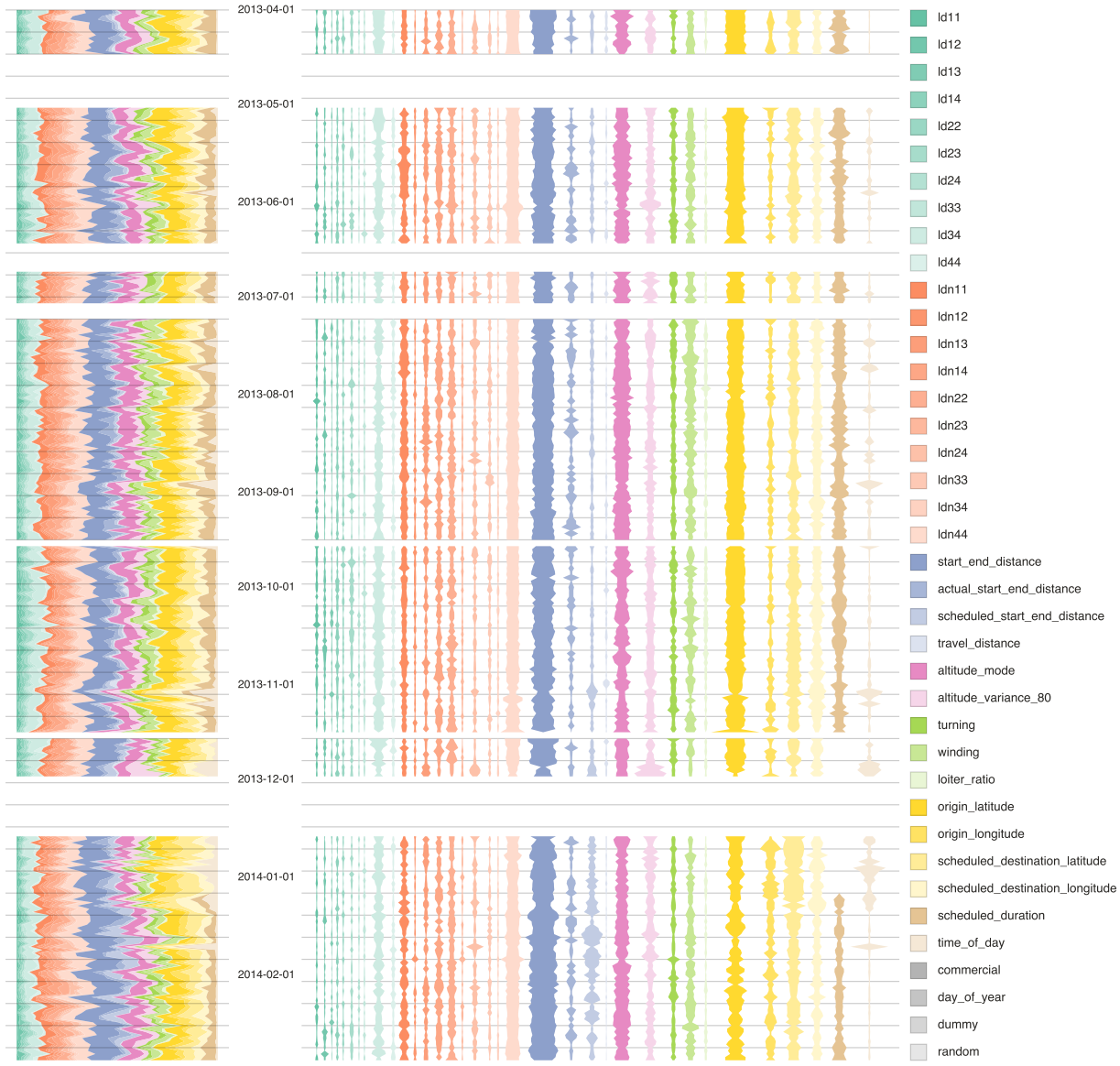


Figure 18. Which Trajectory Descriptors Mattered, and When?

relative importance of that feature on that date.

The left and right side of the figure depict the same information, the only difference is whether the importance ribbons were placed directly adjacent to each other. When they are, as on the left side of the figure, it is easily to see how the *relative* importance of the features, and of families of features, varies over time. When the feature ribbons are not adjacent, as on the right side, it is easier to see how the importance of an individual feature waxes and wanes over time.

A few points of observation, intended partly as examples to help indicate how to interpret this figure:

- In the right figure, note the consistent thickness of the **start_end_distance** (dark blue, as indicated in the key) ribbon. This means **start_end_distance** is reliably useful for predicting diversion, which makes some sense, as the longer the flight, the more time there is for a possible diversion, and the more uncertainty there is, at the beginning, about the end conditions.

- Similarly, **origin_latitude** is consistently important, which again makes sense, as the more northern flights are more likely to be diverted.

- A third feature of consistent importance is **altitude_mode**. This could be because constantly changing attitude might be a marker for bad weather, but it might also simply be the fact that longer commercial flights tend to fly higher.

- Note that none of the *individual* distance geometry features (the ldNM and ldnNM) features, are comparatively significant; which makes some sense, as each individual geometric feature captures only one abstract part of a trajectory's shape. Still, it is clear from the left side that in aggregate they account for a roughly a third of the explanatory power of the model.

Skew Corrected Baseline

A final note about Figure 17 is that the swings in CAA were quite large. We hypothesized that this could be due to day-to-day variations in the percentage of diverted flights, as the effect of such variations tends to be exacerbated by skew data. We also suspected that explicit correction for skew could improve overall CAA.

Accordingly, we altered our "ensemble of decision trees" method to ensure that each bag for each new tree is constructed to have 50% diverted cases by including *all* of the diverted cases, and then including an equal number of undiverted cases selected by uniform random selection with replacement. This is referred to as "roughly balanced bagging" in the literature [26], or as "majority bagging" in the Avatar Tools suite.

The result is in Figure 19. Here we see both the original performance plot and the perfor-

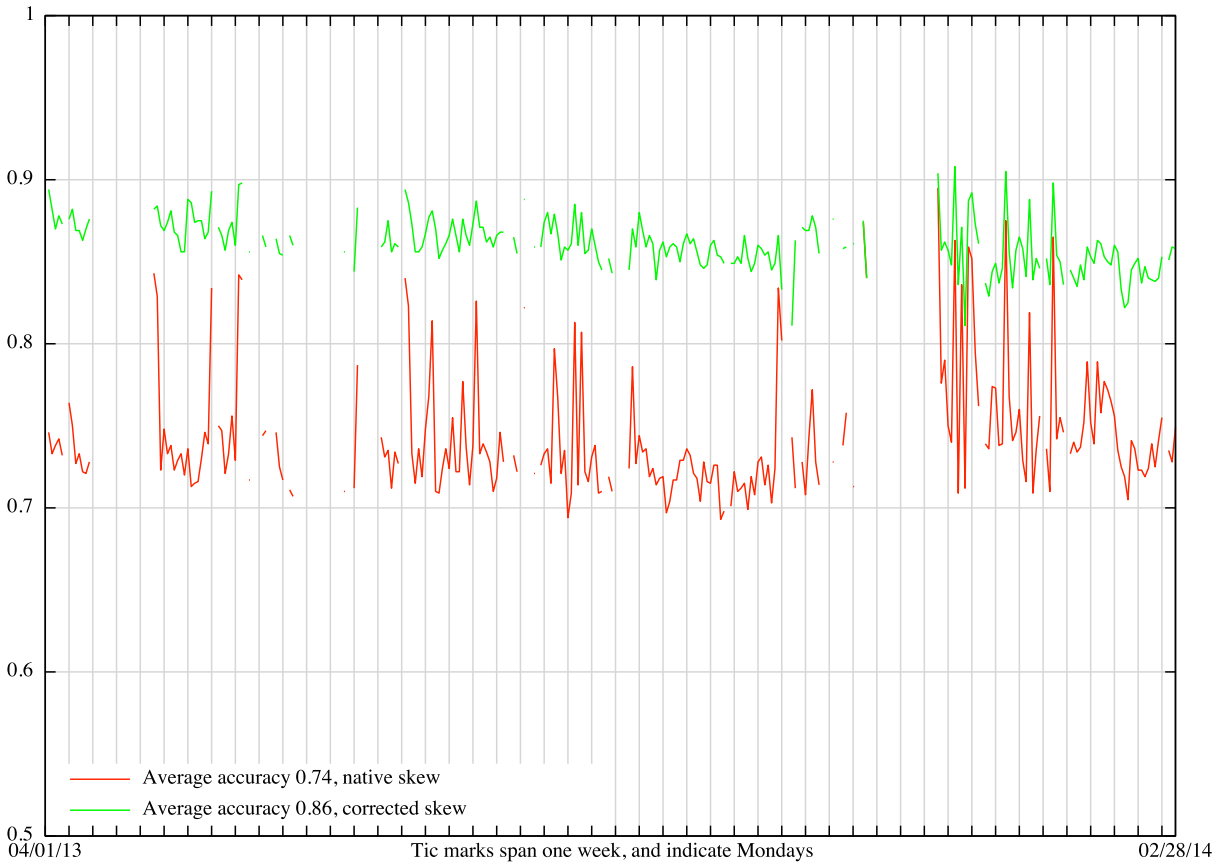


Figure 19. Improve Accuracy by Handling Skew

mance after skew correction. The average CAA has improved from 0.72 to 0.86 (primarily by stealing a little bit of accuracy from the prediction of "undiverted" in order to make the "diverted" prediction much more accurate), and the day to day variance has markedly declined.

As a result, all subsequent plots will show skew-corrected results only.

Prediction With Fewer Features

A class averaged accuracy of 0.86 was gratifyingly better than subject matter experts expected. But it was also pointed out that the wealth of information in an ASDI data stream is much richer than would be available in some operational contexts of interest to the US government. In order to set expectations for performance in those restricted contexts, we repeated the analysis *without* use of the following features:

- scheduled destination latitude and longitude,
- scheduled duration,
- altitude variance and mode, and
- scheduled and actual start-end distance.

Those features were selected by a subject matter expert as likely being unavailable in the contexts of interest.

The result of this restricted feature analysis is in Figure 20, which depicts performance with and without those features. As can be seen there, removing those features had a very minor effect, reducing average CAA only from 0.86 to 0.84. This is particularly surprising given that **start_end_distance** and **altitude_mode** were two of the three most important features. The likely explanation is that the third of the three important features, **origin_latitude**, picked up the slack, perhaps aided by retained features that were mildly correlated with the removed features.

It would have been straightforward to confirm all this by repeating the sort of analysis that lead to Figure 18, but only on the retained features. Instead, however, encouraged by our success on reduced features, we were encouraged to make the problem harder along a different dimension.

Prediction From Truncated Trajectories

That is, another unrealistic aspect of the initial analysis in Figure 19 is that it assumes that we have the entire trajectory to analyze. In practice, it's not very useful to anyone, and

Figure 20. Diversion Prediction with Fewer Features

especially not to the passengers, if we have to wait until the flight completes before we can predict whether it will diverted.

So we truncated each trajectory to just the first 5% of its length, recomputed features on the stumpy truncated trajectories, and repeated our accuracy analysis. The result is in Figure 21. Note that using only the first 5% of the trajectory degrades matters only

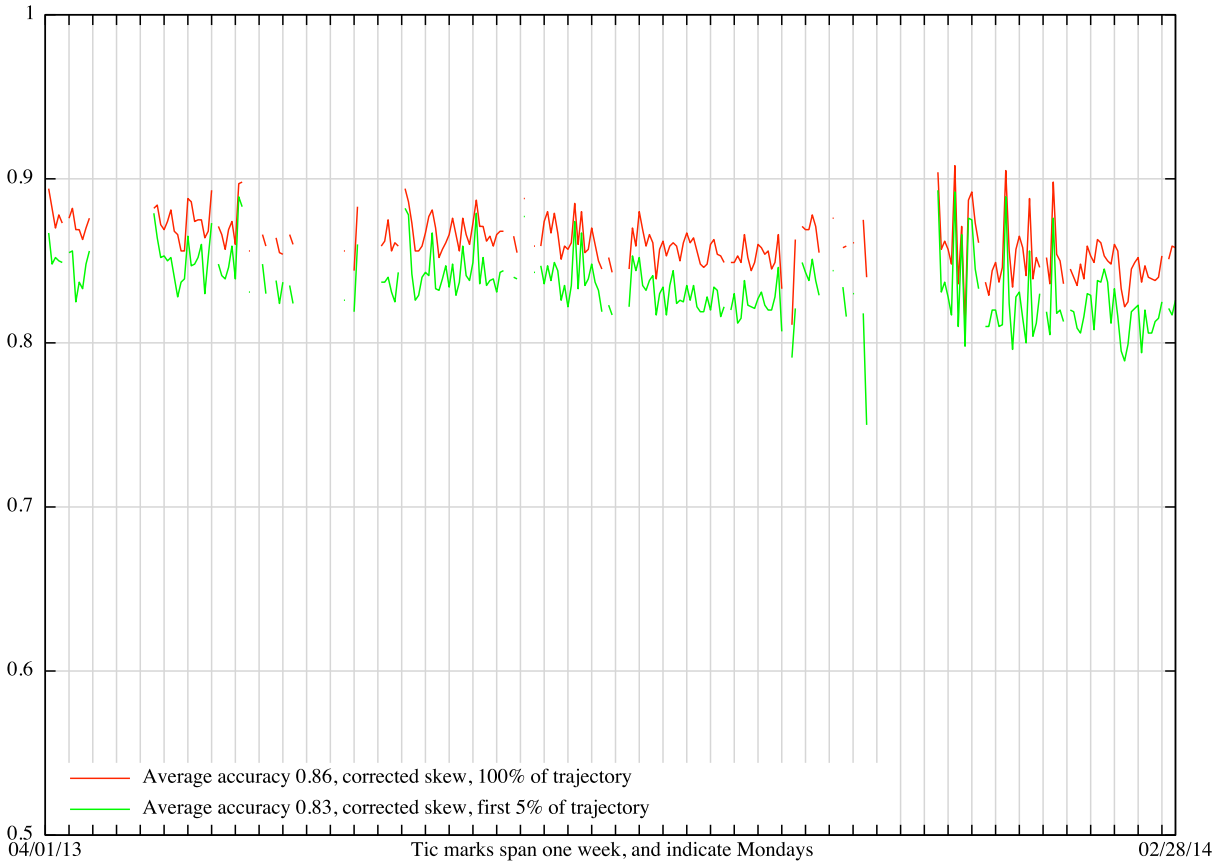


Figure 21. Prediction from Truncated Trajectories

slightly, dropping average CAA from 0.86 to 0.83. In fact, we actually started with 50% of the trajectory, dropping to 25% and 10% before settling on 5%, as the less severe levels showed near imperceptible effects.

In fact, given our success with analysis of the first 5% of the flight, the next question we were posed is what we could do with a *random* 5% of the flight. The idea is that this would reflect beginning to observe a flight only in mid-course.

The result is in Figure 22, where we can see that there is essentially no difference between using the first 5% of a trajectory and a random 5%. (Though see the discussion of **origin_latitude** in Section 6.5.)

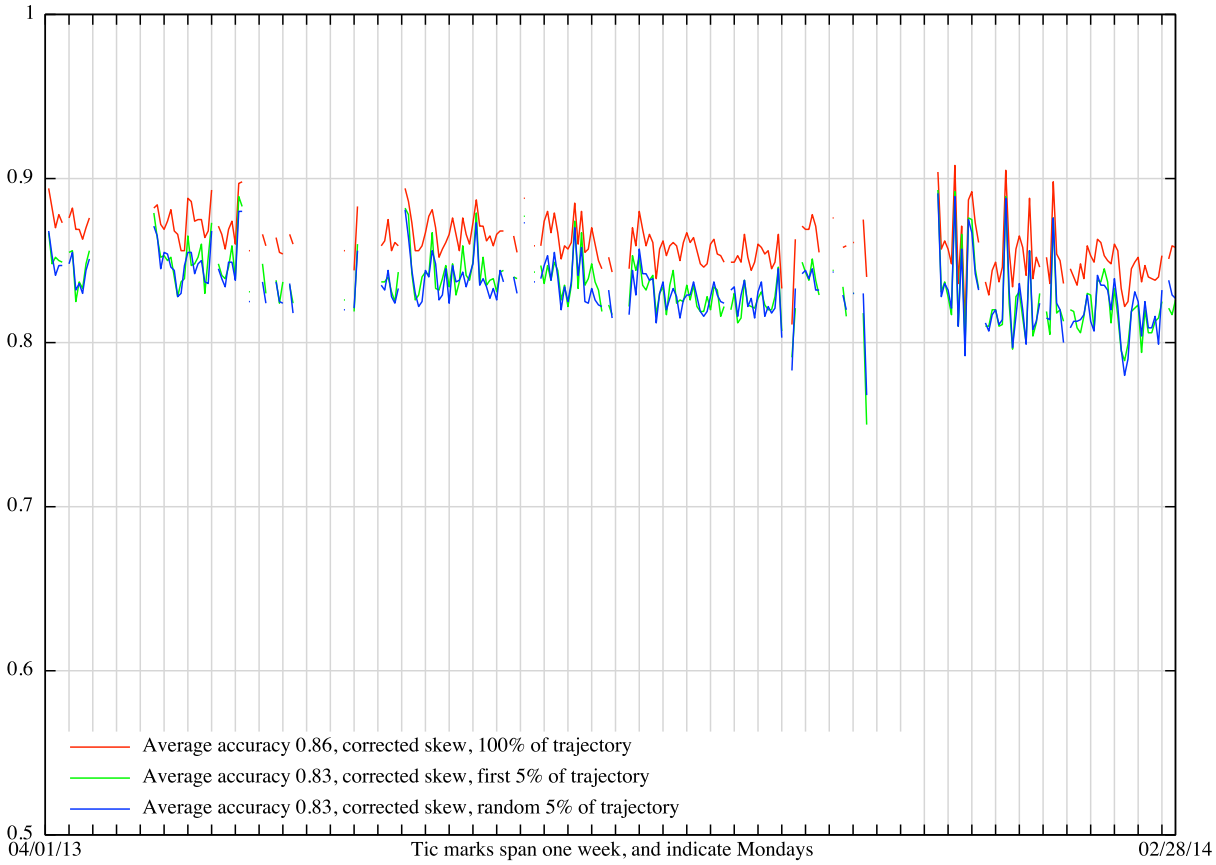


Figure 22. Prediction from Random Truncated Trajectories

Prediction from Fewer Features and Truncated Trajectories

Given that the machine learning prediction seems to well tolerate a reduced feature set or truncated trajectories, the next natural question is how it would respond if *both* effects were in play. The result is in Figure 23, which indicates that combining both restrictions only

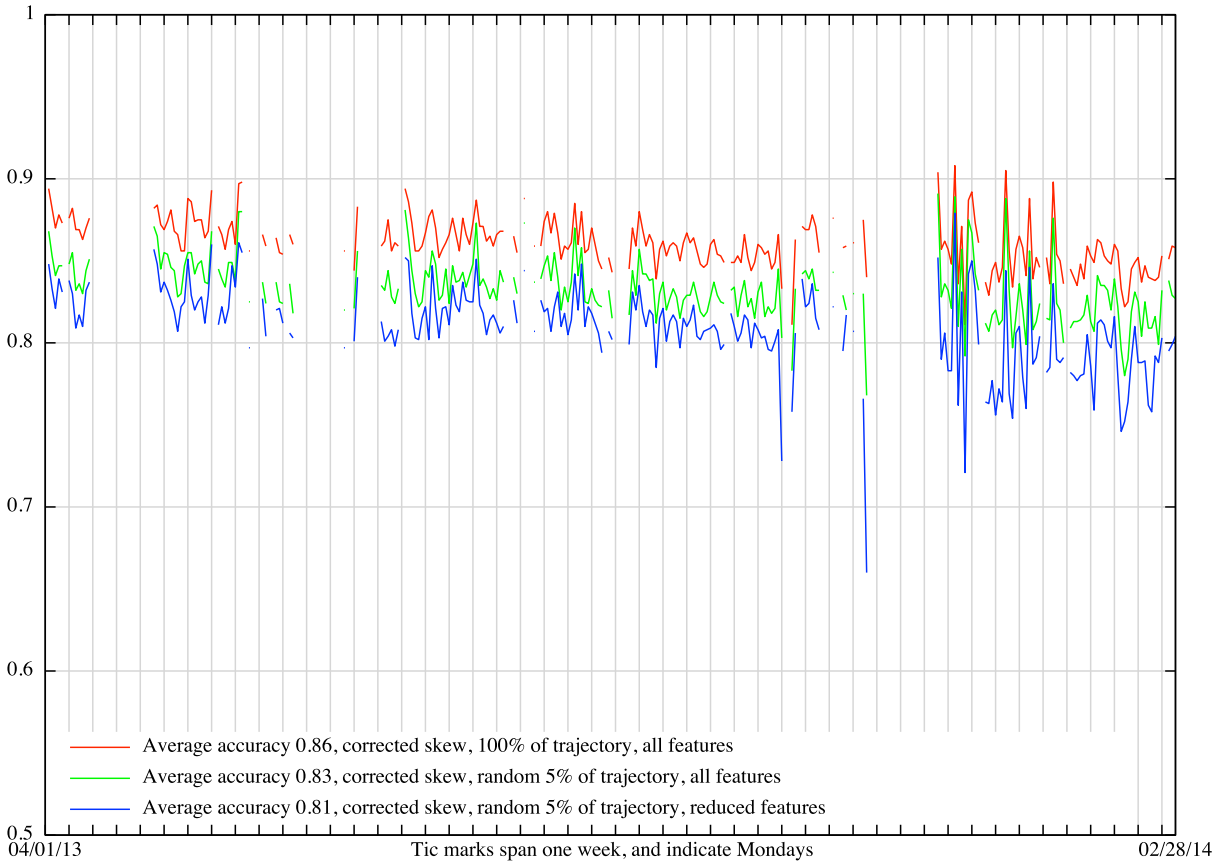


Figure 23. Prediction from Fewer Features and Random Truncated Trajectories

drops average CAA from 0.86 (with full features, full trajectories) to 0.81.

Prediction from Trajectory-Only Features and Random Truncated Trajectories

As a final test, and to get a sense of how these methods would fare in the most stringent of operational settings, we reduced the features to only what can be observed passively. That is, we do not use any of the information as provided in the ASDI data points, but base all features only on the observed latitude, longitude, and time stamp, and whatever derived features (such as the distance geometry features) might be computable from that series of triplets.

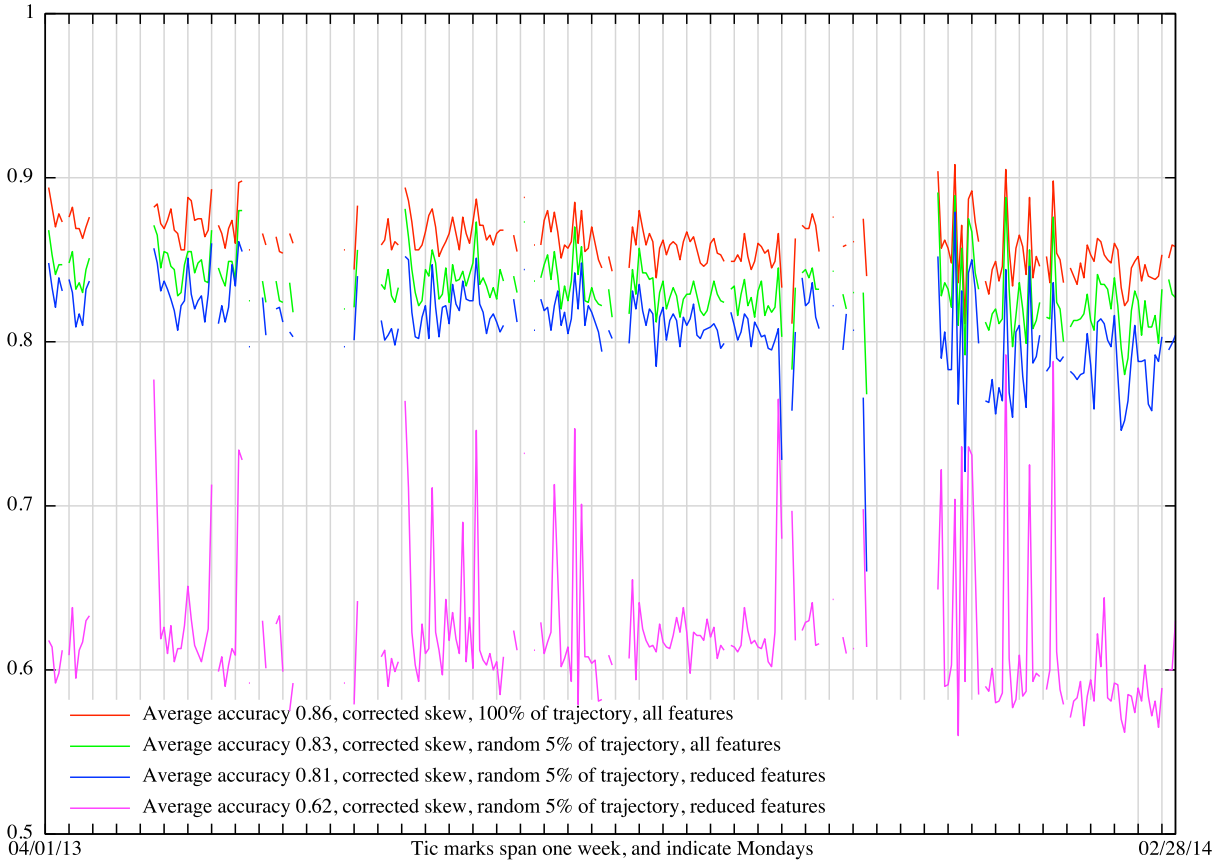


Figure 24. Prediction from Trajectory-Only Features and Random Truncated Trajectories

The result is in Figure 24. Finally we can see a substantial effect: if we only get to see a random 5% of the trajectory, and we only get to operate on non-ASDI features that we can observe (the 20 distance geometry features, plus turning, winding, and loiter ratio), our class averaged accuracy drops to 62%.

We did some feature analysis to determine the cause of the drop, and it turns out to be primarily due to the loss of **origin_latitude**. Which, in retrospect, is unsurprising. That feature was available even in the randomly truncated ASDI-trajectories because it is loaded into the data structure at the first moment of the flight and then emitted with every report at every point on the flight's trajectory. Further, Figure 18 clearly suggests that **origin_latitude** is one of the most relevant features, and since it is, unsurprisingly, uncorrelated with any of the observation-only features, its loss is clearly felt.

Summary and Next Steps

In summary, in assessing how well can we predict a diverted flight using supervised machine learning, we conclude:

- We can do fairly well, around 86% accuracy ...
- ... if we account for skew in the training data.
- Accuracy drops only 2% if we restrict ourselves to an operationally relevant subset of the flight metadata.
- Accuracy drops only 3% if we restrict ourselves to knowledge of just a random 5% of each flight.
- Accuracy drops only 5% if we restrict ourselves to an operationally relevant subset of the flight metadata *and* to knowledge of just the first 5% of each flight.
- Accuracy drops to 62% if we restrict ourselves to only distance geometry features and to knowledge of just the first 5% of each flight.

All of which suggests a handful of ideas for next steps:

Feature Analysis: It would be attractive, and likely generate some insight, to repeat the analysis that led to Figure 18 for each of the "reduced feature, reduced trajectory" scenarios considered above. So doing might suggest additional features that could be fairly added to improve results, or might simply lead to greater insight into what sorts of flight behavior are quantifiably associated with diversion.

Parameter Optimization: The 62% accuracy achieved in the most challenging case was achieved with the default machine learning control parameters, largely because performance prior to that final scenario was so good as to not motivate further tweaking. It

is quite possible that application of automatic wrapper methods would find a set of algorithm control parameters that would materially improve performance.

Cyclic Temporal Analysis: There is a clear, roughly weekly, cyclic basis to CAA accuracy. This seems to be because there is a more or less weekly cycle to the degree of skew in the number of diverted flights. It easy to set the machine learning parameters to optimally adjust to that skew, but it is not quite legitimate to do so, as the actual skew is not, of course, known *a priori* for any given day. The proportion of private and commercial flights did not turn out to be a useful proxy for the proportion of skew flights, but perhaps something else might.

General Temporal Analysis: The results above reflect using trajectories only from the current day to predict other trajectories in the same day. On one hand, that might seem to be the best case scenario, as the training data is very close, temportally, to the test data. On the other hand, each machine learning model is restricted to training from only a day's worth of data, when it could potentially build much richer models from years of data.

We started, but were unable to complete, some analyses intended to exploit additional data in model building. We looked at the effect of building models from the last seven days of data, or the last seven matching days of the week. We also worked on clustering every day of the year into a small set of contiguous chunks, based on the similarity of each day's features and the importance of those features, which we describe in Section 6.6.

6.6 Feature Importance Clustering

Having focused on creating per-day machine learning models in our earlier analyses, we hypothesized that it might be useful to partition the calendar year into a small number of contiguous date ranges so that we could compute supervised machine learning models for those dates.

For example, there might be significant differences between a "summer" model and a "winter" model, due to travel and/or weather patterns. However, instead of simply generating a set of models based on arbitrary dates, we opted instead to try to partition the calendar based on objective features. Because we had already observed some interesting changes in the relative magnitude of our per-day feature importance values over the course of the year, we speculated that they might act as useful features for an unsupervised clustering algorithm.

Our intent was to try to understand the waxing and waning of feature importance, while building a practical model deployment scheme. The idea being that a machine learning model would be built separately from each date range, so that in future years when analyzing a new trajectory, you'd simply pick the historical chunk matching the current date to determine which model to deploy.

Our first step was to load the feature importance values calculated in previous Avatar

runs for every day in our "year". This left us with an $M \times N$ matrix containing N feature importance values for each of 334 days. Since 48 of these days contained null values that could not be used with our clustering algorithm, we replaced each null with the mean value for its feature. We further normalized every feature (column) in the matrix to the range $[0, 1]$.

At this point, we could have applied a typical clustering algorithm to our feature matrix, but this would not have achieved our goal of generating contiguous clusters. For example, had we applied K-Means clustering with $K = 4$, the result would have been a set of assignments associating each individual day with whichever cluster was most representative of that day, independent of any nearby days. From our perspective this would have amounted to overfitting since our goal was to have a small set of robust models that would allow us to choose the best model for a given date based on its position within the year.

Thus, we chose to use a constrained hierarchical clustering algorithm provided by the scikit-learn [33] library. This algorithm allowed us to impose connectivity constraints on the analysis, constraining each observation in our data to be assigned to the same cluster as the previous and subsequent observations. When we applied the algorithm with this constraint to our data, it returned a set of K contiguous clusters that honored the constraints as-closely-as-possible, while still maximizing the intra-cluster proximity.

The alternating gray bars seen at the top of Figure 25 show the boundaries that were selected by the algorithm for $K = 2$ and $K = 14$. Note that the boundaries between clusters appear more often during the later part of the year, which matched our intuition that rapidly-changing fall weather patterns, combined with major holidays, might call for an uneven distribution of seasonal models. Unfortunately, we ran out of time before we could run experiments testing the accuracy of models based on these date ranges.

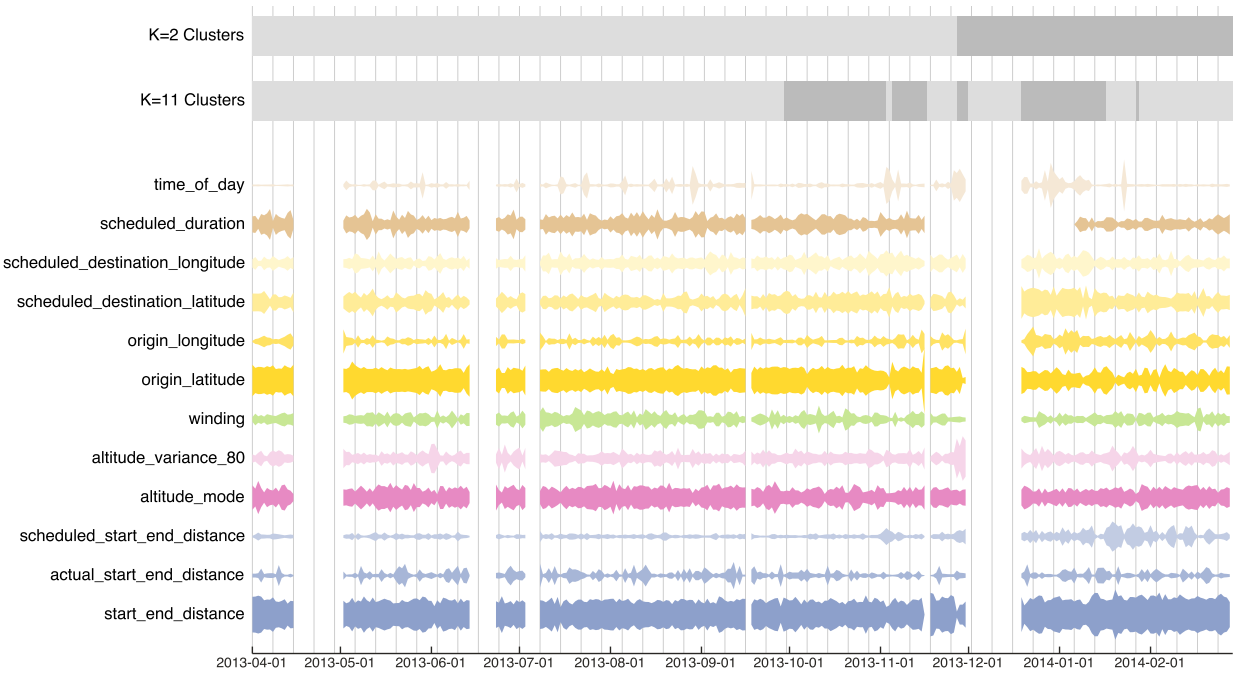


Figure 25. Feature importance clusters for $K = 2$ and $K = 14$, displayed above the raw feature importance values

7 Conclusions

For many cases, working in feature space rather than the physical space in which trajectories are embedded is a more effective way of finding trajectories that match a given set of criteria than using dynamic programming approaches that employ more local comparisons. This is partially due to computational issues, but very preliminary discussions have also indicated that these more global geometric features also generally correspond better to how people see trajectories. This is also more aligned with our overall goal of building tools for analysts to use to find trajectories that correspond to specific *behaviors* and not necessarily to narrowly-defined numerical quantities.

We anticipate that follow-on work will focus on two general areas. The first will center on computational improvements that include implementation on a database machine, a more thorough analysis of the information content in the different features, and examination of more efficient ways to break up the trajectories into segments to find smaller features. We also would like to work with analysts to understand better how people currently compare trajectories based on their experience.

References

[1] Google earth. https://www.google.com/earth. Accessed: 2015-09-09.

[2] Openflights airport database. http://openflights.org/data.html. Accessed: 2015-09-10.

[3] Python. http://www.python.org. Accessed: 2014-09-10.

[4] J. Alon, S. Sclaroff, G. Kollios, and V. Pavlovic. Discovering clusters in motion time-series data. In *Proc. Computer Vision and Pattern Recognition (CVPR '03)*, pages 375–381, 2003.

[5] B.G. Amidan and T.A. Ferryman. Atypical event and typical pattern detection within complex systems. In *Aerospace Conference, 2005 IEEE*, pages 3620–3631, March 2005.

[6] R. Annoni and C.H.Q. Forster. Analysis of aircraft trajectories using Fourier descriptors and kernel density estimation. In *Intelligent Transportation Systems (ITSC), 2012 15th International IEEE Conference on*, pages 1441–1446, Sept 2012.

[7] Robert E. Banfield, Lawrence O. Hall, Kevin W. Bowyer, and W. Philip Kegelmeyer. A comparison of decision tree ensemble creation techniques. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 29(1):173–180, January 2007.

[8] Faisal I. Bashir, Ashfaq A. Khokhar, and Dan Schonfeld. Object trajectory-based activity classification and recognition using hidden Markov models. *IEEE Trans. Image Process*, 2005, 2007.

[9] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[10] Donald J. Berndt and James Clifford. Using dynamic time warping to find patterns in time series. In Usama M. Fayyad and Ramasamy Uthurusamy, editors, *KDD Workshop*, pages 359–370. AAAI Press, 1994.

[11] Francesca Boem, Felice Andrea Pellegrino, Gianfranco Fenu, and Thomas Parisini. Multi-feature trajectory clustering using earth mover's distance. In *CASE*, pages 310–315. IEEE, 2011.

[12] Leo Breiman. Bagging predictors. *Machine Learning*, 24:123–140, 1996.

[13] Rich Caruana, Mohamed Elhawary, Art Munson, Mirek Riedewald, Daria Sorokina, Daniel Fink, Wesley M Hochachka, and Steve Kelling. Mining citizen science data to predict orevalence of wild bird species. In *Proc. of the ACM SIGKDD*, 2006.

[14] Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall, and W. Philip Kegelmeyer. SMOTE: Synthetic minority over-sampling technique. *Journal of Artificial Intelligence Research*, 16:321–357, 2002.

[15] Lei Chen, M. Tamer Özsu, and Vincent Oria. Robust and fast similarity search for moving object trajectories. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 491–502, New York, NY, USA, 2005. ACM.

[16] Ling Chen, Mingqi Lv, and Gencai Chen. A system for destination and future route prediction based on trajectory mining. *Pervasive Mob. Comput.*, 6(6):657–676, December 2010.

[17] G.M. Crippen and T.F. Havel. *Distance geometry and molecular conformation*. Chemometrics series. Research Studies Press, 1988.

[18] Defense Industry Daily. Too much information: Taming the UAV data explosion. *Defense Industry Daily*, May 16 2010.

[19] Y. Deguchi, K. Kuroda, M. Shouji, and T. Kawabe. Hev charge/discharge control system based on navigation information. In *SAE Convergence International Congress and Exposition On Transportation Electronics*, Detroit, Michigan USA, 2004.

[20] Ian L. Dryden and K.V. Mardia. *Statistical shape analysis*. Wiley series in probability and statistics: Probability and statistics. J. Wiley, 1998.

[21] Thomas Eiter and Heikki Mannila. Computing discrete Fréchet distance. Technical Report CD-TR 94/64, Technische Universitat Wien, April 1994.

[22] Jon Froehlich and John Krumm. Route prediction from trip observations. In *Society of Automotive Engineers (SAE) 2008 World Congress*, 2008.

[23] Bo Guan, Liangxu Liu, and Jinyang Chen. Using relative distance and Hausdorff distance to mine trajectory clusters. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 11(1):115–122, 2013.

[24] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM.

[25] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[26] Shohei Hido, Hisashi Kashima, and Yutaka Takahashi. Roughly balanced bagging for imbalanced data. *Statistical Analysis and Data Mining*, 2(5-6):412–426, 2009.

[27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 2015-09-04].

[28] Julian F. P. Kooij, Gwenn Englebienne, and Dariu M. Gavrila. A non-parametric hierarchical model to discover behavior dynamics from tracks. In *Proceedings of the 12th European Conference on Computer Vision - Volume Part VI*, ECCV'12, pages 270–283, Berlin, Heidelberg, 2012. Springer-Verlag.

[29] John Krumm and Eric Horvitz. Predestination: Inferring destinations from partial trajectories. In *Proceedings of the 8th International Conference on Ubiquitous Computing*, UbiComp'06, pages 243–260, Berlin, Heidelberg, 2006. Springer-Verlag.

[30] Ludmila (Lucy) Kuncheva. *Combining Pattern Classifiers: Methods and Algorithms*. Wiley-IEEE, 2004. ISBN 0471210781.

[31] DonaldJ. Patterson, Lin Liao, Dieter Fox, and Henry Kautz. Inferring high-level behavior from low-level sensors. In AnindK. Dey, Albrecht Schmidt, and JosephF. McCarthy, editors, *UbiComp 2003: Ubiquitous Computing*, volume 2864 of *Lecture Notes in Computer Science*, pages 73–89. Springer Berlin Heidelberg, 2003.

[32] Toby A. Patterson, Len Thomas, Chris Wilcox, Otso Ovaskainen, and Jason Matthiopoulos. State space models of individual animal movement. *Trends in Ecology & Evolution*, 23(2):87 – 94, 2008.

[33] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[34] Alexander V. Popov, Yury N. Vorobjev, and Dmitry O. Zharkov. MDTRA: A molecular dynamics trajectory analyzer with a graphical user interface. *Journal of Computational Chemistry*, 34(4):319–325, 2013.

[35] M. D. Rintoul and A. T. Wilson. Trajectory analysis via a geometric feature space approach. *to appear in Statistical Analysis and Data Mining*, 2016.

[36] R. T. Rockafellar and R. J.-B. Wets. *Variational Analysis*. Springer Verlag, 1998.

[37] Sandia National Labs. Tracktable, 2015. http://tracktable.sandia.gov.

[38] Ken Shoemake. Animating rotation with quaternion curves. *SIGGRAPH Comput. Graph.*, 19(3):245–254, July 1985.

[39] R. Simmons, B. Browning, Yilu Zhang, and V. Sadekar. Learning to predict driver route and destination intent. In *Intelligent Transportation Systems Conference, 2006. ITSC '06. IEEE*, pages 127–132, Sept 2006.

[40] K. Torkkola, K. Zhang, H. Li, H. Zhang, C. Schreiner, and M. Gardner. Traffic advisories based on route prediction. In *Workshop on Mobile Interaction with the Real World (MIRW 2007)*, Singapore, 2007.

[41] Stefan van der Walt, S Chris Colbert, and Gael Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

A Tracktable 0.9 Documentation

This section contains documentation for the Tracktable toolkit as of Version 0.9.

Tracktable Documentation

Release 0.9.1

Andy Wilson and Danny Rintoul

September 09, 2015

Tracktable's purpose is to load, assemble, analyze and render the paths traced out by moving objects. We combine the best tools and techniques we can find from both Python and C++ with the intent of making all of our capabilities easily accessible from both languages. We want to make it easy to...

- Render trajectories as histograms (heatmaps), track plots and movies.
- Run heavy-duty analysis in C++ and manipulate the results quickly in Python.
- Couple algorithms from top to bottom: - databases to store raw data, - filtering and cleaning techniques to assemble points into trajectories, - computational geometry to characterize them, - clustering and spatial data structures to find groups, and - visualization to help communicate your findings.
- Have fun!

We'll warn you up front that this is Tracktable's first release. Version 0.1 is definitely an alpha test. We've done our best to make everything run smoothly but it's safe to expect a few rough edges. Please tell us about them so that we can improve Tracktable!

CHAPTER

ONE

DOCUMENTATION

1.1 Installation

We currently use CMake to manage building Tracktable and running its tests. We hope to someday have a Python-centric install as simple as 'pip install tracktable'. For now, you'll need a little bit more expertise.

1.1.1 Step 0: Audience

We assume that you are familiar with downloading, compiling and installing software from source as well as with your operating system's package manager (if any). You will need to know how to set or modify environment variables, run the compiler and find libraries or header files on your system.

1.1.2 Step 1: Dependencies

Tracktable has the following required dependencies:

Python

- Python 2.7 - http://python.org
- numpy 1.7+ - http://numpy.org
- Matplotlib 1.2.1+ - http://matplotlib.org
- Basemap - http://matplotlib.org/basemap
- PyTZ - https://pypi.python.org/pypi/pytz/
- Shapely - https://pypi.python.org/pypi/Shapely

C++

- Compiler - GCC 4.4.7 or newer (http://gcc.gnu.org), clang 3.5 or newer (http://clang.llvm.org)
- Boost 1.57 or newer - http://www.boost.org
- GEOS library - http://geos.osgeo.org

Other

- CMake 2.8+ - http://www.cmake.org

If you want to build documentation you will also need the following packages:

- Sphinx - http://sphinx-doc.org
- napoleon - https://sphinxcontrib-napoleon.readthedocs.org/en/latest
- Breathe - http://breathe.readthedocs.org/en/latest/

If you want to render movies you will need FFMPEG:

- FFMPEG - https://www.ffmpeg.org - If you build from source please be sure to include the MPEG4 and FFV1 codecs. Both of these are included with the standard FFMPEG download. Tracktable can use other codecs but does not require them.

Build Notes for Dependencies

If you can possibly help it, install all the dependencies using package managers like `pip` (comes with Python), `yum`, `apt-get` (both of these are common in Linux environments), MacPorts (http://macports.org) or Homebrew (http://brew.sh). The notes in this section are for cases when you have no choice but to build external packages from source.

Building Python

The option we care about the most if you build your own Python is `--enabled-shared`. This will leave you with a shared library version of the Python C API. Tracktable will need to link against this.

Building Boost

We need several of Boost's compiled libraries including chrono, date_time, iostreams, log, random, timer and especially Boost.Python. As with other dependencies, check your operating system's package manager first. It's possible that you can install Boost with all its optional components from there.

If you already have a recent Boost installation you can check for Boost.Python by looking for files named `(prefix)boost_python.(suffix)` where (prefix) is `lib` on Unix-like systems and (suffix) is `.so` on Unix systems, `.so` or `.dylib` on Mac OSX and `.dll` (and `.lib`) on Windows.

If you really do have to build Boost from source – for example, if you had to build your own Python installation – then make sure to configure it to use the proper Python installation. Information about how to do this can be found in the Boost.Python documentation at http://www.boost.org/doc/libs/1_57_0/libs/python/doc/building.html

One final note: We know that it's often a pain to try to keep up with recent versions of a library as big as Boost. We will not require a newer version unless absolutely necessary.

Building FFMPEG

For up-to-date instructions on building FFMPEG please refer to https://trac.ffmpeg.org/wiki/CompilationGuide and choose your OS. We recommend that you compile in support for H264 video (via libx264). While this is not required, it is widely supported by current devices such as iPads, iPhones and Android systems.

You are now ready to configure and build the C++ part of Tracktable. Install the Python dependencies whenever convenient.

1.1.3 Step 2: Configuration

CMake enforces what we call "out-of-source" builds: that is, you cannot build object files alongside source code files. This makes it much easier to manage multiple build configurations. It also means that the first thing you must do is create a build directory. In the rest of this section we will use `TRACKTABLE_HOME` to refer to the directory where you unpacked the Tracktable source.:

```
$ cd TRACKTABLE_HOME
$ mkdir build
$ cd build
```

(You can also put your build directory anywhere else you please.)

Next, use CMake's configuration utility `ccmake` (or its GUI tool if you prefer) to configure compile settings:

If you made your build directory inside the source directory:

```
$ ccmake ..
```

If you made it someplace else:

```
$ ccmake TRACKTABLE_HOME/
```

Once CMake starts you will see a mostly empty screen with the message `EMPTY CACHE`. Press 'c' (if you use `ccmake`) or click 'Configure' (if you use the CMake GUI) to start configuration. After a moment, several new options will appear including `BUILD_PYTHON_WRAPPING` and `BUILD_SHARED_LIBS`. Leave these set to ON – without them you will not be able to use any of Tracktable's Python components. Set the value of `CMAKE_INSTALL_PREFIX` to the directory where you want to install the software. Press 'c' or click the 'Configure' button again to incorporate your choice.

Now you need to set options that are normally hidden. Press 't' or select the Show Advanced Options checkbox. Here are the variables you need to check:

1. `Boost_INCLUDE_DIR` and `Boost_LIBRARY_DIR`.

   These should point to your Boost 1.57 install with Boost.Python. Filenames for the `boost_date_time` and `boost_python` libraries should appear automatically.

   If you change either of these directories in CMake, press 'c' or click 'Configure' to make your changes take effect.

2. `PYTHON_EXECUTABLE`, `PYTHON_LIBRARY`, `PYTHON_INCLUDE_DIR`

   Make sure that all three of these point to the same installation. On Mac OSX with MacPorts in particular, CMake has a habit of using whatever Python executable is first in your path, the include directory from `/System/Library/Frameworks/Python.framework` and the library from `/usr/lib/`. MacPorts installs its Python library in `/opt/local/Library/Frameworks/Python.framework/Versions/2.7` with headers in `Headers/` and the Python library in `lib/libpython2.7.dylib`. If you have installed your own Python interpreter then use whatever path you chose for its installation.

   If you change any of these variables, press 'c' or click Configure' to make your changes take effect.

Now press 'g' or click 'Generate' to confirm all of your choices and generate Makefiles, Visual Studio project files or your chosen equivalent.

Note

Some older CMake installations have an odd bug that shows up with certain Linux installations. You may see `Boost_DIR` set to something like `/usr/lib64` no matter what value you try to set for `Boost_INCLUDE_DIR`

and Boost_LIBRARY_DIR. If you experience this, try adding the line:

```
set(Boost_NO_BOOST_CMAKE ON)
```

to TRACKTABLE_HOME/tracktable/CMakeLists.txt and then rerun CMake as described above.

1.1.4 Step 3: Build and Test

On Unix-like systems, type make. For Visual Studio, run nmake or open up the project files in your IDE (as appropriate).

Once the build process has finished go to your build directory and run ctest (part of CMake) to run all the tests. They should all succeed. Some of the later Python tests such as P_Mapmaker may take a minute or two.

Common Problems

1. Python tests crashing

   If the tests whose names begin with P_ crash, you probably have a mismatch between PYTHON_EXECUTABLE and PYTHON_LIBRARY. Check their values in ccmake / CMake GUI. If your Python executable is in (for example) /usr/local/python/bin/python then its corresponding library will usually be in /usr/local/python/lib/libpython2.7.so instead of halfway across the system.

2. Python tests running but failing

   - Cause #1: One or more required libraries missing.

     Check to make sure you have installed everything listed in the Dependencies section.

   - Cause #2: The wrong Python interpreter is being invoked.

     This really shouldn't happen: we use the same Python interpreter that you specify in PYTHON_EXECUTABLE and set PYTHONPATH ourselves while running tests.

3. Nearby stars go nova

   - We're afraid you're on your own if this happens.

1.1.5 Step 4: Install

You can use Tracktable as-is from its build directory or install it elsewhere on your system. To install it, type make install in the build directory (or, again, your IDE's equivalent).

You will also need to add Tracktable to your system's Python search path, usually stored in an environment variable named PYTHONPATH.

- If you are going to run Tracktable from the directory where you unpacked it then add the directory TRACKTABLE_HOME/tracktable/Python/ to your PYTHONPATH.
- If you installed Tracktable via make install then you will need to add INSTALL_DIR/Python/ to your PYTHONPATH. Here INSTALL_DIR is the directory you specified for installation when running CMake.

Finally, you will need to tell your system where to find the Tracktable C++ libraries.

- If you are running from your build tree (common during development) then the libraries will be in BUILD/lib and BUILD/bin (XXX Check where Windows puts its DLLs).
- If you are running from an installed location the libraries will be in INSTALL_DIR/lib and INSTALL_DIR/bin (XXX same check).

- On Windows, add the library directory to your `PATH` environment variable.
- On Linux and most Unix-like systems, add the library directory to your `LD_LIBRARY_PATH` environment variable.
- On Mac OSX, add the library directory to your `DYLD_LIBRARY_PATH` variable.

On Unix-like systems you can also add the library directory to your system-wide ld.so.conf file. That is beyond the scope of this document.

1.2 Using Tracktable With Python

For this first release we're going to focus on getting point data into Tracktable and rendered out as a heatmap, trajectory map or movie using the Python interface. We will add more sections to this user guide as we add capability to the toolkit.

1.2.1 Basic Classes

Point Domains

Tracktable operates on points. A trajectory is just a connected sequence of points. In order to do meaningful computations on points we have to specify two very important properties: what coordinate system describes the points and how shall we measure distances?

We embed this information in the points themselves. In Tracktable, a **point domain** is a combination of dimension, coordinate system and units. In each point domain we provide the following classes:

- BasePoint - Bare coordinates with no metadata
- TrajectoryPoint - Coordinates plus ID, timestamp and user-defined properties
- LineString - An ordered sequence of BasePoints
- Trajectory - A sequence of TrajectoryPoints with its own ID and properties

In this release of Tracktable the rendering pipeline (in Python) has full support for terrestrial and 2D Cartesian coordinates. We have partial support in Python for the 3D Cartesian domain.

NOTE: In this guide we will assume you are working with TrajectoryPoint data rather than BasePoint data and that you are in the terrestrial domain.

Timestamp

We use the Python notion of an *aware datetime* as our timestamp. This is a `datetime.datetime` (http://docs.python.org/library/datetime.html#datetime.datetime) object whose `tzinfo` field is not `None`.

The *`tracktable.core.Timestamp`* class contains several convenience methods for manipulating timestamps. A full list is in the reference documentation. We generally use the following ones most frequently.

- *`Timestamp.from_any`*: Try to convert whatever argument we supply into a timestamp. The input can be a `dict` (http://docs.python.org/library/stdtypes.html#dict), a `datetime` (http://docs.python.org/library/datetime.html#datetime.datetime), a string in the format `YYYY-MM-DD HH:MM:SS` or `YYYY-MM-DD HH:MM:SS+ZZ` (for a time zone offset from UTC).
- *`Timestamp.to_string`*: Convert a timestamp into its string representation. By default this will produce a string like `2014-08-28 13:23:45`. Optional arguments to the function will allow us to change the output format and include a timezone indicator.

Base Points

Within a domain, Tracktable uses the `BasePoint` class to store a bare set of coordinates. These behave like vectors or sequences in that we use square brackets to set and get coordinates:

```
from tracktable.domain.terrestrial import BasePoint

my_point = BasePoint()
my_point[0] = my_longitude
my_point[1] = my_latitude

longitude = my_point[0]
latitude = my_point[1]
```

Longitude is always coordinate 0 and latitude is always coordinate 1.

Regardless of how we get them, once we have a set of points we will want to do one of three things with them:

- Arbitrary further computation (whatever you want)
- Assemble points into trajectories (see next section)
- Render them onto a map as individual points or as a heatmap (see *Rendering*)

NOTE: In this initial version we only support points on the globe. There is code in the C++ library for points in Cartesian space that will be exposed (with luck) in the next release.

Trajectory Points

The things that make a point part of a trajectory are (1) its coordinates, already covered by BasePoint; (2) an identifier for the moving object, and (3) a timestamp recording when the object was observed. These are the main differences between BasePoint and TrajectoryPoint.

```
from tracktable.domain.terrestrial import TrajectoryPoint
from tracktable.core import Timestamp

my_point = TrajectoryPoint()
my_point[0] = my_longitude
my_point[1] = my_latitude

my_point.object_id = 'FlyingThing01'
my_point.timestamp = Timestamp.from_any('2015-02-01 12:23:56')
```

You may want to associate other data with a point as well. For example:

```
my_point.properties['altitude'] = 13400
my_point.properties['origin'] = 'ORD'
my_point.properties['destination'] = 'LAX'
my_point.properties['departure_time'] = Timestamp.from_any('2015-02-01 18:00:00')
```

For the most part you can treat the properties array like a Python dict. However, it can only hold values that are of numeric, string or Timestamp type.

Operations On Points

The module *`tracktable.core.geomath`* has most of the operations we want to perform on two or more points. Here are a few common ones. These work with both BasePoint and TrajectoryPoint unless otherwise noted.

- `distance_between(A, B)`: Compute distance between A and B

- `bearing(origin, destination)`: Compute the bearing from the origin to the destination
- `speed_between(here, there)`: Compute speed between two TrajectoryPoints
- `signed_turn_angle(A, B, C)`: Angle between vectors AB and BC
- `unsigned_turn_angle(A, B, C)`: Absolute value of angle between vectors AB and BC

Trajectories

Just as each domain has `BasePoint` and `TrajectoryPoint` classes, we include `LineString` and `Trajectory` for ordered sequences of points.

`LineString` is analogous to `BasePoint` in that it has no decoration at all. It is just a sequence of points. `Trajectory` has its own ID (`trajectory_id`) as well as its own properties array.

From within Python, a trajectory looks and acts exactly like a list of `TrajectoryPoint` objects. There are two different ways to create one:

```
from tracktable.domain.terrestrial import Trajectory

# Populate a trajectory as if it were an array
traj = Trajectory()
for point in mypoints:
    traj.append(mypoint)

# Populate a trajectory from an existing array of points
traj = Trajectory.from_position_list(mypoints)
```

These methods do effectively the same thing. Internally, `from_position_list` will allocate room for exactly as many points as we provide while `append` may allocate more.

Tracktable provides two kinds of trajectory source to help us create trajectories from scratch or assemble them from existing points. See the *Point Sources* and *Assembling Points into Trajectories* sections for details.

Tracktable expects that all points in a given trajectory will have the same object ID. Timestamps must not decrease from one point to the next.

Trajectories have several properties and methods. We expect that you will use the following ones most often:

- `object_id`: Return the ID of the first point. If the trajectory is empty (contains zero points) the string `(empty)` will be returned.
- `start_time`, `end_time`: Return the timestamp of the first and last points respectively. If the trajectory is empty these will return an invalid timestamp.
- `point_at_time(when: Timestamp)`: Given a timestamp, interpolate between points on the trajectory to find the point at exactly the specified time. Timestamps before the beginning or after the end of the trajectory will return the start and end points, respectively. Tracktable will try to interpolate all properties that are defined on the trajectory points.
- `subset_in_window(start, end: Timestamp)`: Given a start and end timestamp, extract the subset of the trajectory between those two times. The start and end points will be at exactly the start and end times you specify. These will be interpolated if there are no points in the trajectory at precisely the right time. Points in between the start and end times will be copied from the trajectory without modification.
- `recompute_speed`, `recompute_heading`: Compute new values for the `speed` and `heading` numeric properties at each point given the position and timestamp attributes. These are convenient if our original data set lacks speed/heading information or if the original values are corrupt.

1.2.2 Input

There are three ways to get point data into Tracktable in version 0.1.0. We can instantiate and populate TrajectoryPoint objects by hand (as shown in the code snippets above), use one of the algorithmic point sources, or load points from a delimited text file.

Loading Points from Delimited Text

Tracktable has a configurable point reader for delimited text files. Each point domain contains an appropriate instance of the `BasePointReader` and `TrajectoryPointReader` classes. As with the templates for the base point and trajectory point classes, we intend for you to interact with the fully-kitted-out versions instead of the raw template. This does mean that you have to decide what point domain your points belong to before you read them into your application.

Default File Format

By default, the trajectory point reader expects its input to be a comma-delimited text file containing the following fields in the first 4 columns:

- Column 0: Object ID. This can be a string of any length.
- Column 1: Timestamp. This must be formatted as 'YYYY-MM-DD HH:MM:SS'. For example, December 23, 2014 at 1:56:01 AM would be '2015-12-23 01:56:01'.
- Column 2: First coordinate. This is longitude in the terrestrial domain and X in the Cartesian domain.
- Column 3: Second coordinate. This is latitude in the terrestrial domain and Y in the Cartesian domain.

Assuming your file is in this format, code similar to the following will let you load points into memory. We will cover custom configurations in the next section.

```
from tracktable.domain.terrestrial import TrajectoryPointReader

my_reader = TrajectoryPointReader

with open('point_file.csv', 'rb') as infile:
   my_reader.input = infile

   for my_point in my_reader:
      # Do whatever you want with your points.
```

Custom File Format

It would be convenient if our data files always came in exactly the format expected by our libraries. Since it is not so, we allow you to change the delimiter and comment character and assign columns in your input file arbitrarily to properties on your points. In this example we will load terrestrial points that have coordinates, timestamps, IDs and the three properties "altitude", "origin" and "departure_time". Specifically, the object ID is in column 0, the timestamp is in column 1, longitude in column 12, latitude in column 20, altitude in column 21, origin in 22 and departure time in 23. Finally, just to throw in one more quirk, suppose that the columns in the data are separated by the pipe character (|) instead of commas.

```
from tracktable.domain.terrestrial import TrajectoryPointReader

with open('point_data.csv', 'rb') as infile:
    my_reader = TrajectoryPointReader()
```

```
    my_reader.input = infile

    my_reader.delimiters = "|"
    my_reader.object_id_column = 0
    my_reader.timestamp_column = 1
    my_reader.longitude_column = 12
    my_reader.latitude_column = 20

    my_reader.numeric_fields["altitude"] = 21
    my_reader.string_fields["origin"] = 22
    my_reader.timestamp_fields["departure_time"] = 23

    for point in my_reader:
         # Do whatever you want with the points here
```

Point Sources

There are algorithmic point generators in the `tracktable.source.path_point_source` module that are suitable for trajectory-building. The ones most likely to be useful are `GreatCircleTrajectoryPointSource` and `LinearTrajectoryPointSource`. Give them start and end points, start and end times, a number of points to generate and an object ID and you should be ready to go.

Assembling Points into Trajectories

Creating trajectories from a set of points is simple conceptually but logistically annoying when we write the code ourselves. The overall idea is as follows:

1. Group points together by object ID and increasing timestamp.
2. For each object ID, connect one point to the next to form trajectories.
3. Break the sequence to create a new trajectory whenever it doesn't make sense to connect two neighboring points.

This is common enough that Tracktable includes a filter (*tracktable.source.trajectory.AssembleTrajectoryFromPo* to perform the assembly starting from a Python iterable of points sorted by non-decreasing timestamp. We can specify two parameters to control part 3 (when to start a new trajectory):

- `separation_time`: A `datetime.timedelta` (http://docs.python.org/library/datetime.html#datetime.timedelta) specifying the longest permissible gap between points in the same trajectory. Any gap longer than this will start a new trajectory.
- `separation_distance`: Maximum permissible distance (in kilometers) between two points in the same trajectory. Any gap longer than this will start a new trajectory.

We can also specify a `minimum_length`. Trajectories with fewer than this many points will be silently discarded.

Example

```
trajectory_builder = AssembleTrajectoryFromPoints()
trajectory_builder.input = point_reader

trajectory_builder.separation_time = datetime.timedelta(minutes=30)
trajectory_builder.separation_distance = 100
trajectory_builder.minimum_length = 10
```

```
for traj in trajectory_builder.trajectories():
    # process trajectories here
```

Annotations

Once we have points or trajectories in memory we may want to annotate them with derived quantities for analysis or rendering. For example, we might want to color an airplane's trajectory using its climb rate to indicate takeoff, landing, ascent and descent. we might want to use acceleration, deceleration and rates of turning to help classify moving objects.

The module `tracktable.feature.annotations` contains functions to do this. Every feature defined in that package has two functions associated with it: a *calculator* and an *accessor*. The calculator computes the values for a feature and stores them in the trajectory. The accessor takes an already-annotated trajectory and returns a 1-dimensional array containing the values of the already-computed feature. This allows us to attach as many annotations to a trajectory as we like and then select which one to use (and how) at render time.

Todo

Code example for annotations

1.2.3 Rendering

Now we come to the fun part: making images and movies from data.

Tracktable 0.9.0 supports three kinds of visualization: a heatmap (2D histogram), a trajectory map (lines/curves drawn on the map) and a trajectory movie. We render heatmaps directly from points. Trajectory maps and movies require assembled trajectories.

In all cases we render points into a projection of some part of the globe. We use the Basemap (http://matplotlib.org/basemap) toolkit for the map projection and Matplotlib (http://matplotlib.org) for the actual rendering.

Setting Up a Map

The easiest way to create and decorate a map is with the `tracktable.render.mapmaker.mapmaker()` function. It can create maps of common (named) areas of the world, regions surrounding airports, and user-specified regions. Here's an example that will create a map of Australia with coastlines and longitude/latitude graticules rendered every 2 degrees.

```
from tracktable.render.mapmaker import mapmaker
from matplotlib import pyplot

f = pyplot.figure(size=(8, 6), dpi=100)

(mymap, initial_artists) = mapmaker('australia',
                                    draw_coastlines=True,
                                    draw_countries=False,
                                    draw_states=False,
                                    draw_lonlat=True,
                                    lonlat_spacing=2,
                                    lonlat_linewidth=0.5)
```

We always return two values from Mapmaker. The first is the `mpl_toolkits.basemap.Basemap` (http://matplotlib.org/basemap/api/basemap_api.html#mpl_toolkits.basemap.Basemap) instance that will convert

points between world coordinates (longitude/latitude) and map coordinates. The second is a list of Matplotlib `artists` (http://matplotlib.sourceforge.net/api/artist_api.html#matplotlib.artist.Artist) that define all the decorations added to the map.

There are several predefined map areas. Their names can be retrieved by calling `tracktable.render.maps.available_maps()`. If you would like to have a region included please send us its name and longitude/latitude bounding box. We will add it to the next release.

This map of Australia was generated by passing the map name `australia` to Mapmaker.

This next example will render a 200x200 km area around Boston's Logan Airport (BOS). Coastlines will be drawn with an extra-thick (2-point-wide) red line. US state borders will be drawn in blue. Land will be filled in using solid white. We use high-resolution borders since we're zoomed in fairly far.:

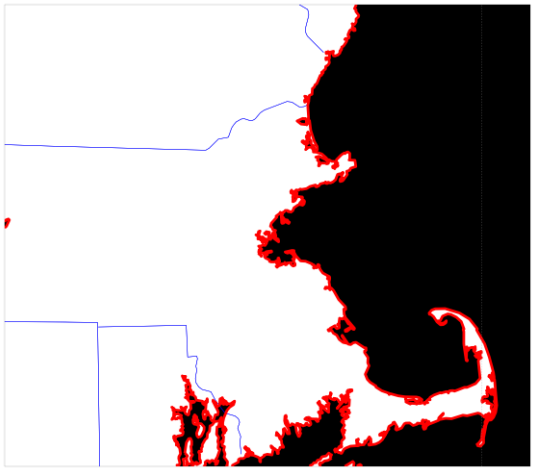
```
from tracktable.render.mapmaker import mapmaker
from matplotlib import pyplot

f = pyplot.figure(size=(8, 6), dpi=100)
(mymap, initial_artists) = mapmaker('airport:BOS',
                                    border_resolution='h',
                                    draw_coastlines=True,
                                    draw_states=True,
                                    land_color='white',
                                    coastline_color='red',
                                    coastline_linewidth=2,
                                    country_color='blue')
```

This map of the area around Boston's Logan Airport was generated by passing the map name `airport:BOS` to Mapmaker.

Note: The underlying `maps.map_for_airport()` function allows you to change the size of the displayed area from 200x200 km to anything you want. We will expose this parameter via Mapmaker in a future release. In the meantime, if you need that level of control we suggest using `map_name = 'custom'` and `map_bbox` to get the area you need.

If we want a map that does not correspond to any of the predefined ones then we can use the 'custom' map type. This example will create a map of Japan and the Korean Peninsula with all cities labeled whose population is larger than 2 million.



```
from tracktable.render.mapmaker import mapmaker
from matplotlib import pyplot

f = pyplot.figure(size=(8, 6), dpi=100)

# Bounding box is [ longitude_min, latitude_min,
#                   longitude_max, latitude_max ]
(mymap, initial_artists) = mapmaker(
           'custom',
           map_bbox = [ 123.5, 23.5, 148, 48 ],
           draw_cities_larger_than=2000000
      )
```

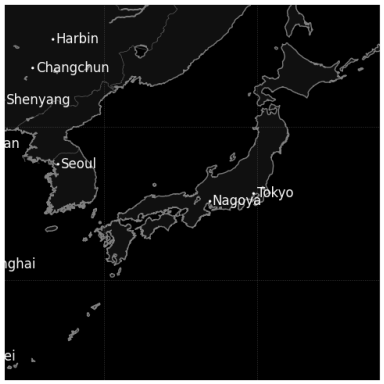


Fig. 1.1: This map was generated by passing the map name `custom` and a longitude/latitude bounding box to Mapmaker.

Note: To define a map area that crosses the discontinuity at longitude +/- 180 degrees, use coordinates that wrap around beyond 180. The bounding boxes (-200, 0, -160, 40) and (160, 0, 200, 40) both define a region that extends from 0 to 40 degrees latitude and 20 degrees to either side of 180 degrees longitude.

Todo

We haven't described how to set up a map projection for the Cartesian domain.

Rendering Onto the Map

Since Tracktable uses Matplotlib (http://matplotlib.org) as its underlying renderer you can immediately render almost anything you want on top of a map. Remember, however, that Matplotlib does not know about the map projection. In order to draw things that will be properly registered onto the map you need to use the `Basemap` (http://matplotlib.org/basemap/api/basemap_api.html#mpl_toolkits.basemap.Basemap) instance that we got earlier when we set up our map using Mapmaker. By calling the map instance as if it were a function you can convert coordinates from world space (longitude/latitude) to axis space (arbitrary coordinates established by Matplotlib).

There are many ways to draw things like contours, points, curves, glyphs and text directly onto the map. Please refer to the example gallery (http://matplotlib.org/basemap/users/examples.html) for demonstrations. Tracktable provides code to render two of the most common use cases for trajectory data: heatmaps (2D histograms) and trajectory maps.

Heat Maps

A *heat map* (Wikipedia page (http://en.wikipedia.org/wiki/Heat_map) is a two-dimensional histogram – that is, a density plot. We use heat maps to illustrate the density of points that compose a set of trajectories. We are typically looking for areas of high traffic and areas of coverage.

This release of Tracktable supports heat maps rendered on top of geographic maps using the `tracktable.render.histogram2d.geographic` function. You must call it with at least two arguments – a `Basemap` (http://matplotlib.org/basemap/api/basemap_api.html#mpl_toolkits.basemap.Basemap) instance and an iterable of points. Other optional arguments will let you control the histogram bin size, color map and where on the map the heatmap is rendered.

We include a start-to-finish example of how to load points and render a heat map in the `heatmap_from_csv.py` script in the `tracktable/examples/` subdirectory of our Python code. This example has *its own page* in the documentation.

Note: The `histogram2d.geographic()` heat map generator only traverses its input data once to keep memory requirements low. You can safely use it with point sets too large to load into memory at once.

Trajectory Maps

A *trajectory map* is an ordinary map with one or more trajectories drawn on it. We may want to decorate a trajectory with any of the following:

- Colors defined as a function of some quantity computed for the trajectory such as speed, turn rate or altitude
- Variable line widths (such as a trajectory that is broad at its head and narrow at its tail)
- A dot of some color and size at the head of the trajectory to mark the object's actual position
- A label at the head of the trajectory to display an object ID

All of this is packaged into the function *`draw_traffic`* in the *`tracktable.render.paths`* module.

Note: The argument names for that function are slightly misleading. Pay careful attention to the documentation for that function. Specifically, the arguments `trajectory_linewidth_generator` and `trajectory_scalar_generator` seem to indicate by their names that you must compute the linewidths and scalars at render time. This is fine for single images. For movies, we find it more useful to compute as much as we can before rendering and then pass an accessor function in as the generator.

Like *heat maps*, trajectory maps have their own example script `trajectory_map_from_csv.py` in the `tracktable/examples` directory. This script has *its own page* in the documentation.

Making Movies

To a first approximation, making a movie is the same as making a single image many, many times. The part that takes some care is minimizing the number of times we perform expensive operations such as loading data and configuring/decorating a map.

Our approach looks like this:

```
all_data = load_data()
figure = setup_matplotlib_figure()
setup_map_projection(figure)
movie_writer = setup_movie_writer()

with movie_writer.saving(figure, 'movie_filename.mp4'):
    for frame_num in xrange(num_frames):
        frame_data = render_frame(frame_num, all_data)
        movie_writer.grab_frame()
        cleanup_frame(frame_data)
```

The setup phase is exactly the same as it would be if we were rendering a single image. The conceptual differences are in `render_frame()`, which must take into account which frame it's drawing, and `cleanup_frame()`, which restores the drawing area to its beginning-of-frame state. We adopt the convention that `render_frame()` shall return a list of all Matplotlib artists that were added to the figure while rendering the current frame. That way we can clean up by removing each artist after the frame has been saved by a call to `movie_writer.grab_frame()`.

Although Matplotlib supports several different animation back ends including live on-screen animation, Mencoder, FFMPEG, ImageMagick, Tracktable 0.9.0 only supports the FFMPEG back end. There are two reasons. First, FFMPEG is available for nearly all platforms and is quite capable. By supporting it before any others we can help as many users as possible render movies as quickly as possible. Second, FFMPEG has a few extra capabilities that make it well suited to rendering movies in parallel.

Please refer to the files `example_movie_rendering.py`, `movie_from_csv.py` and `parallel_movie_from_csv.py` in the directory `tracktable/Python/tracktable/examples` for an illustration of how to render a movie. More thorough documentation will follow soon.

1.2.4 Command Line

Tracktable's various rendering facilities have a lot of options. Python makes it easy for us to expose these as command-line options that can be passed to scripts. However, that just pushes the problem out one level: now the user has to remember the values for all of those options, or else write shell scripts that call Python scripts in order to keep track of what parameters were used where.

We introduce two facilities to help tame this morass:

1. **Argument Groups**: An argument group is a set of command-line arguments that all pertain to a single capability. For example, the argument group for trajectory assembly has entries for the maximum separation distance, maximum separation time and minimum length as described above in *Assembling Points into Trajectories*.
2. **Response Files**: A response file is a way to package up arbitrarily many command-line arguments in a file and pass them to a script all at once. It is independent of which script is being run. Since a response file is just text it is easy to place under version control. We provide a slightly modified version of the standard Python `argparse` (http://docs.python.org/library/argparse.html#module-argparse) module that includes support for response files containing comments and response files that load other response files.

Argument Groups

The point of an argument group is to save us from having to cut and paste the same potentially-lengthy list of arguments and their respective handlers into each new script we write. When we render a movie of data over time, for example, we will always need several pieces of information including resolution, frame rate, and the duration of our movie.

Since we're human we are guaranteed to forget an argument here, spell one differently there, and before long we have a dozen scripts that all take completely different command-line arguments. Bundling arguments in an easy-to-reuse fashion makes it easy for us to use the same ones consistently.

We derive another benefit at the same time. By abstracting away a set of arguments into a semi-opaque module, we can add capability to (for example) the mapmaker without having to change our movie-making script. Once the argument group for the mapmaker is updated, any script that uses the mapmaker's argument group will automatically gain access to the new capability.

There are three parts to using argument groups. First they must be created and registered. Second, they are applied when we create an argument parser for a script. Finally, once command-line arguments have been parsed, we (as the programmers) can extract values for each argument group that you used. All of these functions are in the `tracktable.script_helpers.argument_groups.utilities` module.

Creating an Argument Group

We create an argument group first by declaring it with `create_argument_group()` and then populating it with calls to `add_argument()`. Here is an example from the `movie_rendering` group:

```
create_argument_group("movie_rendering",
                      title="Movie Parameters",
                      description="Movie-specific parameters such as frame rate, encoder options, tit

add_argument("movie_rendering", [ "--duration" ],
             type=int,
             default=60,
             help="How many seconds long the movie should be")

add_argument("movie_rendering", [ "--fps" ],
             type=int,
             default=30,
             help="Movie frame rate in frames/second")

add_argument("movie_rendering", [ "--encoder-args" ],
             default="-c:v mpeg4 -q:v 5",
             help="Extra args to pass to the encoder (pass in as a single string)")
```

All of Tracktable's standard argument groups are in files in the `Python/tracktable/script_helpers/argument_groups` directory. Look at `__init__.py` in that directory for an example of how to add one to the registry. You can register your own groups anywhere in your code that you choose.

Applying Argument Groups

We use argument groups by applying their arguments to an already-instantiated argument parser. That can be an instance of the standard `argparse.ArgumentParser` (http://docs.python.org/library/argparse.html#argparse.ArgumentParser) or our customized version *`tracktable.script_helpers.argparse.ArgumentParser`*. Here is an example:

```
from tracktable.script_helpers import argparse, argument_groups

parser = argparse.ArgumentParser()
argument_groups.use_argument_group("delimited_text_point_reader", parser)
argument_groups.use_argument_group("trajectory_assembly", parser)
argument_groups.use_argument_group("trajectory_rendering", parser)
argument_groups.use_argument_group("mapmaker", parser)
```

We can interleave calls to `use_argument_group()` freely with calls to other functions defined on `ArgumentParser` (http://docs.python.org/library/argparse.html#argparse.ArgumentParser). We recommend reading the code for `use_argument_group()` if you need to do especially complex things with `argparse` such as mutually exclusive sets of options.

Using Parsed Argument Values

After we call `parser.parse_args()` (http://docs.python.org/library/argparse.html#argparse.ArgumentParser.parse_args) we are left with a `Namespace` object containing all the values for our command-line options, both user-supplied and default. We use the `extract_arguments()` function to retrieve sets of arguments that we configured using `use_argument_group()`. Our practice is to define handler functions that take every argument in a group so that we can write code like the following:

```
def setup_trajectory_source(point_source, args):
    trajectory_args = argument_groups.extract_arguments("trajectory_assembly", args)
    source = example_trajectory_builder.configure_trajectory_builder(
        **trajectory_args
       )
    source.input = point_source

    return source.trajectories()
```

Since we are not required to refer to the individual arguments directly the user can take advantage of new capabilities added to the underlying modules whether or not we know about them when we write our script.

Todo

Add tracktable.script_helpers.argument_groups to the documentation

---

Response Files

Todo

Document response files in full

Once we start calling scripts with more than 3 or 4 options it becomes difficult to keep track of all the arguments and difficult to edit the command line. We address this with *response files*, textual listings of command-line options and their values that we can pass to scripts. The standard Python `argparse` module has limited support for response files. We expand upon it with our own extended `argparse`.

Fuller documentation is coming soon. This should be enough to get you started:

```
$ cd tracktable/Python/tracktable/examples
$ python heatmap_from_csv.py --write-response-file > heatmap_response_file.txt
```

Now open up `heatmap_response_file.txt` in your favorite editor. Lines that begin with # are comments. Uncomment any arguments you please and add or change values for them. After you save the file, run the script as follows:

```
$ python heatmap_from_csv.py @heatmap_response_file.txt
```

That will tell the script to read arguments from `heatmap_response_file.txt` as well as from the command line.

You can freely mix response files and standard arguments on a single command line. You can also use multiple response files. The following command line would be perfectly valid:

```
$ python make_movie.py @hd_movie_params.txt @my_favorite_map.txt movie_outfile.mkv
```

Have fun!

1.3 Examples

To help you get started using Tracktable we have included sample data files and scripts to render a heatmap, a trajectory map and a trajectory movie. You will need to have ffmpeg installed to render movies. The heatmap and trajectory map will work with or without ffmpeg.

All of these scripts will be run from the command line. Before you begin you should build Tracktable (see *Installation*) and make sure its tests run successfully.

In the examples below we assume that `TRACKTABLE` is the root of the Tracktable Python source code. This is either `...SOURCE_DIR/tracktable/Python/tracktable` when running from the source tree or `...INSTALL_DIR/Python/tracktable` if installed elsewhere. We further assume that `python` is whichever Python executable you specified at build time.

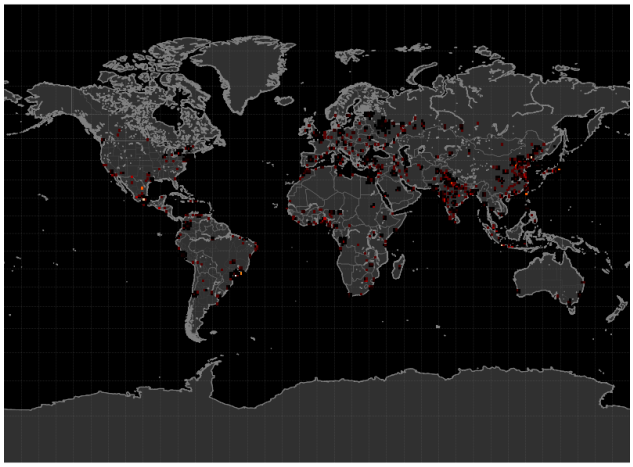
1.3.1 Heat Map (2D Histogram)

The simplest display type that Tracktable supports is the 2-dimensional histogram or heatmap (http://en.wikipedia.org/wiki/Heat_map). It requires points that contain longitude/latitude coordinates. The points can contain any number of other attributes but they will be ignored.

Run the example as follows:

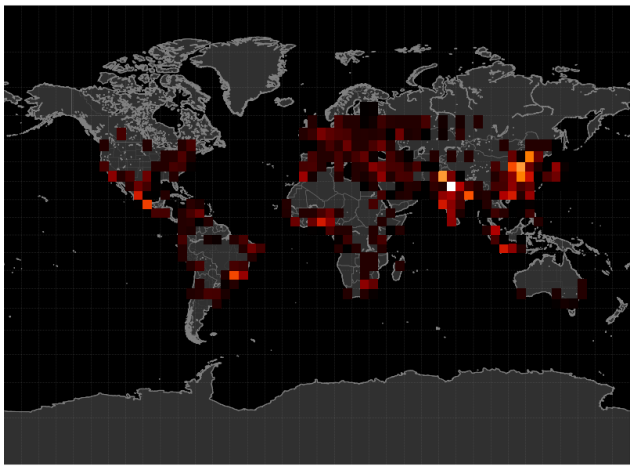
```
$ python TRACKTABLE/examples/heatmap_from_csv.py TRACKTABLE/examples/data/SampleHeatmapPoints.tsv Hea
```

Open the resulting image (`HeatmapExample1.png`) in your favorite image viewer. You will see a map of the Earth with a smattering of red and yellow dots. These are our example points, all generated in the neighborhood of population centers.



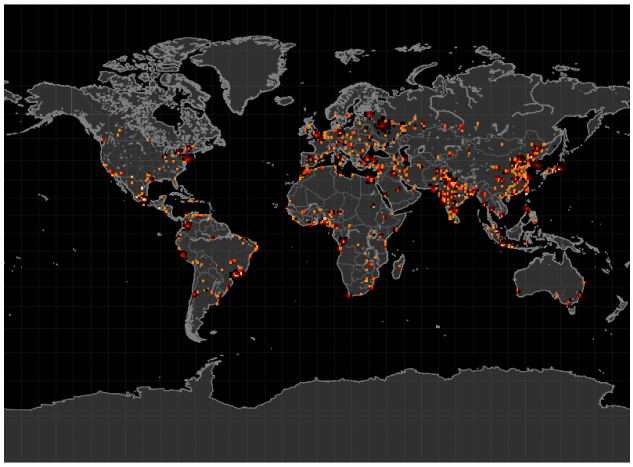
Now it's time to change things around. Let's suppose that you want to see larger-area patterns with a coarser distribution. You can change the histogram resolution with the `--histogram-bin-size` argument:

```
$ python TRACKTABLE/examples/heatmap_from_csv.py --histogram-bin-size 5 TRACKTABLE/examples/data/Samp
```



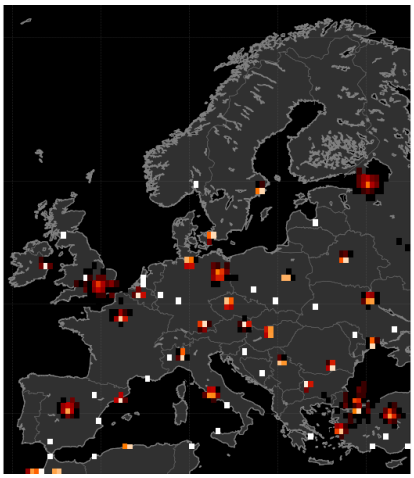
Perhaps when you open up that image you find that the bins are now too large. The earlier size was good but the histogram is too sparse. If you change the color map to use a logarithmic scale instead of a linear one you might get more detail:

```
$ python TRACKTABLE/examples/heatmap_from_csv.py --scale logarithmic TRACKTABLE/examples/data/SampleI
```



That doesn't help much. What if we zoom in on Europe and make the bins smaller?

```
$ python TRACKTABLE/examples/heatmap_from_csv.py --scale logarithmic --map europe --histogram-bin-si
```



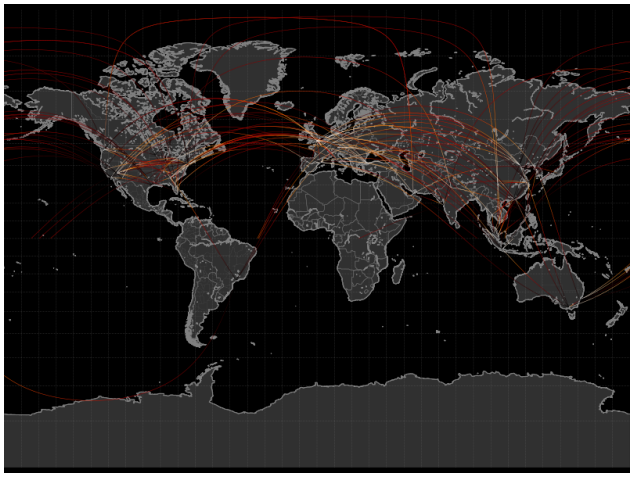
There are many more options that you can change including map region, decoration, colors, image resolution and input configuration. You can get a full list of options with the `--help` argument:

```
$ python TRACKTABLE/examples/heatmap_from_csv.py --help
```

1.3.2 Trajectory Map

As soon as we add timestamps to our (longitude, latitude) points we can sensibly assemble sequences of points into trajectories. Trajectories lend themselves to being plotted as lines on a map. That's our second example. We have provided a sample data set of trajectories between many of the world's busiest airports for you to use.

```
$ python TRACKTABLE/examples/trajectory_map_from_csv.py
    TRACKTABLE/examples/data/SampleTrajectories.tsv
    TrajectoryMapExample1.png
```

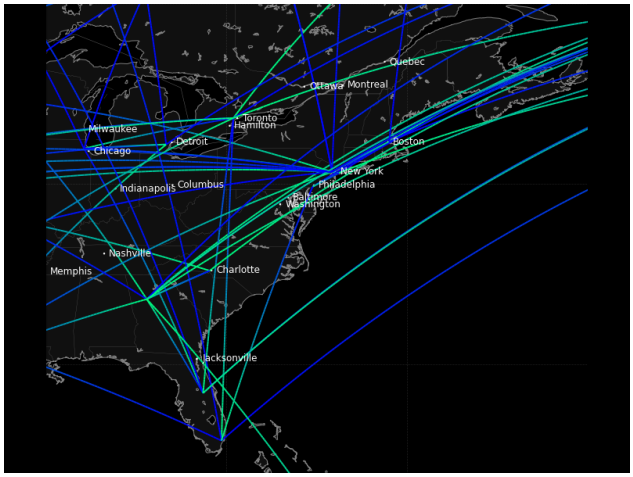


The trajectories are colored according to the 'progress' feature that ranges from 0 at the beginning of a trajectory to 1 at its end. However, the thin lines make them difficult to see with this resolution and color map. Let's make the lines for the trajectories wider and change the color map.

```
$ python TRACKTABLE/examples/trajectory_map_from_csv.py
    --trajectory-linewidth 2
    --trajectory-colormap winter
    TRACKTABLE/examples/data/SampleTrajectories.tsv
    TrajectoryMapExample2.png
```

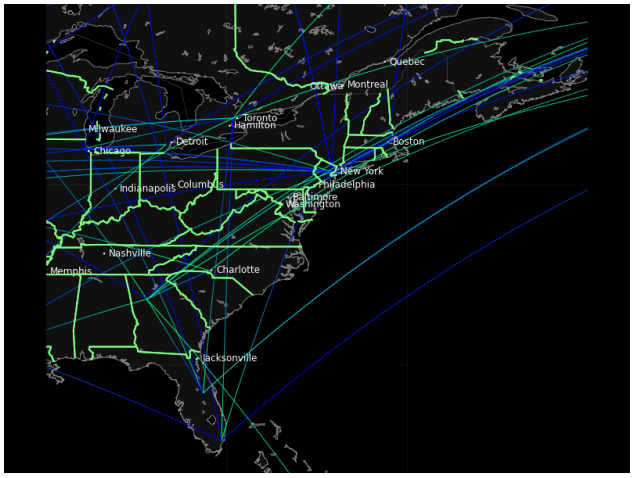
Just for the sake of argument, let's zoom in on the eastern US. We don't have a predefined map for that but we can come up with a bounding box. We want the region from (-90, 24) to (-60, 50). Recall that in our longitude-first convention that's (90W, 24N) to (60W, 50N). While we're at it, let's also draw and label every city with a population over half a million people.

```
$ python TRACKTABLE/examples/trajectory_map_from_csv.py
    --trajectory-linewidth 2
    --trajectory-colormap winter
    --map custom
    --map-bbox -90 24 -60 50
    --draw-cities-larger-than 500000
    TRACKTABLE/examples/data/SampleTrajectories.tsv
    TrajectoryMapExample3.png
```



Last and not least, let's highlight the borders of the US states and Canadian provinces in bright green lines 2 points wide. We'll also decrease the trajectory width so that the city labels aren't so overwhelmed.

```
$ python TRACKTABLE/examples/trajectory_map_from_csv.py
    --state-color #80FF80
    --state-linewidth 2
    --trajectory-linewidth 1
    --trajectory-colormap winter
    --map custom
    --map-bbox -90 24 -60 50
    --draw-cities-larger-than 500000
    TRACKTABLE/examples/data/SampleTrajectories.tsv
    TrajectoryMapExample3.png
```



This result is not going to win any beauty contests but you've now seen a few more options available to you. Tracktable allows you to change the presence, appearance and style of boundaries for continents, countries and states (US/Canada only at present). You can filter and draw city locations by population (given some minimum threshold) or by ranking. You can change the line style, appearance and color map for the rendered trajectories. All of this is explained in the *Using Tracktable With Python* and the *Reference Documentation*.

Movies

To render a movie, we render short subsets of trajectories over and over. As such we can re-use all of the arguments and algorithms we already have for rendering trajectory maps with just a few additions for movie duration, frames per

second, and trajectory length.

We'll begin with a short movie (10 seconds long, 10 frames per second) where each moving object has a trail showing the last hour of its motion:

```
$ python TRACKTABLE/examples/movie_from_csv.py
    --trail-duration 3600
    --trajectory-linewidth 2
    --fps 10
    --duration 10
    TRACKTABLE/examples/data/SampleTrajectories.tsv
    MovieExample1.mp4
```

This will encode a movie using vanilla MPEG4 that will be playable by anything less than ten years old. Quicktime Player (http://www.apple.com/quicktime/download/), iTunes (http://www.apple.com/itunes), and Windows Media Player (http://windows.microsoft.com/en-us/windows/download-windows-media-player) can all handle this. If you don't already have VLC (http://www.videolan.org) installed we recommend that as well.

We have two more features to demonstrate here. First, instead of having the trajectory lines be of constant width along their length we can have them taper as they get older. We do this with `--trajectory-width taper`, `trajectory-initial-linewidth` and `trajectory-final-linewidth`. We will also put a dot at the head of each trajectory with `--decorate-trajectory-head` and `trajectory-head-dot-size`.

```
$ python TRACKTABLE/examples/movie_from_csv.py
   --trail-duration 3600
   --trajectory-linewidth taper
   --trajectory-initial-linewidth 3
   --trajectory-final-linewidth 0
   --decorate-trajectory-head
   --trajectory-head-dot-size 3
   --fps 10
   --duration 10
   TRACKTABLE/examples/data/SampleTrajectories.tsv MovieExample2.mp4
```

Too Many Arguments!

Todo

Document response files here.

1.4 Reference Documentation

Todo

In a future release there will be a detailed explanation of the differences between the C++ and Python interfaces and how to go back and forth between them.

1.4.1 Python Interface

Right now Tracktable's functions are accessible principally via the Python interface. We prefer to implement things in Python first for ease, speed and malleability, then choose parts to re-implement in C++ based on speed, memory usage and algorithmic needs.

tracktable package

Subpackages

tracktable.core module

Module contents TrailMix Trajectory Library - Core

Basic types (point, trajectory) live in this module. They are in turn imported from the small C extension libraries.

class `tracktable.core.`**`TrajectoryPoint`**(*longitude=0*, *latitude=0*, *altitude=0*)
Bases: `object` (http://docs.python.org/library/functions.html#object)

This is the core geographic point class for Tracktable. NOTE: The 'Bases' is not accurate but will do for now.

This is the core geographic point class for Tracktable.

TrajectoryPoint specifies a point on the surface of a sphere that is annotated with an object ID, coordinates, altitude, heading and speed. Any or all of these may be left uninitialized depending on the user's actions although a point without coordinates is not especially useful.

The altitude value defaults to zero. Leave it this way when representing objects such as automobiles, trains or surface ships for which altitude doesn't matter.

Note: TrajectoryPoint is implemented in C++ and exposed to Python via the Boost.Python module. I will include a link to the C++ class definition as soon as I figure out how to do so within Sphinx, Breathe and Napoleon.

`longitude`
float

Longitude of point (between -180 and 180)

`latitude`
float

Latitude of point (between -90 and 90)

`object_id`
string

Object ID of entity referred to by this point

`timestamp`
datetime

Timestamp corresponding to this point

`speed`
float

Current speed of object in kilometers per hour

`heading`
float

Current heading of object in degrees. 0 degrees is due north and angles increase clockwise (90 degrees is due east).

`altitude`
float

Current altitude of object in feet.

TODO: Convert to meters.

properties
dict

User-defined properties. Names are strings, values are numbers, timestamps and strings.

TrajectoryPoint.**has_property**(*property_name*)
Check to see whether an entry is present in the property map

Parameters **property_name** (*string* (http://docs.python.org/library/string.html#module-string)) – Name of property to look for

Returns True or false depending on whether property was found

class tracktable.core.**Trajectory**
Bases: object (http://docs.python.org/library/functions.html#object)

Ordered sequence of timestamped points.

This class is the heart of most of what Tracktable does. It implements an ordered sequence of TrajectoryPoint objects, each of which has an ID, coordinates and a timestamp. Those compose a trajectory. All points in a single Trajectory should have the same *object_id*.

Note: The Trajectory class is implemented in C++ and exposed to Python via the Boost.Python module. We use the vector indexing suite from Boost to make a trajectory act like a list of points.

---

object_id
string

ID of object described in this trajectory. This will be the string "(empty)" if the trajectory does not contain any points.

start_time
datetime

Timestamp of the first point in the trajectory. This will be invalid if the trajectory contains no points.

end_time
datetime

Timestamp of the last point in the trajectory. This will be invalid if the trajectory contains no points.

Trajectory.**subset_in_window**(*start_time*, *end_time*)
Compute the subset of a trajectory between two timestamps.

We often want to clip a trajectory to just the part that fits within a certain window of time. This method interpolates new start and end positions that fall exactly at *start_time* and *end_time* and includes between them all of the points in the trajectory that occur within the specified interval. If the trajectory does not intersect the interval at all then the return value will be an empty Trajectory.

Parameters

- **start_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Beginning of time window
- **end_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – End of time window

Returns New Trajectory with endpoints at *start_time* and *end_time*

Trajectory.**point_at_time**(*when*)
Find the position on a trajectory at a specified time.

If the specified time falls between points on the trajectory then we will interpolate as needed. If you ask for a point before the trajectory begins or after the trajectory ends you will get the first or last point of the trajectory respectively.

Parameters **when** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Time at which to evaluate trajectory

Returns TrajectoryPoint at specified time

Trajectory.**recompute_speed**()
Compute speeds at each point of a trajectory.

Sometimes the speed measurements that come with a data set are noisy or outright wrong. Sometimes they're missing. This method is for those situations. When you call it, it will use the position and timestamp data to compute a speed at each point in the trajectory.

Trajectory.**recompute_heading**()
Compute headings at each point of a trajectory.

The heading at each point of a trajectory is the direction to the next point. The heading at the very last point is the same as the heading at the penultimate point.

Trajectory.**intersects_box**(*southwest_corner*, *northeast_corner*)
Test to see whether a trajectory intersects a rectangle in longitude/latitude space.

A trajectory intersects a box if any of its coordinates falls within the box.

Parameters

- **southwest_corner** (*TrajectoryPoint or (longitude, latitude) tuple*) – Southwest corner of box of interest
- **northwest_corner** (*TrajectoryPoint or (longitude, latitude) tuple*) – Northwest corner of box of interest

Returns True or False

static Trajectory.**from_position_list**(*points*)
Assemble a Trajectory from a list of TrajectoryPoints.

Parameters **points** (*list* (http://docs.python.org/library/functions.html#list)) – List of TrajectoryPoint objects

Returns New Trajectory instance containing copies of all supplied points

class tracktable.core.**Timestamp**
Convenience methods for working with timestamps.

Recall that a Timestamp is just a timezone-aware datetime.datetime (http://docs.python.org/library/datetime.html#datetime.datetime) object. That gives us all of the standard Python date/time functions to use for manipulating them. As a result we only need convenience functions to take care of the awkward parts: turning strings into timestamps, timestamps into strings, and imbuing a naive datetime object with a time zone.

static Timestamp.**beginning_of_time**()
Return a timestamp equal to January 1, 1400. No valid timestamp should ever be this old.

Returns Timestamp object set to the first day of 1400

static Timestamp.**from_struct_time**(*mytime*)
Construct a datetime from a time.struct_time object.

Parameters **mytime** (time.struct_time (http://docs.python.org/library/time.html#time.struct_time)) – Source time

Returns An aware datetime object imbued with tracktable.core.timestamp.DEFAULT_TIMEZONE

static Timestamp.**from_dict**(*mydict*)

Construct a datetime from a dict with named elements.

Parameters **mydict** (dict (http://docs.python.org/library/stdtypes.html#dict)) – Dict with zero or more of 'hour', 'minute', 'second', 'year', 'month', 'day', and 'utc_offset' entries. Missing entries will be set to their minimum legal values.

Returns An aware datetime object imbued with tracktable.core.DEFAULT_TIMEZONE unless a 'utc_offset' value is specified, in which case the specified time zone will be used instead.

static Timestamp.**from_datetime**(*mytime*)

Convert a datetime (http://docs.python.org/library/datetime.html#datetime.datetime) into an aware timestamp.

Parameters **mytime** (datetime.datetime (http://docs.python.org/library/datetime.html#datetime.datetime)) – datetime object to convert

Returns datetime that will definitely have a time zone attached. If the input already has a time zone then it will be returned without modification. If not, a new timestamp will be created with the supplied date/time and the default time zone.

static Timestamp.**from_any**(*thing*)

Try to construct a timestamp from whatever we're given.

The possible inputs can be:

- a Python datetime (in which case we just return a copy of the input)
- a string in the format 2013-04-05 11:23:45, in which case we will assume that it resides in timestamp.DEFAULT_TIMEZONE
- a string in the format 2013-04-05 11:23:45-05, in which case we will assume that it's UTC-5 (or other time zone, accordingly)
- a string in the format 2013-04-05T11:23:45 or 2013-04-05T11:23:45-05 – just like above but with a T in the middle instead of a space
- a string in the format 20130405112345 - these are assumed to reside in the default time zone
- a string in the format MM-DD-YYYY HH:MM:SS
- a string such as 08-Aug-2013 12:34:45 where 'Aug' is the abbreviated name for a month in your local environment
- a dict containing at least year, month, day entries and optionally hour, minute, second and utc_offset

Internally this method dispatches to all of the other Timestamp.from_xxx() methods.

Parameters **thing** – Thing to try to convert to a timestamp

Returns Timezone-aware datetime (http://docs.python.org/library/datetime.html#datetime.datetime) object

static Timestamp.**to_string**(*dt*, *format_string='%Y-%m-%d %H:%M:%S'*, *include_tz=False*)

Create a string from a timestamp.

Format contents as a string, by default formatted as `2013-04-21 14:45:00`. You may supply an argument `format_string` if you want it in a different form. See the documentation for `datetime.strftime()` for information on what this format string looks like.

Parameters

- **dt** (`datetime.datetime` (http://docs.python.org/library/datetime.html#datetime.datetime)) – Timestamp object to stringify
- **format_string** (*string* (http://docs.python.org/library/string.html#module-string)) – String to pass to `datetime.strftime()`
- **include_tz** (*Boolean*) – Whether or not to append timezone as UTC offset

Returns String representation of timestamp

tracktable.feature module

Submodules

tracktable.feature.annotations module tracktable.features.annotations:

Annotate points or trajectories (or the points in a trajectory) with useful derived quantities

`tracktable.feature.annotations.`**available_annotations**`()`

`tracktable.feature.annotations.`**climb_rate**(*trajectory*, *max_climb=2000*)
climb_rate: Annotate points in an AirTrajectory with climb rate

usage: climb_rate(t: AirTrajectory) -> None

This will add a property 'climb_rate' to each point in the input trajectory. This is measured in units/sec and is computed as (points[n].altitude - points[n-1].altitude) / (points[n].timestamp - points[n-1].timestamp).

`tracktable.feature.annotations.`**compute_speed_from_positions**(*trajectory*)

`tracktable.feature.annotations.`**get_airspeed**(*trajectory*, *min_speed=0*, *max_speed=980*)
Return a vector of scalars for point-to-point speeds

This is a feature accessor that can be used to color a trajectory. It will map the 'speed' property into a range from 0 to 1.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Kwargs: min_speed (float): Minimum speed in kilometers per hour. This will be mapped to the bottom of the scalar range and thus the bottom of the color map. Defaults to 0. max_speed (float): Maximum speed in kilometers per hour. This will be mapped to the top of the scalar range and thus the top of the color map. Defaults to 980 (0.8 Mach, a common maximum permitted speed for civilian airliners).

Returns A vector of scalars that can be used as input to a colormap.

`tracktable.feature.annotations.`**get_climb_rate**(*trajectory*, *max_velocity=2000*)

`tracktable.feature.annotations.`**get_progress**(*trajectory*)

`tracktable.feature.annotations.`**get_speed**(*trajectory*)
Get the speed for a trajectory without any scaling.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Returns Numpy array containing the speed value for each point

tracktable.feature.annotations.**get_speed_over_water**(*trajectory*, *min_speed=0*, *max_speed=60*)

Return a vector of scalars for point-to-point speeds over water

This is a feature accessor that can be used to color a trajectory. It will map the 'speed' property into a range from 0 to 1.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Kwargs: min_speed (float): Minimum speed in kilometers per hour. This will be mapped to the bottom of the scalar range and thus the bottom of the color map. Defaults to 0. max_speed (float): Maximum speed in kilometers per hour. This will be mapped to the top of the scalar range and thus the top of the color map. Defaults to 60 km/h (32 knots, very fast for big ships but slower than the maximum speed of high-speed civilian ferries).

Returns A vector of scalars that can be used as input to a colormap.

tracktable.feature.annotations.**progress**(*trajectory*)

progress: Annotate points in an AirTrajectory with flight progress

usage: progress(t: AirTrajectory) -> None

This will add a property "progress" to each point in the input trajectory. This property will be 0 at the first point, 1 at the last point, and spaced evenly in between.

tracktable.feature.annotations.**register_annotation**(*feature_name*, *compute_feature*, *retrieve_feature*)

tracktable.feature.annotations.**retrieve_feature_accessor**(*name*)

tracktable.feature.annotations.**retrieve_feature_function**(*name*)

Module contents TrailMix Trajectory Library - Features module

All the code for dealing with features (derived quantities) on points and trajectories lives in this module.

tracktable.filter module

Submodules

tracktable.filter.trajectory module

Module contents tracktable.filter.trajectory - Filters that take trajectories as input

class tracktable.filter.trajectory.**ClipToTimeWindow**

Bases: object (http://docs.python.org/library/functions.html#object)

Truncate trajectories to fit within a time window

Given an iterable of Trajectory objects, return those portions of each trajectory that fit within the specified time window. Interpolate endpoints as necessary.

input

iterable

Source of Trajectory objects

start_time
datetime

Beginning time for window of interest

end_time
datetime

End time for window of interest

trajectories()
Return sub-trajectories within window.

Note: Since this is a generator, you can only traverse the sequence once unless you collect it in a list yourself.

Yields Trajectories derived from input trajectories. Each trajectory is guaranteed to fall entirely within the window specified by self.start_time and self.end_time. If one of the input trajectories extends beyond that boundary, a new endpoint will be interpolated so that it begins or ends precisely at the boundary. Trajectories entirely outside the boundary will be returned as empty trajectories with 0 points.

class tracktable.filter.trajectory.**FilterByAltitude**
Bases: object (http://docs.python.org/library/functions.html#object)

Filter out trajectories that don't intersect an interval of altitude

Given a source that produces Trajectories, return only those trajectories that have at least one point between min_altitude and max_altitude.

Like FilterByBoundingBox, no clipping will take place. If a trajectory crosses the specified interval at all then you will get the whole thing back.

input
iterable

Iterable containing Trajectory objects

min_altitude
float

Minimum altitude for acceptance

max_altitude
float

Maximum altitude for acceptance

trajectories()
Return just the trajectories that intersect the bounding box

Yields Trajectory objects that fall within the altitude region

class tracktable.filter.trajectory.**FilterByBoundingBox**
Bases: object (http://docs.python.org/library/functions.html#object)

FilterByBoundingBox: Eliminate trajectories that don't intersect a given box

Given a source that produces Trajectories, return only those trajectories that intersect the specified bounding box. No clipping or subsetting is performed: if at least one point is within the desired region you will get the entire trajectory back.

input
iterable

Sequence of Trajectory objects

bbox_min
TrajectoryPoint

Southwest corner of bounding box

bbox_max
TrajectoryPoint

Northeast corner of bounding box

trajectories()
Return just those trajectories that intersect the bounding box

Yields Trajectory objects with at least one point inside the bounding box

Module contents tracktable.filters - Generators that transform things

This module contains Filters. A Filter is an object that takes in points or trajectories and produces points or trajectories. They all accept generators as their sources and only traverse their inputs once.

tracktable.info module

Submodules

tracktable.info.airports module

class tracktable.info.airports.**Airport**
Bases: object (http://docs.python.org/library/functions.html#object)

Information about a single airport

iata_code
string

3-letter IATA airport identifier

icao_code
string

4-leter ICAO airport identifier

name
string

Human-readable airport name

city
string

City where airport is located

country
string

Country where airport is located

position
tuple

(longitude, latitude, altitude) position of airport

size_rank
integer

Approximate rank among all the world's airports

utc_offset
integer

Local time zone as an offset from UTC

__str__()
Return human-readable representation of airport object

tracktable.info.airports.**airport_information**(*airport_code*)
Look up information about an airport

Parameters **airport_code** – ICAO or IATA code for an airport

Returns Airport object containing requested information.

Raises KeyError – no such airport

tracktable.info.airports.**airport_size_rank**(*airport_code*)
Return an airport's global rank by size

Parameters **airport_code** (*string* (http://docs.python.org/library/string.html#module-string)) – IATA or ICAO airport identifier

Returns Integer ranking. 1 is the largest, higher values are smaller.

tracktable.info.airports.**airport_tier**(*airport_code*)
Return an estimated tier for an airport

We divide airports roughly into 4 tiers (chosen purely by hand) for a classification task. This function lets us retrieve the tier assigned to any given airport.

Parameters **airport_code** (*string* (http://docs.python.org/library/string.html#module-string)) – IATA/ICAO airport identifier

Returns tier1, tier2, tier3 or tier4

Return type String

Raises KeyError – no such airport

tracktable.info.airports.**all_airports**()
Return all the airport records we have

Returns Unsorted list of airport objects.

tracktable.info.airports.**build_airport_dict**()
Assemble the airport dictionary on first access

This function is called whenever the user tries to look up an airport. It checks to make sure the table has been populated and, if not, loads it from disk.

Returns None

Side Effects: Airport data will be loaded if not already in memory

tracktable.info.cities module cities.py - Locations and population values for many cities of the world

class tracktable.info.cities.**CityInfo**
Bases: object (http://docs.python.org/library/functions.html#object)

Brief information about a city.

country_code
string

2-character abbreviation for country

name
string

City name

population
integer

Estimated population

latitude
float

Latitude of city location

longitude
float

Longitude of city location

tracktable.info.cities.**cities_in_bbox**(*bbox_min=(-180, -90)*, *bbox_max=(180, 90)*, *minimum_population=0*)

Return all the cities in a bounding box.

Kwargs:

bbox_min (TrajectoryPoint): Southwest corner of bounding box, default (-180, -90)

bbox_max (TrajectoryPoint): Northeast corner of bounding box, default (180, 90)

minimum_population (integer): Cities with lower population than this will not be returned. Default 0.

Returns List of CityInfo objects.

tracktable.info.cities.**largest_cities_in_bbox**(*bbox_min=(-180, -90)*, *bbox_max=(180, 90)*, *count=10*)

Return the largest N cities in a bounding box.

A city's size is measured by its population.

Parameters

- **bbox_min** (TrajectoryPoint) – Southwest corner of bounding box. Defaults to (-180, -90).
- **bbox_max** (TrajectoryPoint) – Northeast corner of bounding box. Defaults to (180, 90).
- **count** (*integer*) – How many cities to return. Defaults to 10.

Returns A list of CityInfo objects.

tracktable.info.timezones module

tracktable.info.timezones.**find_containing_timezone**(*longitude*, *latitude*)

tracktable.info.timezones.**load_timezone_shapefile**()

tracktable.info.timezones.**local_time_for_position**(*position*)

tracktable.info.timezones.**print_file**()

tracktable.info.timezones.**retrieve_file**()

Module contents tracktable.info: Various sources of information on cities and airports that we can use to annotate maps.

XXX TODO: Cite each source and its license here.

tracktable.render

Submodules

tracktable.render.clock module Draw a digital clock on a Matplotlib figure

tracktable.render.clock.**digital_clock**(*time*, *formatter*, *position*, ***kwargs*)

Add a digital clock (a string representation of time) to the image at a user-specified location

XXX: This function may go away since it doesn't provide any useful enhancement over just adding a text actor ourselves.

tracktable.render.clock.**draw_analog_clock_on_map**(*time*, *offset=None*, *center=(0.5, 0.5)*, *radius=0.05*, *axes=None*, *use_short_side_for_radius=True*, *long_hand_radius=0.9*, *short_hand_radius=0.5*, *tickmarks=True*, *tickmark_length=0.25*, *color='white'*, *linewidth=1*, *zorder=4*, *label=None*, *label_placement='top'*, *label_kwargs={}*)

Draw an analog clock on a Matplotlib figure.

NOTE: THIS INTERFACE IS SUBJECT TO CHANGE. LOTS OF CHANGE.

Parameters

- **time** (*datetime.datetime* (http://docs.python.org/library/datetime.html#datetime.datetime)) – Time to display
- **offset** (*datetime.timedelta* (http://docs.python.org/library/datetime.html#datetime.timedelta)) – Offset for local time zone. This will be added to the 'time' argument to determine the time that will actually be displayed.
- **center** (*2-tuple in [0,1] * [0, 1]*) – image-space center of clock
- **radius** (*float in [0, 1]*) – Radius of clock face
- **axes** (*matplotlib.axes.Axes* (http://matplotlib.sourceforge.net/api/axes_api.html#matplotlib.axes.Axes)) – axes to render into
- **use_short_side_for_radius** (*boolean*) – Clock radius will be calculated relative to short side of figure

- **long_hand_radius** (*float in [0, 1]*) – Radius of minute hand relative to radius of clock face
- **short_hand_radius** (*float in [0,1]*) – Radius of hour hand relative to radius of clock face
- **tickmarks** (*boolean*) – Whether or not to place tickmarks at every hour around the clock face
- **tickmark_length** (*float in [0,1]*) – Length of tickmarks relative to clock face radius
- **color** (*colorspec*) – Color for clock face and hands
- **linewidth** (*float* (http://docs.python.org/library/functions.html#float)) – Thickness (in points) of lines drawn
- **zorder** (*integer*) – Ordering of clock in image element stack (higher values are on top)
- **label** (*string* (http://docs.python.org/library/string.html#module-string)) – Label text to display near clock
- **label_placement** (*string* (http://docs.python.org/library/string.html#module-string)) – One of 'top', 'bottom'. Determines where label will be rendered relative to clock face
- **label_kwargs** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Arguments to be passed to Matplotlib text renderer for label

Side Effects:

The clock actors will be added to the 'axes' argument or (if no axes are supplied) to the current Matplotlib axes.

Returns A list of Matplotlib artists added to the figure.

tracktable.render.colormaps module Several usable, mostly-accessible color maps for use when rendering points and trajectories

These colormaps are registered with matplotlib so that you can refer to them by name just as you can the standard ones.

tracktable.render.colormaps.**make_simple_colormap**(*color_list*, *name=None*)

Create a colormap by spacing colors evenly from 0 to 1

Take a list of 2 or more colors and space them evenly from 0 to 1. The result will be instantiated as a Matplotlib color map and returned. If you include a name, the color map will also be registered so that you can refer to it with that name instead of needing the object itself.

Note that this function does not handle transparency at all. If you need that capability you will need to work directly with matplotlib.colors.LinearSegmentedColormap.

Colors are supplied to this function as 'color specifications'. Here's the documentation of those, borrowed from the documentation for matplotlib.colors.ColorConverter:

•a letter from the set 'rgbcmykw'

•a hex color string, like '#0000FF'

•a standard name, like 'aqua'

•a float, like '0.4', indicating gray on a 0-1 scale

Parameters

- **color_list** (*list* (http://docs.python.org/library/functions.html#list)) – List of color specifications (described above)
- **name** (*string* (http://docs.python.org/library/string.html#module-string)) – Optional name for the color map for registration

Returns A new matplotlib colormap object.

Side Effects: If you specify a value for the 'name' parameter then the created colormap will be registered under that name.

tracktable.render.histogram2d module

tracktable.render.mapmaker module

tracktable.render.maps module

tracktable.render.paths module tracktable.render.paths - Functions to render trajectories as curves on a map

If you're trying to figure out how to render a bunch of trajectories, take a look at draw_traffic and at tracktable/examples/render_trajectories_from_csv.py. Those will get you pointed in the right direction.

tracktable.render.paths.**draw_traffic**(*traffic_map*, *trajectory_iterable*, *color_map='BrBG'*, *color_scale=<matplotlib.colors.Normalize object>*, *trajectory_scalar_generator=None*, *trajectory_linewidth_generator=None*, *linewidth=0.1*, *dot_size=2*, *dot_color='white'*, *label_objects=False*, *label_generator=None*, *label_kwargs={}*, *axes=None*, *zorder=8*, *log_function=<built-in method write of file object>*)

Draw a set of (possibly decorated trajectories.

Parameters

- **traffic_map** – Map projection (Basemap or Cartesian)
- **color_map** – String (colormap name) or Matplotlib colormap object
- **color_scale** – Linear or logarithmic scale (matplotlib.colors.Normalize() or LogNorm())
- **trajectory_scalar_generator** – Function to generate scalars for a trajectory (default None)
- **trajectory_linewidth_generator** – Function to generate path widths for a trajectory (default None)
- **linewidth** – Constant linewidth if no generator is supplied (default 0.1, measured in points)
- **dot_size** – Radius (in points, default 2) of spot that will be drawn at the head of each trajectory
- **dot_color** – Color of spot that will be drawn at the head of each trajectory
- **label_objects** – Boolean (default False) deciding whether to draw object_id at head of each trajectory
- **label_generator** – Function to generate label for a trajectory (default None)

- **label_kwargs** – Dictionary of arguments to be passed to labeler (FIXME)
- **axes** – Matplotlib axes object into which trajectories will be rendered
- **zorder** – Layer into which trajectories will be drawn (default 8).
- **log_function** – Function to print, save or discard informational messages

Returns List of Matplotlib artists

Parameters in more detail:

`traffic_map`: Basemap instance (no default)

Basemap instance of the space in which trajectories will be rendered. We don't actually render into this object. Instead we use it to project points from longitude/latitude space down into the map's local coordinate system. Take a look at tracktable.render.maps for several ways to create this map including a few convenience functions for common regions.

`trajectory_iterable`: iterable(Trajectory) (no default)

Sequence of Trajectory objects. We will traverse this exactly once. It can be any Python iterable.

`color_map`: Matplotlib color map or string (default 'BrBG')

Either a Matplotlib color map object or a string denoting the name of a registered color map. See the Matplotlib documentation for the names of the built-ins and the tracktable.render.colormaps module for several more examples and a way to create your own.

`color_scale`: Matplotlib color normalizer (default Normalize())

Object that maps your scalar range onto a colormap. This will usually be either matplotlib.colors.Normalize() or matplotlib.colors.LogNorm() for linear and logarithmic mapping respectively.

`trajectory_scalar_generator`: function(Trajectory) -> list(float)

You can color each line segment in a trajectory with your choice of scalars. This argument must be a function that computes those scalars for a trajectory or else None if you don't care. The scalar function should take a Trajectory as its input and return a list of len(trajectory) - 1 scalars, one for each line segment to be drawn.

`trajectory_linewidth_generator`: function(Trajectory) -> list(float)

Just as you can generate a scalar (and thus a color) for each line segment, you can also generate a width for that segment. If you supply a value for this argument then it should take a Trajectory as its input and return a list of len(trajectory)-1 scalars specifying the width for each segment. This value is measured in points. If you need a single linewidth all the way through use the 'linewidth' argument.

`linewidth`: float (default 0.1)

This is the stroke width measured in points that will be used to draw the line segments in each trajectory. If you need different per-segment widths then use trajectory_linewidth_generator.

`dot_size`: float (default 2)

If this value is non-zero then a dot will be drawn at the head of each trajectory. It will be dot_size points in radius and will be colored with whatever scalar is present at the head of the trajectory.

TODO: Add a dot_size_generator argument to allow programmatic control of this argument.

`label_objects`: boolean (default False)

You can optionally label the head of each trajectory. To do so you must supply 'label_objects=True' and also a function for the label_generator argument.

`label_generator`: function(TrajectoryPoint) -> string

Construct a label for the specified trajectory. The result must be a string. This argument is ignored if label_objects is False.

TODO: Test whether Unicode strings will work here.

`label_text_kwargs`: dict (default empty)

We ultimately render labels using matplotlib.axes.Text(). If you want to pass in arguments to change the font, font size, angle or other parameters, specify them here.

`axes`: matplotlib.axes.Axes (default pyplot.gca())

This is the axis frame that will hold the Matplotlib artists that will render the trajectories. By default it will be whatever pyplot thinks the current axis set is.

`zorder`: int (default 8)

Height level where the trajectories will be drawn. If you want them to be on top of the map and anything else you draw on it then make this value large. It has no range limit.

`log_function`: function(string) (default sys.stderr.write)

The draw_paths function generates occasional status messages. It will pass them to this function. You can use this to display them to the user, write them to a file or ignore them entirely.

`tracktable.render.paths.`**`dump_trajectory`**(*trajectory*, *out=<open file '<stdout>', mode 'w'>*)

`tracktable.render.paths.`**`points_to_segments`**(*point_list*, *maximum_distance=None*) → segment_list

Given a list of N points, create a list of the N-1 line segments that connect them. Each segment is a list containing two points. If a value is supplied for maximum_distance, any segment longer than that distance will be ignored.

In English: We discard outliers.

`tracktable.render.paths.`**`remove_duplicate_points`**(*trajectory*)

Create a new trajectory with no adjacent duplicate points

Duplicate positions in a trajectory lead to degenerate line segments. This, in turn, gives some of the back-end renderers fits. The cleanest thing to do is to use the input trajectory to compute a new one s.t. no segments are degenerate.

There's still one problem case: if the entire trajectory is a single position, you will get back a trajectory where the only two points (beginning and end) are at the same position.

Parameters **`trajectory`** (tracktable.core.Trajectory) – trajectory to de-duplicate

Returns New trajectory with some of the points from the input

Module contents Tracktable Trajectory Library - Render module

This module contains code to render points or trajectories either in regular Cartesian space or on a (world) map. The 'maps' module provides a friendly way to get a map of various parts of the world.

tracktable.script_helpers package

Submodules

tracktable.script_helpers.argument_groups package These are the predefined argument groups that come with Tracktable.

tracktable.script_helpers.argument_groups.dt_point_loader module

tracktable.script_helpers.argument_groups.mapmaker module

tracktable.script_helpers.argument_groups.movie_rendering module

tracktable.script_helpers.argument_groups.parallel module

tracktable.script_helpers.argument_groups.trajectory_assembly module

tracktable.script_helpers.argument_groups.trajectory_rendering module

tracktable.script_helpers.argument_groups.utilities module

Module contents

Module contents Tracktable Trajectory Toolkit - Helper functions for creating scripts Customized version of argparse with Response Files

Regular argparse has some support for response files but they're pretty bare. There's nowhere to put comments, for example. This subclass enhances that. It adds a few capabilities:

- Argument Groups - Use a group of thematically-related arguments all at once instead of having to insert them one by one
- Response Files - Response files can have comments in them. They will be automatically parsed from the command line.
- –write-response-file - Like –help, this will write an example response file and then exit.

class `tracktable.script_helpers.argparse.`**`ArgumentParser`**(*add_response_file=True*, *fromfile_prefix_chars='@'*, *prefix_chars='-'*, *comment_character='#'*, *conflict_handler='resolve'*, ***kwargs*)

Enhanced version of the standard Python ArgumentParser.

`add_response_file`
boolean

Add the '–write-response-file' option.

`comment_character`
string

This character indicates that a line in a response file should be ignored.

ORIGINAL ARGPARSE DOCUMENTATION:

Object for parsing command line strings into Python objects.

Keyword Arguments

- **- prog – The name of the program (default** – sys.argv[0])

- **- usage – A usage message (default** – auto-generated from arguments)
- **- description – A description of what the program does**
- **- epilog – Text following the argument descriptions**
- **- parents – Parsers whose arguments should be copied into this one**
- **- formatter_class – HelpFormatter class for printing help messages**
- **- prefix_chars – Characters that prefix optional arguments**
- **- fromfile_prefix_chars – Characters that prefix files containing** –

  additional arguments
- **- argument_default – The default value for all arguments**
- **- conflict_handler – String indicating how to handle conflicts**
- **- add_help – Add a -h/-help option**

write_response_file(*out=<open file '<stdout>', mode 'w'>*)

Write an example response file

Write a response file with every line commented out to the specified file-like object. It will be populated with every argument (positional and optional) that has been configured for this parser including descent into any argument groups. The '–help' and '–write-response-file' arguments will be omitted.

Parameters **out** – File-like object for output

Returns None

tracktable.source package

Submodules

tracktable.source.combine module

tracktable.source.combine.**interleave_points_by_timestamp**(**point_sources*)

From a series of point sources, generate a new sequence sorted by timestamp.

Given one or more point sources that are themselves sorted by timestamp, generate a new sequence containing all of the points from all sources, again sorted by increasing timestamp.

Note that this function reads all the points into memory in order to build a priority queue. If you're feeling ambitious, feel free to write a new version that keeps only a single point in memory from each source at any time.

Parameters ***point_sources** (*iterables*) – One or more iterables of points

Yields TrajectoryPoint instances sorted by increasing timestamp

tracktable.source.path_point_source module trailmix.source.path_point_source - Generate points along a path

class tracktable.source.path_point_source.**TrajectoryPointSource**

Bases: object (http://docs.python.org/library/functions.html#object)

Generate points interpolated between start and finish.

start_point
TrajectoryPoint

Location for first point

end_point
TrajectoryPoint

Location for last point

num_points
integer

Number of points in the path (at least 2)

points()
Return an iterable containing the generated points in the trajectory.

Longitude, latitude, altitude (if present) and time will be interpolated evenly from start_point to end_point and start_time to end_time. Each point will have the object ID specified in self.object_id.

Returns An iterable of TrajectoryPoint instances

Raises `ValueError` – impossible / illegal values specified for one or more parameters

tracktable.source.trajectory module tracktable.source.trajectory - Sources that turn a sequence of points into a sequence of trajectories

class tracktable.source.trajectory.**AssembleTrajectoryFromPoints**
Bases: `object` (http://docs.python.org/library/functions.html#object)

Turn a sequence of points into a set of trajectories

We begin with an input sequence of TrajectoryPoints sorted by increasing timestamp. As we iterate over that sequence, we separate points by their object IDs and build up a new trajectory for each object ID. When we see a gap of duration 'separation_time' or distance 'separation_distance' between the previous and latest point for a given object ID, we package up the points so far, emit a new trajectory and use the latest point to start a new one.

input
iterable

Sequence of TrajectoryPoint objects sorted by timestamp

separation_time
datetime.timedelta

Maximum permissible time difference between adjacent points in a trajectory

separation_distance
float

Maximum permissible geographic distance (in km) between adjacent points in a trajectory

minimum_length
integer

Complete trajectories with fewer than this many points will be discarded

Example

p_source = SomePointSource() (configure point source here)

t_source = AssembleTrajectoryFromPoints() t_source.input = p_source.points() t_source.separation_time = datetime.timedelta(minutes=30) t_source.separation_distance = 100 t_source.minimum_length = 10

for trajectory in t_source.trajectories(): (do whatever you want)

trajectories()
Return trajectories assembled from input points.

Once you have supplied a point source in the 'input' attribute (which can be any iterable but is commonly the output of a PointSource) you can call trajectories() to get an iterable of trajectories. All the computation happens on demand so the execution time between getting one trajectory and getting the next one is unpredictable.

There are only loose guarantees on the order in which trajectories become available. Given trajectories A and B, if timestamp(A.end) < timestamp(B.end) then A will come up before B.

The input sequence of trajectories will only be traversed once.

Yields Trajectories built from input points

Module contents tracktable.source - Point/trajectory sources for Tracktable trajectories

This module contains Sources. A Source is an object that produces points or trajectories. These can come from anywhere else, whether loaded from a file, extracted from a database or created algorithmically.

Module contents

Tracktable Trajectory Library - Top-level module

Welcome to Tracktable!

UNITS:

All user-facing functions will measure angles in degrees and distances in kilometers.

tracktable.core module

Module contents

TrailMix Trajectory Library - Core

Basic types (point, trajectory) live in this module. They are in turn imported from the small C extension libraries.

class tracktable.core.**TrajectoryPoint**(*longitude=0*, *latitude=0*, *altitude=0*)
Bases: object (http://docs.python.org/library/functions.html#object)

This is the core geographic point class for Tracktable. NOTE: The 'Bases' is not accurate but will do for now.

This is the core geographic point class for Tracktable.

TrajectoryPoint specifies a point on the surface of a sphere that is annotated with an object ID, coordinates, altitude, heading and speed. Any or all of these may be left uninitialized depending on the user's actions although a point without coordinates is not especially useful.

The altitude value defaults to zero. Leave it this way when representing objects such as automobiles, trains or surface ships for which altitude doesn't matter.

Note: TrajectoryPoint is implemented in C++ and exposed to Python via the Boost.Python module. I will include a link to the C++ class definition as soon as I figure out how to do so within Sphinx, Breathe and Napoleon.

---

longitude
float

Longitude of point (between -180 and 180)

latitude
float

Latitude of point (between -90 and 90)

object_id
string

Object ID of entity referred to by this point

timestamp
datetime

Timestamp corresponding to this point

speed
float

Current speed of object in kilometers per hour

heading
float

Current heading of object in degrees. 0 degrees is due north and angles increase clockwise (90 degrees is due east).

altitude
float

Current altitude of object in feet.

TODO: Convert to meters.

properties
dict

User-defined properties. Names are strings, values are numbers, timestamps and strings.

`TrajectoryPoint.`**has_property**(*property_name*)
Check to see whether an entry is present in the property map

Parameters **property_name** (*string* (http://docs.python.org/library/string.html#module-string)) – Name of property to look for

Returns True or false depending on whether property was found

class `tracktable.core.`**Trajectory**
Bases: `object` (http://docs.python.org/library/functions.html#object)

Ordered sequence of timestamped points.

This class is the heart of most of what Tracktable does. It implements an ordered sequence of TrajectoryPoint objects, each of which has an ID, coordinates and a timestamp. Those compose a trajectory. All points in a single Trajectory should have the same *object_id*.

Note: The Trajectory class is implemented in C++ and exposed to Python via the Boost.Python module. We use the vector indexing suite from Boost to make a trajectory act like a list of points.

---

object_id
string

ID of object described in this trajectory. This will be the string "(empty)" if the trajectory does not contain any points.

start_time
datetime

Timestamp of the first point in the trajectory. This will be invalid if the trajectory contains no points.

end_time
datetime

Timestamp of the last point in the trajectory. This will be invalid if the trajectory contains no points.

Trajectory.**subset_in_window**(*start_time*, *end_time*)
Compute the subset of a trajectory between two timestamps.

We often want to clip a trajectory to just the part that fits within a certain window of time. This method interpolates new start and end positions that fall exactly at *start_time* and *end_time* and includes between them all of the points in the trajectory that occur within the specified interval. If the trajectory does not intersect the interval at all then the return value will be an empty Trajectory.

Parameters

- **start_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Beginning of time window
- **end_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – End of time window

Returns New Trajectory with endpoints at *start_time* and *end_time*

Trajectory.**point_at_time**(*when*)
Find the position on a trajectory at a specified time.

If the specified time falls between points on the trajectory then we will interpolate as needed. If you ask for a point before the trajectory begins or after the trajectory ends you will get the first or last point of the trajectory respectively.

Parameters **when** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Time at which to evaluate trajectory

Returns TrajectoryPoint at specified time

Trajectory.**recompute_speed**()
Compute speeds at each point of a trajectory.

Sometimes the speed measurements that come with a data set are noisy or outright wrong. Sometimes they're missing. This method is for those situations. When you call it, it will use the position and timestamp data to compute a speed at each point in the trajectory.

Trajectory.**recompute_heading**()
Compute headings at each point of a trajectory.

The heading at each point of a trajectory is the direction to the next point. The heading at the very last point is the same as the heading at the penultimate point.

`Trajectory.`**`intersects_box`**(*southwest_corner*, *northeast_corner*)
Test to see whether a trajectory intersects a rectangle in longitude/latitude space.

A trajectory intersects a box if any of its coordinates falls within the box.

Parameters

- **`southwest_corner`** (*TrajectoryPoint or (longitude, latitude) tuple*) – Southwest corner of box of interest
- **`northwest_corner`** (*TrajectoryPoint or (longitude, latitude) tuple*) – Northwest corner of box of interest

Returns True or False

static `Trajectory.`**`from_position_list`**(*points*)
Assemble a Trajectory from a list of TrajectoryPoints.

Parameters **`points`** (*list* (http://docs.python.org/library/functions.html#list)) – List of TrajectoryPoint objects

Returns New Trajectory instance containing copies of all supplied points

class `tracktable.core.`**`Timestamp`**
Convenience methods for working with timestamps.

Recall that a Timestamp is just a timezone-aware `datetime.datetime` (http://docs.python.org/library/datetime.html#datetime.datetime) object. That gives us all of the standard Python date/time functions to use for manipulating them. As a result we only need convenience functions to take care of the awkward parts: turning strings into timestamps, timestamps into strings, and imbuing a naive `datetime` object with a time zone.

static `Timestamp.`**`beginning_of_time`**()
Return a timestamp equal to January 1, 1400. No valid timestamp should ever be this old.

Returns Timestamp object set to the first day of 1400

static `Timestamp.`**`from_struct_time`**(*mytime*)
Construct a datetime from a time.struct_time object.

Parameters **`mytime`** (`time.struct_time` (http://docs.python.org/library/time.html#time.struct_time)) – Source time

Returns An aware datetime object imbued with `tracktable.core.timestamp.DEFAULT_TIMEZONE`

static `Timestamp.`**`from_dict`**(*mydict*)
Construct a datetime from a dict with named elements.

Parameters **`mydict`** (`dict` (http://docs.python.org/library/stdtypes.html#dict)) – Dict with zero or more of 'hour', 'minute', 'second', 'year', 'month', 'day', and 'utc_offset' entries. Missing entries will be set to their minimum legal values.

Returns An aware datetime object imbued with `tracktable.core.DEFAULT_TIMEZONE` unless a 'utc_offset' value is specified, in which case the specified time zone will be used instead.

static `Timestamp.`**`from_datetime`**(*mytime*)
Convert a `datetime` (http://docs.python.org/library/datetime.html#datetime.datetime) into an aware timestamp.

Parameters **mytime** (datetime.datetime (http://docs.python.org/library/datetime.html#datetime.datetime)) – datetime object to convert

Returns datetime that will definitely have a time zone attached. If the input already has a time zone then it will be returned without modification. If not, a new timestamp will be created with the supplied date/time and the default time zone.

static Timestamp.**from_any**(*thing*)

Try to construct a timestamp from whatever we're given.

The possible inputs can be:

- a Python datetime (in which case we just return a copy of the input)
- a string in the format 2013-04-05 11:23:45, in which case we will assume that it resides in timestamp.DEFAULT_TIMEZONE
- a string in the format 2013-04-05 11:23:45-05, in which case we will assume that it's UTC-5 (or other time zone, accordingly)
- a string in the format 2013-04-05T11:23:45 or 2013-04-05T11:23:45-05 – just like above but with a T in the middle instead of a space
- a string in the format 20130405112345 - these are assumed to reside in the default time zone
- a string in the format MM-DD-YYYY HH:MM:SS
- a string such as 08-Aug-2013 12:34:45 where 'Aug' is the abbreviated name for a month in your local environment
- a dict containing at least year, month, day entries and optionally hour, minute, second and utc_offset

Internally this method dispatches to all of the other Timestamp.from_xxx() methods.

Parameters **thing** – Thing to try to convert to a timestamp

Returns Timezone-aware datetime (http://docs.python.org/library/datetime.html#datetime.datetime) object

static Timestamp.**to_string**(*dt*, *format_string='%Y-%m-%d %H:%M:%S'*, *include_tz=False*)

Create a string from a timestamp.

Format contents as a string, by default formatted as 2013-04-21 14:45:00. You may supply an argument format_string if you want it in a different form. See the documentation for datetime.strftime() for information on what this format string looks like.

Parameters

- **dt** (datetime.datetime (http://docs.python.org/library/datetime.html#datetime.datetime)) – Timestamp object to stringify
- **format_string** (*string* (http://docs.python.org/library/string.html#module-string)) – String to pass to datetime.strftime()
- **include_tz** (*Boolean*) – Whether or not to append timezone as UTC offset

Returns String representation of timestamp

tracktable.core.geomath module

This file contains useful math functions. Some of them will be related to geography such as 'find the distance between these two points on the globe'.

tracktable.core.geomath.**almost_equal**(*a*, *b*, *relative_tolerance=1e-06*)
Check two numbers for equality within a tolerance

Parameters

- **a** (*float* (http://docs.python.org/library/functions.html#float)) – First number
- **b** (*float* (http://docs.python.org/library/functions.html#float)) – Second number
- **relative_tolerance** (*float* (http://docs.python.org/library/functions.html#float)) – Numbers must be close to within this fraction of their average to be considered equal

Returns True/false depending on whether or not they're equal-ish

tracktable.core.geomath.**altitude**(*thing*)
Return the altitude from a point or tuple

It is often convenient to specify a point as a (lon, lat, altitude) tuple instead of a full-fledged TrajectoryPoint. By using this function to look up altitude we can cope gracefully with both.

Parameters **point** – TrajectoryPoint or (lon, lat, altitude) tuple

Returns Altitude as float

Raises AttributeError – if attempt at access fails

tracktable.core.geomath.**bearing**(*origin*, *destination*)
Compute angular bearing between two points

Source: http://gagravarr.livejournal.com/109998.html with modifications.

Parameters

- **origin** (*BasePoint or TrajectoryPoint*) – start point
- **destination** (*BasePoint or TrajectoryPoint*) – end point

Returns Bearing from origin to destination.

Domain Information:

Terrestrial: Returned in degrees. 0 is due north, 90 is due east.

Cartesian2D: Returned in radians. 0 is positive X, pi/2 is positive Y.

Cartesian3D: Not defined.

tracktable.core.geomath.**compute_bounding_box**(*point_sequence*)
Compute a bounding box for a sequence of points.

This function will construct a domain-specific bounding box over an arbitrary sequence of points. Those points must all have the same type.

Parameters **point_sequence** – Iterable of points

Returns Bounding box with min_corner, max_corner attributes

Domain: Each domain returns a separate bounding box type.

tracktable.core.geomath.**distance**(*hither*, *yon*)
Return the distance between two points

This function will compute the distance between two points in domain-specific units.

The points being measured must be from the same domain.

Parameters

- **hither** (*BasePoint*) – point 1
- **yon** (*BasePoint*) – point 2

Returns Distance between hither and yon

Domain: Terrestrial domain returns distance in km. Cartesian domains return distance in native units.

tracktable.core.geomath.**end_to_end_distance**(*trajectory*)
Return the distance between a path's endpoints

This is just the distance between start and end points rather than the total distance traveled.

Parameters **trajectory** (Trajectory) – Path whose length we want

Returns Length in domain-dependent units

Domain: Terrestrial: distance in km Cartesian2D: distance in units Cartesian3D: distance in units

tracktable.core.geomath.**interpolate**(*start*, *end*, *t*)
Interpolate between two points

This function will interpolate linearly between two points. It is aware of the underlying coordinate system: interpolation on the globe will be done along great circles and interpolation in Cartesian space will be done along a straight line.

The points being measured must be from the same domain.

Parameters

- **start** (*BasePoint or TrajectoryPoint*) – point 1
- **end** (*BasePoint or TrajectoryPoint*) – point 2
- **t** (*float in [0, 1]*) – interpolant

Returns New point interpolated between start and end

tracktable.core.geomath.**intersects**(*thing1*, *thing2*)
Check to see whether two geometries intersect

The geometries in question must be from the same domain. They can be points, trajectories, linestrings or bounding boxes.

Parameters

- **thing1** – Geometry object
- **thing2** – Geometry object

Returns True or False

tracktable.core.geomath.**latitude**(*thing*)
Return the latitude from a point or tuple

It is often convenient to specify a point as a (lon, lat) tuple instead of a fullfledged TrajectoryPoint. By using this function to look up latitude we can cope gracefully with both.

Parameters **point** – TrajectoryPoint or (lon, lat) tuple

Returns Latitude as float

Raises AttributeError – if attempt at access fails

tracktable.core.geomath.**latitude_or_y**(*thing*)
Return the latitude or Y-coordinate from a point or tuple

Parameters **point** – TrajectoryPoint or (lon, lat) tuple or (x, y) point

Returns Latitude/Y as float

Raises AttributeError – if attempt at access fails

tracktable.core.geomath.**length**(*trajectory*)
Return the length of a path in domain-dependent units

This is the total length of all segments in the trajectory.

Parameters **trajectory** (Trajectory) – Path whose length we want

Returns Length in domain-dependent units

Domain: Terrestrial: distance in km Cartesian2D: distance in units Cartesian3D: distance in units

tracktable.core.geomath.**longitude**(*thing*)
Return the longitude from a point or tuple

It is often convenient to specify a point as a (lon, lat) tuple instead of a fullfledged TrajectoryPoint. By using this function to look up longitude we can cope gracefully with both.

Parameters **point** – TrajectoryPoint or (lon, lat) tuple

Returns Longitude as float

Raises AttributeError – if attempt at access fails

tracktable.core.geomath.**longitude_or_x**(*thing*)
Return the longitude or X-coordinate from a point or tuple

Parameters **point** – TrajectoryPoint or (lon, lat) tuple or (x, y) point

Returns Longitude/X as float

Raises AttributeError – if attempt at access fails

tracktable.core.geomath.**point_at_time**(*trajectory*, *when*)
Return a point from a trajectory at a specific time

This function will estimate a point at a trajectory at some specific time. If the supplied timestamp does not fall exactly on a vertex of the trajectory we will interpolate between the nearest two points.

Times before the beginning or after the end of the trajectory will return the start and end points, respectively.

Parameters

- **trajectory** (Trajectory) – Path to sample
- **when** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp for which we want the point

Returns TrajectoryPoint at specified time

tracktable.core.geomath.**recompute_speed**(*trajectory*, *target_attribute_name='speed'*)
Use points and timestamps to compute speed

The speed data in trajectories is often suspect. This method goes through and recomputes it based on the distance between neighboring points and the time elapsed between those points.

The speed at point N is computed using the distance and time since point N-1. The speed at point 0 is copied from point 1.

Parameters

- **trajectory** – Any Tracktable trajectory

- **target_attribute_name** – Speed will be stored in this property at each point. Defaults to 'speed'.

The trajectory will be modified in place instead of returning a new copy.

tracktable.core.geomath.**sanity_check_distance_less_than**(*max_distance*)

tracktable.core.geomath.**signed_turn_angle**(*a*, *b*, *c*)
Return signed turn angle between (a, b) and (b, c).

The magnitude of the angle tells you how far you turned. The sign of the angle tells you whether you turned right or left. Which one depends on the domain.

Parameters

- **a** (*BasePoint*) – first point
- **b** (*BasePoint*) – second point
- **c** (*BasePoint*) – third point

Returns Signed angle in domain-dependent units

Domain: Terrestrial: angle in degrees, positive angles are clockwise Cartesian2D: angle in radians, positive angles are counterclockwise Cartesian3D: not defined

tracktable.core.geomath.**speed_between**(*point1*, *point2*)
Return speed in km/hr between two timestamped points.

Parameters

- **point1** (TrajectoryPoint) – Start point
- **point2** (TrajectoryPoint) – End point

Returns Speed (as a float) measured in domain-specific units

Domain Info: Terrestrial: Speed is in km/hr Cartesian2D: Speed is in units/sec Cartesian3D: Speed is in units/sec

tracktable.core.geomath.**subset_during_interval**(*trajectory*, *start_time*, *end_time*)
Return a subset of a trajectory between two times

This function will extract some (possibly empty) subset of a trajectory between two timestamps.

If the time interval is entirely outside the trajectory, the result will be an empty trajectory. Otherwise we will use point_at_time to find the two endpoints and build a new trajectory from the endpoints and all trajectory points between them.

Parameters

- **trajectory** (Trajectory) – Path to sample
- **start_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp for beginning of subset
- **end_time** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp for end of subset

Returns Trajectory for desired interval

tracktable.core.geomath.**unsigned_turn_angle**(*a*, *b*, *c*)
Return unsigned turn angle between (a, b) and (b, c).

The magnitude of the angle tells you how far you turned. This function will not tell you whether you turned right or left - for that you need signed_turn_angle.

Parameters

- **a** (*BasePoint*) – first point
- **b** (*BasePoint*) – second point
- **c** (*BasePoint*) – third point

Returns Angle in domain-dependent units

Domain: Terrestrial: angle in degrees between 0 and 180 Cartesian2D: angle in radians between 0 and pi Cartesian3D: angle in radians between 0 and pi

tracktable.core.geomath.**xcoord**(*thing*)

Return what we think is the X-coordinate for an object.

If the supplied thing has a property named 'x' then we return that. Otherwise we try to return its first element.

Parameters **point** – Object with an 'x' property or tuple of numbers

Returns Number corresponding to x coordinate

Raises **AttributeError if attempt at access fails** –

tracktable.core.geomath.**ycoord**(*thing*)

Return what we think is the Y-coordinate for an object.

If the supplied thing has a property named 'y' then we return that. Otherwise we try to return its second element.

Parameters **point** – Object with an 'y' property or tuple of numbers

Returns Number corresponding to y coordinate

Raises **AttributeError if attempt at access fails** –

tracktable.core.simple_timezone module

simple_timezone - Timezone struct based on UTC offset

class tracktable.core.simple_timezone.**SimpleTimeZone**(*hours=0*, *minutes=0*, *name='UTC'*)

Bases: datetime.tzinfo (http://docs.python.org/library/datetime.html#datetime.tzinfo)

Trivial time zone struct for use with datetime.

We use this when we don't need the full power of the pytz package.

offset

datetime.timedelta

Offset from UTC

name

string

Human-readable name for timezone

__repr__()

Return machine-parseable representation of time zone

dst(*dt*)
Return daylight savings time offset

Since we don't support daylight savings time in SimpleTimeZone this is always zero.

Parameters **dt** (*datetime.datetime* (http://docs.python.org/library/datetime.html#datetime.datetime)) – Timestamp to check for DST

Returns Zero time offset

localize(*timestamp*)
Convert a datetime into an 'aware' datetime object by replacing its time zone

Parameters **timestamp** (*datetime.datetime* (http://docs.python.org/library/datetime.html#datetime.datetime)) – Timestamp object to make aware

Returns New datetime.datetime object with this timezone attached

utcoffset(*dt*)
Return offset from UTC

This method is required by the Python datetime library in order to create 'aware' datetime objects.

Parameters **dt** (*datetime.timedelta* (http://docs.python.org/library/datetime.html#datetime.timedelta)) – Not used

tracktable.core.timestamp module

Utility classes for position update data

class tracktable.core.timestamp.**Timestamp**
Bases: object (http://docs.python.org/library/functions.html#object)

Convenience class that can convert from different formats to an 'aware' datetime

BEGINNING_OF_TIME = datetime.datetime(1400, 1, 1, 0, 0)

static beginning_of_time()
Return a timestamp guaranteed to be before any legal data point

Returns Timestamp equal to January 1, 1400.

static from_any(*thing*)
Try to construct a timestamp from whatever we're given.

The possible inputs can be:

- a Python datetime (in which case we just return a copy of the input)
- a string in the format '2013-04-05 11:23:45', in which case we will assume that it resides in timestamp.DEFAULT_TIMEZONE
- a string in the format '2013-04-05 11:23:45-05', in which case we will assume that it's UTC-5 (or other time zone, accordingly)
- a string in the format '2013-04-05T11:23:45' or '2013-04-05T11:23:45-05' – just like above but with a T in the middle instead of a space
- a string in the format '20130405112345' - these are assumed to reside in timestamp.DEFAULT_TIMEZONE
- a string in the format 'MM-DD-YYYY HH:MM:SS'
- a string such as '08-Aug-2013 12:34:45' where 'Aug' is the abbreviated name for a month in your local environment

- a dict containing at least 'year', 'month', 'day' entries and optionally 'hour', 'minute' and 'second' - these will always represent UTC times until I implement it otherwise

Parameters **thing** – String, datetime, or dict (see above)

Returns Timezone-aware datetime object

static from_datetime(*mytime*)
Convert a datetime to an aware timestamp

Parameters **mytime** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Possibly-naive timestamp

Returns New datetime that will definitely have a time zone attached

static from_dict(*mydict*)
Construct a datetime from a dict with named elements.

Parameters **mydict** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Dict with zero or more of 'hour', 'minute', 'second', 'year', 'month', 'day', and 'utc_offset' entries. Missing entries will be set to their minimum legal values.

Returns An aware datetime object imbued with tracktable.core.DEFAULT_TIMEZONE unless a 'utc_offset' value is specified, in which case the specified time zone will be used instead.

static from_string(*timestring*, *format_string='%Y-%m-%d %H:%M:%S'*)
Convert from a string to a datetime

Populate from a string such as '2012-09-10 12:34:56' or '2012-09-10T12:34:56'. Note that you *must* have both a date and a time in that format or else the method will fail.

Also note that this method expects its input times to be in UTC.

Parameters **timestring** (*string* (http://docs.python.org/library/string.html#module-string)) – String containing your timestamp

Kwargs: format_string (string): Format string for datetime.strptime

Returns An aware datetime object imbued with tracktable.core.timestamp.DEFAULT_TIMEZONE.

static from_struct_time(*mytime*)
Construct a datetime from a time.struct_time object.

Parameters **mytime** (*time.struct_time* (http://docs.python.org/library/time.html#time.struct_time)) – Source time

Returns An aware datetime object imbued with tracktable.core.timestamp.DEFAULT_TIMEZONE.

static sanity_check(*timestamp*)
Check to see whether a timestamp might be real

We assume that any timestamp after the year 1600 has a non-zero chance of being real and that anything before that is bogus.

Parameters **timestamp** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp to check

Returns Timestamp argument if sane, None if not

static **to_iso_string**(*dt*, *include_tz=True*)
Convert a timestamp to a string in format YYYY-MM-DDTHH:MM:SS

Parameters

- **dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timezone-aware datetime object
- **include_tz** (*boolean*) – Whether or not to append a '+XXXX' timezone offset

Returns String representation of the timestamp

static **to_string**(*dt*, *format_string='%Y-%m-%d %H:%M:%S'*, *include_tz=True*)
Convert a datetime to a string

Format contents as a string, by default formatted as '2013-04-21 14:45:00'. You may supply an argument 'format_string' if you want it in a different form. See the documentation for datetime.strftime() for information on what this format string looks like.

Parameters

- **dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp object to stringify
- **format_string** (*string* (http://docs.python.org/library/string.html#module-string)) – String to pass to datetime.strftime() that describes format
- **include_tz** – Whether or not to append timezone UTC offset

Returns String version of timestamp

static **truncate_to_day**(*orig_dt*)
Zero out the time portion of a timestamp

Parameters **orig_dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Input datetime

Returns New timestamp with hours=0, minutes=0 and seconds=0

static **truncate_to_hour**(*orig_dt*)
Zero out the minutes and seconds in a timestamp

Parameters **orig_dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Input datetime

Returns New timestamp with minutes=0 and seconds=0

static **truncate_to_minute**(*orig_dt*)
Zero out the seconds in a timestamp

Parameters **orig_dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Input datetime

Returns New timestamp with seconds=0

static **truncate_to_year**(*orig_dt*)
Zero out all but the year in a timestamp

Parameters **orig_dt** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Input datetime

Returns New timestamp with month=1, day=1, hours=0, minutes=0 and seconds=0

tracktable.core.timestamp.**localize_timestamp**(*naive_ts*, *utc_offset=0*)
Imbue a naive timestamp with a timezone

Python has two kinds of timestamps: naive (just a datetime, no time zone) and aware (time plus time zone). Mixing the two is awkward. Thie routine will assign a time zone to a datetime (UTC by default) for consistency throughout Tracktable.

Note: You can change the default timezone by setting the module-level variable DEFAULT_TIMEZONE. This is not recommended.

Parameters

- **naive_ts** (*datetime* (http://docs.python.org/library/datetime.html#module-datetime)) – Timestamp to localize
- **utc_offset** (*integer*) – Number of hours offset from UTC. You can also specify a fraction of an hour as '0530', meaning 5 hours 30 minutes.

Returns A new datetime imbued with the desired time zone

tracktable.feature module

Submodules

tracktable.feature.annotations module

tracktable.features.annotations:

Annotate points or trajectories (or the points in a trajectory) with useful derived quantities

tracktable.feature.annotations.**available_annotations**()

tracktable.feature.annotations.**climb_rate**(*trajectory*, *max_climb=2000*)
climb_rate: Annotate points in an AirTrajectory with climb rate

usage: climb_rate(t: AirTrajectory) -> None

This will add a property 'climb_rate' to each point in the input trajectory. This is measured in units/sec and is computed as (points[n].altitude - points[n-1].altitude) / (points[n].timestamp - points[n-1].timestamp).

tracktable.feature.annotations.**compute_speed_from_positions**(*trajectory*)

tracktable.feature.annotations.**get_airspeed**(*trajectory*, *min_speed=0*, *max_speed=980*)
Return a vector of scalars for point-to-point speeds

This is a feature accessor that can be used to color a trajectory. It will map the 'speed' property into a range from 0 to 1.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Kwargs: min_speed (float): Minimum speed in kilometers per hour. This will be mapped to the bottom of the scalar range and thus the bottom of the color map. Defaults to 0. max_speed (float): Maximum speed in kilometers per hour. This will be mapped to the top of the scalar range and thus the top of the color map. Defaults to 980 (0.8 Mach, a common maximum permitted speed for civilian airliners).

Returns A vector of scalars that can be used as input to a colormap.

tracktable.feature.annotations.**get_climb_rate**(*trajectory*, *max_velocity=2000*)

tracktable.feature.annotations.**get_progress**(*trajectory*)

tracktable.feature.annotations.**get_speed**(*trajectory*)
Get the speed for a trajectory without any scaling.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Returns Numpy array containing the speed value for each point

tracktable.feature.annotations.**get_speed_over_water**(*trajectory*, *min_speed=0*, *max_speed=60*)

Return a vector of scalars for point-to-point speeds over water

This is a feature accessor that can be used to color a trajectory. It will map the 'speed' property into a range from 0 to 1.

Parameters **trajectory** (tracktable.core.Trajectory) – Trajectory containing speeds

Kwargs: min_speed (float): Minimum speed in kilometers per hour. This will be mapped to the bottom of the scalar range and thus the bottom of the color map. Defaults to 0. max_speed (float): Maximum speed in kilometers per hour. This will be mapped to the top of the scalar range and thus the top of the color map. Defaults to 60 km/h (32 knots, very fast for big ships but slower than the maximum speed of high-speed civilian ferries).

Returns A vector of scalars that can be used as input to a colormap.

tracktable.feature.annotations.**progress**(*trajectory*)

progress: Annotate points in an AirTrajectory with flight progress

usage: progress(t: AirTrajectory) -> None

This will add a property "progress" to each point in the input trajectory. This property will be 0 at the first point, 1 at the last point, and spaced evenly in between.

tracktable.feature.annotations.**register_annotation**(*feature_name*, *compute_feature*, *retrieve_feature*)

tracktable.feature.annotations.**retrieve_feature_accessor**(*name*)

tracktable.feature.annotations.**retrieve_feature_function**(*name*)

Module contents

TrailMix Trajectory Library - Features module

All the code for dealing with features (derived quantities) on points and trajectories lives in this module.

tracktable.filter module

Submodules

tracktable.filter.trajectory module

Module contents tracktable.filter.trajectory - Filters that take trajectories as input

class tracktable.filter.trajectory.**ClipToTimeWindow**

Bases: object (http://docs.python.org/library/functions.html#object)

Truncate trajectories to fit within a time window

Given an iterable of Trajectory objects, return those portions of each trajectory that fit within the specified time window. Interpolate endpoints as necessary.

input
iterable

Source of Trajectory objects

start_time
datetime

Beginning time for window of interest

end_time
datetime

End time for window of interest

trajectories()
Return sub-trajectories within window.

Note: Since this is a generator, you can only traverse the sequence once unless you collect it in a list yourself.

Yields Trajectories derived from input trajectories. Each trajectory is guaranteed to fall entirely within the window specified by self.start_time and self.end_time. If one of the input trajectories extends beyond that boundary, a new endpoint will be interpolated so that it begins or ends precisely at the boundary. Trajectories entirely outside the boundary will be returned as empty trajectories with 0 points.

class tracktable.filter.trajectory.**FilterByAltitude**
Bases: object (http://docs.python.org/library/functions.html#object)

Filter out trajectories that don't intersect an interval of altitude

Given a source that produces Trajectories, return only those trajectories that have at least one point between min_altitude and max_altitude.

Like FilterByBoundingBox, no clipping will take place. If a trajectory crosses the specified interval at all then you will get the whole thing back.

input
iterable

Iterable containing Trajectory objects

min_altitude
float

Minimum altitude for acceptance

max_altitude
float

Maximum altitude for acceptance

trajectories()
Return just the trajectories that intersect the bounding box

Yields Trajectory objects that fall within the altitude region

class tracktable.filter.trajectory.**FilterByBoundingBox**
Bases: object (http://docs.python.org/library/functions.html#object)

FilterByBoundingBox: Eliminate trajectories that don't intersect a given box

Given a source that produces Trajectories, return only those trajectories that intersect the specified bounding box. No clipping or subsetting is performed: if at least one point is within the desired region you will get the entire trajectory back.

input
iterable

Sequence of Trajectory objects

bbox_min
TrajectoryPoint

Southwest corner of bounding box

bbox_max
TrajectoryPoint

Northeast corner of bounding box

trajectories()
Return just those trajectories that intersect the bounding box

Yields Trajectory objects with at least one point inside the bounding box

Module contents

tracktable.filters - Generators that transform things

This module contains Filters. A Filter is an object that takes in points or trajectories and produces points or trajectories. They all accept generators as their sources and only traverse their inputs once.

tracktable.info module

Submodules

tracktable.info.airports module

class tracktable.info.airports.**Airport**
Bases: object (http://docs.python.org/library/functions.html#object)

Information about a single airport

iata_code
string

3-letter IATA airport identifier

icao_code
string

4-leter ICAO airport identifier

name
string

Human-readable airport name

city
string

City where airport is located

country
string

Country where airport is located

position
tuple

(longitude, latitude, altitude) position of airport

size_rank
integer

Approximate rank among all the world's airports

utc_offset
integer

Local time zone as an offset from UTC

__str__()
Return human-readable representation of airport object

tracktable.info.airports.**airport_information**(*airport_code*)
Look up information about an airport

Parameters **airport_code** – ICAO or IATA code for an airport

Returns Airport object containing requested information.

Raises KeyError – no such airport

tracktable.info.airports.**airport_size_rank**(*airport_code*)
Return an airport's global rank by size

Parameters **airport_code** (*string* (http://docs.python.org/library/string.html#module-string)) – IATA or ICAO airport identifier

Returns Integer ranking. 1 is the largest, higher values are smaller.

tracktable.info.airports.**airport_tier**(*airport_code*)
Return an estimated tier for an airport

We divide airports roughly into 4 tiers (chosen purely by hand) for a classification task. This function lets us retrieve the tier assigned to any given airport.

Parameters **airport_code** (*string* (http://docs.python.org/library/string.html#module-string)) – IATA/ICAO airport identifier

Returns tier1, tier2, tier3 or tier4

Return type String

Raises KeyError – no such airport

tracktable.info.airports.**all_airports**()
Return all the airport records we have

Returns Unsorted list of airport objects.

tracktable.info.airports.**build_airport_dict**()
Assemble the airport dictionary on first access

This function is called whenever the user tries to look up an airport. It checks to make sure the table has been populated and, if not, loads it from disk.

Returns None

Side Effects: Airport data will be loaded if not already in memory

tracktable.info.cities module

cities.py - Locations and population values for many cities of the world

class tracktable.info.cities.**CityInfo**
Bases: object (http://docs.python.org/library/functions.html#object)

Brief information about a city.

country_code
string

2-character abbreviation for country

name
string

City name

population
integer

Estimated population

latitude
float

Latitude of city location

longitude
float

Longitude of city location

tracktable.info.cities.**cities_in_bbox**(*bbox_min=(-180, -90)*, *bbox_max=(180, 90)*, *minimum_population=0*)

Return all the cities in a bounding box.

Kwargs:

bbox_min (TrajectoryPoint): Southwest corner of bounding box, default (-180, -90)

bbox_max (TrajectoryPoint): Northeast corner of bounding box, default (180, 90)

minimum_population (integer): Cities with lower population than this will not be returned. Default 0.

Returns List of CityInfo objects.

tracktable.info.cities.**largest_cities_in_bbox**(*bbox_min=(-180, -90)*, *bbox_max=(180, 90)*, *count=10*)

Return the largest N cities in a bounding box.

A city's size is measured by its population.

Parameters

- **bbox_min** (TrajectoryPoint) – Southwest corner of bounding box. Defaults to (-180, -90).
- **bbox_max** (TrajectoryPoint) – Northeast corner of bounding box. Defaults to (180, 90).
- **count** (*integer*) – How many cities to return. Defaults to 10.

Returns A list of CityInfo objects.

tracktable.info.timezones module

tracktable.info.timezones.**find_containing_timezone**(*longitude*, *latitude*)

tracktable.info.timezones.**load_timezone_shapefile**()

tracktable.info.timezones.**local_time_for_position**(*position*)

tracktable.info.timezones.**print_file**()

tracktable.info.timezones.**retrieve_file**()

Module contents

tracktable.info: Various sources of information on cities and airports that we can use to annotate maps.

XXX TODO: Cite each source and its license here.

tracktable.render

Submodules

tracktable.render.clock module Draw a digital clock on a Matplotlib figure

tracktable.render.clock.**digital_clock**(*time*, *formatter*, *position*, ***kwargs*)

Add a digital clock (a string representation of time) to the image at a user-specified location

XXX: This function may go away since it doesn't provide any useful enhancement over just adding a text actor ourselves.

tracktable.render.clock.**draw_analog_clock_on_map**(*time*, *offset=None*, *center=(0.5, 0.5)*, *radius=0.05*, *axes=None*, *use_short_side_for_radius=True*, *long_hand_radius=0.9*, *short_hand_radius=0.5*, *tickmarks=True*, *tickmark_length=0.25*, *color='white'*, *linewidth=1*, *zorder=4*, *label=None*, *label_placement='top'*, *label_kwargs={}*)

Draw an analog clock on a Matplotlib figure.

NOTE: THIS INTERFACE IS SUBJECT TO CHANGE. LOTS OF CHANGE.

Parameters

- **time** (*datetime.datetime* (http://docs.python.org/library/datetime.html#datetime.datetime)) – Time to display
- **offset** (*datetime.timedelta* (http://docs.python.org/library/datetime.html#datetime.timedelta)) – Offset for local time zone. This will be added to the 'time' argument to determine the time that will actually be displayed.
- **center** (*2-tuple in [0,1] * [0, 1]*) – image-space center of clock
- **radius** (*float in [0, 1]*) – Radius of clock face

- **axes** (*matplotlib.axes.Axes* (http://matplotlib.sourceforge.net/api/axes_api.html#matplotlib.axes.Axes)) – axes to render into
- **use_short_side_for_radius** (*boolean*) – Clock radius will be calculated relative to short side of figure
- **long_hand_radius** (*float in [0, 1]*) – Radius of minute hand relative to radius of clock face
- **short_hand_radius** (*float in [0,1]*) – Radius of hour hand relative to radius of clock face
- **tickmarks** (*boolean*) – Whether or not to place tickmarks at every hour around the clock face
- **tickmark_length** (*float in [0,1]*) – Length of tickmarks relative to clock face radius
- **color** (*colorspec*) – Color for clock face and hands
- **linewidth** (*float* (http://docs.python.org/library/functions.html#float)) – Thickness (in points) of lines drawn
- **zorder** (*integer*) – Ordering of clock in image element stack (higher values are on top)
- **label** (*string* (http://docs.python.org/library/string.html#module-string)) – Label text to display near clock
- **label_placement** (*string* (http://docs.python.org/library/string.html#module-string)) – One of 'top', 'bottom'. Determines where label will be rendered relative to clock face
- **label_kwargs** (*dict* (http://docs.python.org/library/stdtypes.html#dict)) – Arguments to be passed to Matplotlib text renderer for label

Side Effects:

The clock actors will be added to the 'axes' argument or (if no axes are supplied) to the current Matplotlib axes.

Returns A list of Matplotlib artists added to the figure.

tracktable.render.colormaps module Several usable, mostly-accessible color maps for use when rendering points and trajectories

These colormaps are registered with matplotlib so that you can refer to them by name just as you can the standard ones.

tracktable.render.colormaps.**make_simple_colormap**(*color_list*, *name=None*)

Create a colormap by spacing colors evenly from 0 to 1

Take a list of 2 or more colors and space them evenly from 0 to 1. The result will be instantiated as a Matplotlib color map and returned. If you include a name, the color map will also be registered so that you can refer to it with that name instead of needing the object itself.

Note that this function does not handle transparency at all. If you need that capability you will need to work directly with matplotlib.colors.LinearSegmentedColormap.

Colors are supplied to this function as 'color specifications'. Here's the documentation of those, borrowed from the documentation for matplotlib.colors.ColorConverter:

•a letter from the set 'rgbcmykw'

•a hex color string, like '#0000FF'

•a standard name, like 'aqua'

•a float, like '0.4', indicating gray on a 0-1 scale

Parameters

- **color_list** (*list* (http://docs.python.org/library/functions.html#list)) – List of color specifications (described above)
- **name** (*string* (http://docs.python.org/library/string.html#module-string)) – Optional name for the color map for registration

Returns A new matplotlib colormap object.

Side Effects: If you specify a value for the 'name' parameter then the created colormap will be registered under that name.

tracktable.render.histogram2d module

tracktable.render.mapmaker module

tracktable.render.maps module

tracktable.render.paths module tracktable.render.paths - Functions to render trajectories as curves on a map

If you're trying to figure out how to render a bunch of trajectories, take a look at draw_traffic and at tracktable/examples/render_trajectories_from_csv.py. Those will get you pointed in the right direction.

tracktable.render.paths.**draw_traffic**(*traffic_map*, *trajectory_iterable*, *color_map='BrBG'*, *color_scale=<matplotlib.colors.Normalize object>*, *trajectory_scalar_generator=None*, *trajectory_linewidth_generator=None*, *linewidth=0.1*, *dot_size=2*, *dot_color='white'*, *label_objects=False*, *label_generator=None*, *label_kwargs={}*, *axes=None*, *zorder=8*, *log_function=<built-in method write of file object>*)

Draw a set of (possibly decorated trajectories.

Parameters

- **traffic_map** – Map projection (Basemap or Cartesian)
- **color_map** – String (colormap name) or Matplotlib colormap object
- **color_scale** – Linear or logarithmic scale (matplotlib.colors.Normalize() or LogNorm())
- **trajectory_scalar_generator** – Function to generate scalars for a trajectory (default None)
- **trajectory_linewidth_generator** – Function to generate path widths for a trajectory (default None)
- **linewidth** – Constant linewidth if no generator is supplied (default 0.1, measured in points)
- **dot_size** – Radius (in points, default 2) of spot that will be drawn at the head of each trajectory

- **dot_color** – Color of spot that will be drawn at the head of each trajectory
- **label_objects** – Boolean (default False) deciding whether to draw object_id at head of each trajectory
- **label_generator** – Function to generate label for a trajectory (default None)
- **label_kwargs** – Dictionary of arguments to be passed to labeler (FIXME)
- **axes** – Matplotlib axes object into which trajectories will be rendered
- **zorder** – Layer into which trajectories will be drawn (default 8).
- **log_function** – Function to print, save or discard informational messages

Returns List of Matplotlib artists

Parameters in more detail:

`traffic_map`: Basemap instance (no default)

Basemap instance of the space in which trajectories will be rendered. We don't actually render into this object. Instead we use it to project points from longitude/latitude space down into the map's local coordinate system. Take a look at tracktable.render.maps for several ways to create this map including a few convenience functions for common regions.

`trajectory_iterable`: iterable(Trajectory) (no default)

Sequence of Trajectory objects. We will traverse this exactly once. It can be any Python iterable.

`color_map`: Matplotlib color map or string (default 'BrBG')

Either a Matplotlib color map object or a string denoting the name of a registered color map. See the Matplotlib documentation for the names of the built-ins and the tracktable.render.colormaps module for several more examples and a way to create your own.

`color_scale`: Matplotlib color normalizer (default Normalize())

Object that maps your scalar range onto a colormap. This will usually be either matplotlib.colors.Normalize() or matplotlib.colors.LogNorm() for linear and logarithmic mapping respectively.

`trajectory_scalar_generator`: function(Trajectory) -> list(float)

You can color each line segment in a trajectory with your choice of scalars. This argument must be a function that computes those scalars for a trajectory or else None if you don't care. The scalar function should take a Trajectory as its input and return a list of len(trajectory) - 1 scalars, one for each line segment to be drawn.

`trajectory_linewidth_generator`: function(Trajectory) -> list(float)

Just as you can generate a scalar (and thus a color) for each line segment, you can also generate a width for that segment. If you supply a value for this argument then it should take a Trajectory as its input and return a list of len(trajectory)-1 scalars specifying the width for each segment. This value is measured in points. If you need a single linewidth all the way through use the 'linewidth' argument.

`linewidth`: float (default 0.1)

This is the stroke width measured in points that will be used to draw the line segments in each trajectory. If you need different per-segment widths then use trajectory_linewidth_generator.

`dot_size`: float (default 2)

If this value is non-zero then a dot will be drawn at the head of each trajectory. It will be dot_size points in radius and will be colored with whatever scalar is present at the head of the trajectory.

TODO: Add a dot_size_generator argument to allow programmatic control of this argument.

`label_objects`: boolean (default False)

You can optionally label the head of each trajectory. To do so you must supply 'label_objects=True' and also a function for the label_generator argument.

`label_generator`: function(TrajectoryPoint) -> string

Construct a label for the specified trajectory. The result must be a string. This argument is ignored if label_objects is False.

TODO: Test whether Unicode strings will work here.

`label_text_kwargs`: dict (default empty)

We ultimately render labels using matplotlib.axes.Text(). If you want to pass in arguments to change the font, font size, angle or other parameters, specify them here.

`axes`: matplotlib.axes.Axes (default pyplot.gca())

This is the axis frame that will hold the Matplotlib artists that will render the trajectories. By default it will be whatever pyplot thinks the current axis set is.

`zorder`: int (default 8)

Height level where the trajectories will be drawn. If you want them to be on top of the map and anything else you draw on it then make this value large. It has no range limit.

`log_function`: function(string) (default sys.stderr.write)

The draw_paths function generates occasional status messages. It will pass them to this function. You can use this to display them to the user, write them to a file or ignore them entirely.

`tracktable.render.paths.`**`dump_trajectory`**(*trajectory*, *out=<open file '<stdout>', mode 'w'>*)

`tracktable.render.paths.`**`points_to_segments`**(*point_list*, *maximum_distance=None*) → segment_list

Given a list of N points, create a list of the N-1 line segments that connect them. Each segment is a list containing two points. If a value is supplied for maximum_distance, any segment longer than that distance will be ignored.

In English: We discard outliers.

`tracktable.render.paths.`**`remove_duplicate_points`**(*trajectory*)

Create a new trajectory with no adjacent duplicate points

Duplicate positions in a trajectory lead to degenerate line segments. This, in turn, gives some of the back-end renderers fits. The cleanest thing to do is to use the input trajectory to compute a new one s.t. no segments are degenerate.

There's still one problem case: if the entire trajectory is a single position, you will get back a trajectory where the only two points (beginning and end) are at the same position.

Parameters **`trajectory`** (tracktable.core.Trajectory) – trajectory to de-duplicate

Returns New trajectory with some of the points from the input

Module contents

Tracktable Trajectory Library - Render module

This module contains code to render points or trajectories either in regular Cartesian space or on a (world) map. The 'maps' module provides a friendly way to get a map of various parts of the world.

tracktable.source package

Submodules

tracktable.source.combine module

tracktable.source.combine.**interleave_points_by_timestamp**(**point_sources*)
From a series of point sources, generate a new sequence sorted by timestamp.

Given one or more point sources that are themselves sorted by timestamp, generate a new sequence containing all of the points from all sources, again sorted by increasing timestamp.

Note that this function reads all the points into memory in order to build a priority queue. If you're feeling ambitious, feel free to write a new version that keeps only a single point in memory from each source at any time.

Parameters ***point_sources** (*iterables*) – One or more iterables of points

Yields TrajectoryPoint instances sorted by increasing timestamp

tracktable.source.path_point_source module

trailmix.source.path_point_source - Generate points along a path

class tracktable.source.path_point_source.**TrajectoryPointSource**
Bases: object (http://docs.python.org/library/functions.html#object)

Generate points interpolated between start and finish.

start_point
TrajectoryPoint

Location for first point

end_point
TrajectoryPoint

Location for last point

num_points
integer

Number of points in the path (at least 2)

points()
Return an iterable containing the generated points in the trajectory.

Longitude, latitude, altitude (if present) and time will be interpolated evenly from start_point to end_point and start_time to end_time. Each point will have the object ID specified in self.object_id.

Returns An iterable of TrajectoryPoint instances

Raises ValueError – impossible / illegal values specified for one or more parameters

tracktable.source.trajectory module

tracktable.source.trajectory - Sources that turn a sequence of points into a sequence of trajectories

class tracktable.source.trajectory.**AssembleTrajectoryFromPoints**
Bases: object (http://docs.python.org/library/functions.html#object)

Turn a sequence of points into a set of trajectories

We begin with an input sequence of TrajectoryPoints sorted by increasing timestamp. As we iterate over that sequence, we separate points by their object IDs and build up a new trajectory for each object ID. When we see a gap of duration 'separation_time' or distance 'separation_distance' between the previous and latest point for a given object ID, we package up the points so far, emit a new trajectory and use the latest point to start a new one.

input
iterable

Sequence of TrajectoryPoint objects sorted by timestamp

separation_time
datetime.timedelta

Maximum permissible time difference between adjacent points in a trajectory

separation_distance
float

Maximum permissible geographic distance (in km) between adjacent points in a trajectory

minimum_length
integer

Complete trajectories with fewer than this many points will be discarded

Example

p_source = SomePointSource() (configure point source here)

t_source = AssembleTrajectoryFromPoints() t_source.input = p_source.points() t_source.separation_time = datetime.timedelta(minutes=30) t_source.separation_distance = 100 t_source.minimum_length = 10

for trajectory in t_source.trajectories(): (do whatever you want)

trajectories()
Return trajectories assembled from input points.

Once you have supplied a point source in the 'input' attribute (which can be any iterable but is commonly the output of a PointSource) you can call trajectories() to get an iterable of trajectories. All the computation happens on demand so the execution time between getting one trajectory and getting the next one is unpredictable.

There are only loose guarantees on the order in which trajectories become available. Given trajectories A and B, if timestamp(A.end) < timestamp(B.end) then A will come up before B.

The input sequence of trajectories will only be traversed once.

Yields Trajectories built from input points

Module contents

tracktable.source - Point/trajectory sources for Tracktable trajectories

This module contains Sources. A Source is an object that produces points or trajectories. These can come from anywhere else, whether loaded from a file, extracted from a database or created algorithmically.

tracktable.script_helpers package

Submodules

tracktable.script_helpers.argument_groups package These are the predefined argument groups that come with Tracktable.

tracktable.script_helpers.argument_groups.dt_point_loader module

tracktable.script_helpers.argument_groups.mapmaker module

tracktable.script_helpers.argument_groups.movie_rendering module

tracktable.script_helpers.argument_groups.parallel module

tracktable.script_helpers.argument_groups.trajectory_assembly module

tracktable.script_helpers.argument_groups.trajectory_rendering module

tracktable.script_helpers.argument_groups.utilities module

Module contents

Module contents

Tracktable Trajectory Toolkit - Helper functions for creating scripts Customized version of argparse with Response Files

Regular argparse has some support for response files but they're pretty bare. There's nowhere to put comments, for example. This subclass enhances that. It adds a few capabilities:

- Argument Groups - Use a group of thematically-related arguments all at once instead of having to insert them one by one
- Response Files - Response files can have comments in them. They will be automatically parsed from the command line.
- –write-response-file - Like –help, this will write an example response file and then exit.

class `tracktable.script_helpers.argparse.`**`ArgumentParser`**(*add_response_file=True*, *fromfile_prefix_chars='@'*, *prefix_chars='-'*, *comment_character='#'*, *conflict_handler='resolve'*, ***kwargs*)

Enhanced version of the standard Python ArgumentParser.

`add_response_file`
boolean

Add the '–write-response-file' option.

comment_character
string

This character indicates that a line in a response file should be ignored.

ORIGINAL ARGPARSE DOCUMENTATION:

Object for parsing command line strings into Python objects.

Keyword Arguments

- **- prog – The name of the program (default** – sys.argv[0])
- **- usage – A usage message (default** – auto-generated from arguments)
- **- description – A description of what the program does**
- **- epilog – Text following the argument descriptions**
- **- parents – Parsers whose arguments should be copied into this one**
- **- formatter_class – HelpFormatter class for printing help messages**
- **- prefix_chars – Characters that prefix optional arguments**
- **- fromfile_prefix_chars – Characters that prefix files containing** –

  additional arguments
- **- argument_default – The default value for all arguments**
- **- conflict_handler – String indicating how to handle conflicts**
- **- add_help – Add a -h/-help option**

write_response_file(*out=<open file '<stdout>', mode 'w'>*)
Write an example response file

Write a response file with every line commented out to the specified file-like object. It will be populated with every argument (positional and optional) that has been configured for this parser including descent into any argument groups. The '–help' and '–write-response-file' arguments will be omitted.

Parameters **out** – File-like object for output

Returns None

1.4.2 C++ Interface

In this release there is minimal functionality available directly in C++. We have the point hierarchy ending in `TrajectoryPoint` (a 2D point on a globe) and `PointBaseCartesian` (an N-dimensional point – bring your own dimension) as well as the Trajectory class. All of these have the necessary typedefs and traits to be used with the `boost::geometry` library. We also have `DelimitedTextPointReader` in the `TracktableIO` library.

Our first release is focused on getting enough capability out there to start rendering maps and movies.

TracktableCore module

template <std::size_t *Dimension*>
class `tracktable::PointBase`
Base class for all points in Tracktable

This class defines a point independent of the number of coordinates or the data type.

You will not use this directly. Instead, you'll use one of the coordinate-specific versions like PointBaseCartesian or PointBaseLonLat.

PointBase and all of its subclasses will be registered with boost::geometry so that you can use all of the generic geometry algorithms.

Public Functions

PointBase()
Initialize an empty point.

PointBase(*PointBase* **const** &*other*)
Initialize a copy of another point.

template <std::size_t *d*>
coordinate_type **const** &**get**() const
Get the value of a particular coordinate.

Since this is Boost, you set and get coordinates by specifying the coordinate at compile time:

```
double x = point.get<0>();
```

template <std::size_t *d*>
void **set**(coordinate_type **const** &*value*)
Set the value of a particular coordinate.

Since this is Boost, you set and get coordinates by specifying the coordinate at compile time:

point.set<0>(new_value);

coordinate_type **const** &**operator[]**(std::size_t *d*) const
Get/set the value of a coordinate.

You can use operator[] whether or not you know the coordinate you want ahead of time.

double x = point[0]; point[0] = x;

coordinate_type &**operator[]**(std::size_t *d*)
Get/set the value of a coordinate.

You can use operator[] whether or not you know the coordinate you want ahead of time.

double x = point[0]; point[0] = x;

bool **operator==**(*PointBase* **const** &*other*) const
Check two points for equality.

This requires that the two points have the same dimension.

bool **operator!=**(*PointBase* **const** &*other*) const
Check two points for inequality.

PointBase &**operator=**(*PointBase* **const** &*other*)
Make this point a copy of a different one.

Protected Attributes

coordinate_type **Coordinates**[Dimension]
Storage for the coordinate values.

Warning: doxygenclass: Cannot find class "tracktable::PointBaseCartesian" in doxygen xml output for project "project0" from directory: /Users/atwilso/projects/tracktable/devel/build/Documentation/doxyxml

Warning: doxygenclass: Cannot find class "tracktable::PointBaseLonLat" in doxygen xml output for project "project0" from directory: /Users/atwilso/projects/tracktable/devel/build/Documentation/doxyxml

template <**class** PointT>
class `tracktable::`**`Trajectory`**

Ordered sequence of points.

This class is the heart of most of what Tracktable does. It implements an ordered sequence of *TrajectoryPoint* objects, each of which has an ID, coordinates and a timestamp. Those compose a trajectory.

We provide accessors so that you can treat a *Trajectory* as if it were a std::vector.

Public Types

typedef PointT **`point_type`**

Convenient aliases for template parameters and types from internal storage.

Public Functions

`Trajectory``()`

Instantiate an empty trajectory.

`Trajectory``(`**const** *Trajectory* &*other*`)`

Create a trajectory a copy of another.

Trajectory &**`operator=`**`(`**const** *Trajectory* &*other*`)`

Make this trajectory a copy of another.

Timestamp **`start_time`**`()` const

Return the start time if available.

If there are any points in the trajectory this method will return the timestamp on the first point. If not, it will return an invalid Timestamp.

Timestamp **`end_time`**`()` const

Return the end time if available.

If there are any points in the trajectory this method will return the timestamp on the last point. If not, it will return an invalid Timestamp.

std::string **`object_id`**`()` const

Return the ID of the moving object.

If there are any points in the trajectory, return the object ID of the first one. Otherwise return the string "(empty)".

std::string **`trajectory_id`**`()` const

Return a unique ID for the trajectory.

Return a unique ID for the trajectory incorporating its object ID, start time and end time. If the trajectory is empty then we will return the string "(empty)".

void **`set_property`**`(`std::string **const** &*name*, double *value*`)`

Set a named property with a double-precision value.

void **set_property**(std::string **const** &*name*, std::string **const** &*value*)
Set a named property with a string value.

void **set_property**(std::string **const** &*name*, Timestamp **const** &*value*)
Set a named property with a timestamp value.

PropertyValueT **property**(std::string **const** &*name*, bool **ok*) const
Retrieve a named property with checking.

PropertyValueT **property_without_checking**(std::string **const** &*name*) const
Retrieve a named property without safety checking.

std::string **string_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a string value.

double **numeric_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a floating-point value.

Timestamp **timestamp_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a timestamp value.

bool **has_property**(std::string **const** &*name*) const
Check whether a property is present.

PropertyMap **const** &**__properties**() const
INTERNAL METHOD.

void **__set_properties**(PropertyMap **const** &*props*)
INTERNAL METHOD.

Trajectory(size_type *n*, *point_type initial_value*)
Create a new trajectory with pre-specified length.

Create a new trajectory with n elements. You may also supply a point that will be copied into each element.

Parameters

- `n` - Length of the trajectory
- `initial_value` - Point to be used to fill the new vector

template <class InputIterator**>**
Trajectory(InputIterator *first*, InputIterator *last*)
Create a new trajectory from a range of points.

Create a new trajectory by copying points from [first, last).

Parameters

- `first` - Iterator pointing to the first point for the new trajectory
- `last` - Iterator pointing past the last point for the new trajectory

size_type **size**() const
Return the length of the trajectory in points.

size_type **max_size**() const
Return the maximum number of entries the points array can hold.

size_type **capacity**() const
Return the current allocated capacity of the points array.

void **resize**(size_type *new_size*, *point_type default_value*)
Resize the points array to contain exactly the number of entries requested.

Parameters

- `new_size` - Desired length of the vector
- `default_value` - Value for newly allocated entries

bool **empty**() const
Return whether or not the trajectory is empty.

void **reserve**(size_type *n*)
Preallocate enough space in the array for the specified number of entries.

Parameters

- `n` - Allocate space for this many points.

bool **operator==**(**const** *Trajectory* &*other*) const
Check whether one trajectory is equal to another by comparing all the points.

Two trajectories are equal if all of their points are equal.

Parameters

- `other` - *Trajectory* for comparison

bool **operator!=**(**const** *Trajectory* &*other*) const
Check whether two trajectories are unequal.

Parameters

- `other` - *Trajectory* for comparison

point_type **const** &**operator[]**(size_type *i*) const
Return a given point from the trajectory.

Return the requested point from the trajectory. It is the caller's responsibility to ensure that a valid index has been requested.

Parameters

- `i` - Index of desired point

point_type &**operator[]**(size_type *i*)
Return a mutable reference to a given point in the trajectory.

As with the const version of operator[], it is the caller's responsibility to ensure that a valid index has been requested.

Parameters

- `i` - Index of desired point

void **add_point**(*point_type* **const** &*pt*)
Append a point to the trajectory.

Note that this uses copy semantics rather than move semantics. Since move semantics are part of C++11 we will not use them until we can be reasonable sure that suitable compilers are available in all environments that we care about.

Parameters

- `pt` - Point to append

void **push_back**(*point_type* **const** &*pt*)

Append a point to the trajectory.

Note that this uses copy semantics rather than move semantics. Since move semantics are part of C++11 we will not use them until we can be reasonable sure that suitable compilers are available in all environments that we care about.

Note

Why do we have this alias?

Parameters

- `pt` - Point to append

point_type &**at**(size_type *i*)

Retrieve the point at a given index with bounds checking.

Retrieve a point. Unlike operator[], *at()* does bounds checking and will throw an exception if you ask for a point outside the range [0, num_points).

This version of the function will be called if you try to modify the point. For example:

my_trajectory.at(3).set_latitude(15);

Return

Mutable reference to the requested point

Parameters

- `i` - Which point to retrieve

point_type **const** &**at**(size_type *i*) const

Retrieve the point at a given index with bounds checking.

Retrieve a point. Unlike operator[], *at()* does bounds checking and will throw an exception if you ask for a point outside the range [0, num_points).

This version of the function will be called if the compiler can tell that you're not trying to modify the point. For example:

TrajectoryPoint my_point = my_trajectory.at(3);

Return

Immutable reference to the requested point

Parameters

- `i` - Which point to retrieve

iterator **erase**(iterator *position*)
Remove a point from the trajectory.

Delete the point at the specified position (specified by an iterator). The points after the one deleted will be moved up one spot to fill the gap.

Note

This operation takes linear time in the number of points in the trajectory.

Parameters

- `position` - Iterator pointing at the location to erase.

iterator **erase**(iterator *first*, iterator *last*)
Remove a range of points from the trajectory.

Delete points from trajectories starting at 'first' and ending at the point just before 'last'. Points after the deleted range will be moved up to fill the gap.

Note

This operation takes linear time in the number of points in the trajectory.

Parameters

- `first` - Iterator pointing at the first point to erase
- `last` - Iterator pointing to the location after the last point to erase

void **clear**()
Reset the trajectory to an empty state.

This clears out the vector of points.

point_type &**front**()
Return the first point in the trajectory.

Return

First point in trajectory (mutable reference)

Note

TODO: Document behavior if trajectory is empty

point_type **const** &**front**() const
Return the (immutable) first point in the trajectory.

Return

First point in trajectory (immutable reference)

Note

TODO: Document behavior if trajectory is empty

point_type &**back**()
Return the last point in the trajectory.

Return

Last point in trajectory (mutable reference)

Note

TODO: Document behavior if trajectory is empty

point_type **const** &**back**() const
Return the (immutable) last point in the trajectory.

Return

Last point in trajectory (immutable reference)

Note

TODO: Document behavior if trajectory is empty

iterator **insert**(iterator *position*, *point_type* **const** &*value*)
Insert a single element into the trajectory at an arbitrary position.

Insert a point into any position in the trajectory. All points after this location will be moved farther down.

Parameters

- `position` - Location to insert the point
- `value` - Point to insert

void **insert**(iterator *position*, size_type *n*, *point_type* **const** &*value*)
Fill a range in the trajectory.

Insert n copes of the point specified as 'value' starting at the specified 'position'. All points after this location will be moved farther down in the trajectory.

Parameters

- `position` - Where to insert the points
- `n` - How many points to insert
- `value` - What point to insert

template <class InputIterator**>**
void **insert**(iterator *position*, InputIterator *first*, InputIterator *last*)
Insert a range of points into the trajectory.

Insert all points in the range [first, last) into the trajectory. All points after this location will be moved farther down in the trajectory.

Parameters

- `position` - Where to start inserting the points
- `first` - The first point to insert
- `last` - The location after the last point to insert

iterator **begin**()
Return an iterator pointing to the beginning of the trajectory.

The point underneath this iterator can be changed.

const_iterator **begin**() const
Return an iterator pointing to the beginning of the trajectory.

The point underneath this iterator cannot be changed.

iterator **end**()
Return an iterator pointing beyond the last point in the trajectory.

The point underneath this iterator, if there is one, can be changed.

const_iterator **end**() const
Return an iterator pointing beyond the last point in the trajectory.

The point underneath this iterator, if there is one, cannot be changed.

Protected Attributes

point_vector_type **Points**
Internal storage for the points in the trajectory.

template <class BasePointT>
class `tracktable::`**TrajectoryPoint**
Add object ID, timestamp, property map.

This class will add trajectory properties (a timestamp, an object ID and storage for named properties) to any point class.

Timestamp is a tracktable::Timestamp which (under the hood) is a boost::posix_time::ptime. Object ID is stored as a string.

We also include an interface to set, get and enumerate arbitrary named properties. The only restriction is that the types of these properties are limited to timestamps, floating-point numbers and strings. If you need something more flexible than that please consider creating your own alternative point class either by subclassing *TrajectoryPoint* or by composition.

Note

Named property support is implemented using boost::variant. You will need to use boost::get<> to cast it to your desired data type or else call one of the (type)_property_value functions to retrieve it with no casting necessary. Take a look at C++/Core/Tests/test_trajectory_point.cpp (XXX CHECK THIS) for a demonstration.

Public Functions

TrajectoryPoint()
Instantiate an uninitialized point.

TrajectoryPoint(*TrajectoryPoint* **const** &*other*)
Initialize a *TrajectoryPoint* as a copy of an existing point.

TrajectoryPoint(base_point_type **const** &*other*)
Initialize with coordinates from a base point.

TrajectoryPoint **operator=**(*TrajectoryPoint* **const** &*other*)
Make this *TrajectoryPoint* a copy of an existing point.

bool **operator==**(*TrajectoryPoint* **const** &*other*) const
Check two points for equality.

bool **operator!=**(*TrajectoryPoint* **const** &*other*) const
Check two points for inequality.

std::string **object_id**() const
Return this point's object ID.

Timestamp **timestamp**() const
Return this point's timestamp.

void **set_object_id**(std::string **const** &*new_id*)
Set this point's object ID.

void **set_timestamp**(Timestamp **const** &*ts*)
Set this point's timestamp.

void **set_property**(std::string **const** &*name*, double *value*)
Set a named property with a double-precision value.

void **set_property**(std::string **const** &*name*, std::string **const** &*value*)
Set a named property with a string value.

void **set_property**(std::string **const** &*name*, Timestamp **const** &*value*)
Set a named property with a timestamp value.

PropertyValueT **property**(std::string **const** &*name*, bool **ok*) const
Retrieve a named property with checking.

PropertyValueT **property**(std::string **const** &*name*, PropertyValueT **const** &*default_value*) const
Retrieve a named property or a default value.

PropertyValueT **property_without_checking**(std::string **const** &*name*) const
Retrieve a named property without safety checking.

std::string **string_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a string value.

double **numeric_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a floating-point value.

Timestamp **timestamp_property**(std::string **const** &*name*, bool **ok*) const
Safely retrieve a named property with a timestamp value.

std::string **string_property_with_default**(std::string **const** &*name*, std::string **const** &*default_value*) const
Safely retrieve a named property with a string value.

double **numeric_property_with_default**(std::string **const** &*name*, double *default_value*) const
Safely retrieve a named property with a floating-point value.

Timestamp **timestamp_property_with_default**(std::string **const** &*name*, Timestamp **const** &*default_value*) const
Safely retrieve a named property with a timestamp value.

bool **has_property**(std::string **const** &*name*) const
Check whether a property is present.

std::string **to_string**() const
Convert point to a human-readable string form.

PropertyMap &**__non_const_properties**()
INTERNAL METHOD.

void **__set_properties**(PropertyMap **const** &*props*)
INTERNAL METHOD.

Protected Attributes

std::string **ObjectId**
: Storage for a point's object ID.

Timestamp **UpdateTime**
: Storage for a point's timestamp.

PropertyMap **Properties**
: Storage for a point's named properties.

TracktableIO module

Module contents

Warning: doxygenclass: Cannot find class "tracktable::DelimitedTextPointReader" in doxygen xml output for project "project0" from directory: /Users/atwilso/projects/tracktable/devel/build/Documentation/doxyxml

TracktableAnalysis module

Module contents

template <class *PointT*>
class `tracktable::DBSCAN`
: Cluster points using the *DBSCAN* algorithm.

  DBSCAN is a non-parametric clustering algorithm that defines a point in a cluster as "a point with more than N neighbors inside a search radius R". N and R are user-specified parameters. The consequence of this definition is that areas of points with a certain minimum density form clusters regardless of their shape.

  In order to use *DBSCAN* you must supply the following parameters:

  - A list of points to cluster. These can be in any coordinate system supported by `boost::geometry` and any container that provides begin() and end() iterators.
  - A search box: the distance that a point can be in any dimension in order to count as "nearby". Note that we use a search box instead of the sphere that the original *DBSCAN* implementation requires. This helps when you have a space where different dimensions have different meaning such as longitude/latitude (sensible values are on the order of 0-200 in each dimension) and altitude (sensible values are up to 15000 meters).
  - A minimum cluster size: Any point that has at least this many points within its search box is part of a cluster.

  If you would rather use a slightly more convenient interface please refer to the functions in `Analysis/ComputeDBSCANClustering.h`.

  For more information about the *DBSCAN* algorithm please refer to the original paper: Ester, Martin; Kriegel, Hans-Peter; Sander, Jörg; Xu, Xiaowei (1996). "A density-based algorithm for discovering clusters in large spatial databases with noise". In Simoudis, Evangelos; Han, Jiawei; Fayyad, Usama M. "Proceedings of the Second International Conference on Knowledge Discovery and Data Mining (KDD-96)".

Public Functions

DBSCAN()
Initialize an empty clusterer.

template <class IteratorT>
int **learn_clusters**(IteratorT *point_begin*, IteratorT *point_end*, point_type **const** &*epsilon_box_half_span*, unsigned int *min_cluster_size*)
Learn cluster labels for a set of points.

This is the method that you will call to run *DBSCAN*.

You will need to call *cluster_membership_lists()* or *point_cluster_labels()* to get the results back.

Return

Number of clusters detected (cluster 0 is noise)

Parameters

- `point_begin` - Iterator for beginning of input points
- `point_end` - Iterator past end of input points
- `epsilon_box_half_span` - "Nearby" distance in each dimension
- `min_cluster_size` - Minimum number of points for a cluster

void **cluster_membership_lists**(int_vector_vector_type &*output*)
Return the point IDs belonging to each cluster.

This method is the first of two ways to get clustering results back from *DBSCAN*. Given a *DBSCAN* run that detected C clusters, this method will return a std::vector of C std::vector<int>. Each vector lists the points that belong to a single cluster.

Clusters 1 through C-1 are the "real" clusters. Cluster 0 is the noise cluster comprising all points that (1) did not have enough nearby neighbors to qualify as cluster points in their own right and (2) were not neighbors of any points that did.

Parameters

- `output` - Cluster membership lists.

void **point_cluster_labels**(int_vector_type &*out_labels*)
Return the cluster ID for each point.

This method is the second of two ways to get clustering results back from *DBSCAN*. Given a *DBSCAN* run on P points that detected C clusters, this method will return a std::vector<int> with P elements. Each element will have a value between 0 and C-1 inclusive.

Clusters 1 through C-1 are the "real" clusters. Cluster 0 is the noise cluster comprising all the points that (1) did not have enough nearby neighbors to qualify as cluster points in their own right and (2) were not neighbors of any points that did.

Parameters

- `cluster_labels` - Labels for each point in the input.

Protected Functions

void **compute_cluster_membership**(indexed_point_vector_type &*points*, unsigned int *min_cluster_size*, **const** point_type &*epsilon_box_half_span*, rtree_type &*rtree*)

Learn cluster assignments for all points.

This is the driver method that implements the skeleton of *DBSCAN*.

Parameters

- `points` - Points with indices attached
- `min_cluster_size` - Minimum number of points in neighborhood required to define a core point
- `epsilon_box_half_span` - Search range in each direction
- `rtree` - Boost RTree to accelerate neighborhood queries

bool **expand_cluster**(rtree_value_type &*seed_point*, unsigned int *min_cluster_size*, **const** point_type &*epsilon_box_half_span*, unsigned int *next_cluster_id*, rtree_type &*rtree*)

Discover a single cluster.

This method contains the heart of *DBSCAN*: searching the neighborhood of a single point to discover whether it is in the interior of a cluster and, if so, discovering the rest of that cluster.

Return

Boolean - whether or not a new cluster was discovered

Parameters

- `seed_point` - Point to examine for cluster-ness
- `min_cluster_size` - Minimum neighbor count for core points
- `epsilon_box_half_span` - Search range in each direction
- `next_cluster_id` - Numeric ID for new cluster
- `rtree` - Search structure for neighbor queries

void **build_cluster_membership_lists**(**const** indexed_point_vector_type &*points*, unsigned int *max_cluster_id*)

Assemble cluster membership lists from points.

The clustering algorithm stores its results in the indexed point list. This function extracts those results and builds cluster membership lists that are more useful to the user.

Parameters

- `points` - Points labeled with cluster IDs
- -

Private Functions

int **get_cluster_id**(rtree_value_type **const** &*iter*)

Internal method.

We use this when processing the results of a range query. This is just an accessor.

void **set_cluster_id**(rtree_value_type &*iter*, int *new_id*)
Internal method.

We use this when setting cluster IDs for many points at once.

Warning: doxygenfunction: Unable to resolve multiple matches for function "tracktable::cluster_with_dbscan" with arguments () in doxygen xml output for project "project0" from directory: /Users/atwilso/projects/tracktable/devel/build/Documentation/doxyxml. Potential matches:

```
- template <class ClusterSpacePointT, class PointIteratorT>
  int tracktable::cluster_with_dbscan(PointIteratorT, PointIteratorT, ClusterSpacePointT,
- template <class ClusterSpacePointT, class PointIteratorT>
  int tracktable::cluster_with_dbscan(PointIteratorT, PointIteratorT, ClusterSpacePointT,
```

Todo

Clean up table of contents. Look at tracktable.render.rst for an example.

Todo

Write an explanatory page for the Python domain module.

1.5 Conventions and Principles

1. All speeds shall be measured in **kilometers per hour**.

   There are almost as many "convenient" units for measuring speed as there are types of moving objects. Rather than expect the user to remember whether the velocity Railway Flibbertigibbet is measured in knots, Mach numbers or furlongs per fortnight, we choose to standardize on kilometers per hour. The geomath module will include functions to convert back and forth between this and other units.

2. All distances on the Earth will be measured in **kilometers**.

   There is a plethora of units of distance to go along with units of speed. On top of those, many algorithms for geographic calculations deal in angles instead of linear distance. Whenever the user must specify a distance we will ask her to do so in kilometers.

3. Positions on the Earth shall be measured in **degrees of longitude and latitude**.

   This one is simpler. The only real question is whether latitude or longitude should come first. Both approaches have their proponents. *We will always specify longitude first in tuples*. The canonical globe will have longitudes from -180 to 180 and latitudes from -90 to 90.

4. Bearings shall be measured in degrees. A bearing of 0 is due north and 90 degrees is due east.

   This has been navigational convention for a very, very long time.

5. All default **color maps shall be accessible** to people with red-green color blindness.

   At a minimum, a color map should be usable when converted to black and white. In addition, its color channels should obey the following rules of thumb: one can use either red or green but not both in the same color map.

   We will never use or encourage the use of the Dreaded Rainbow Color Map. If you are unfamiliar with this controversy we strongly encourage you to read How the Rainbow Color Map Deceives (http://eagereyes.org/basics/rainbow-color-map) and Rainbow Color Map (Still) Considered Harmful (http://people.renci.org/ borland/pdfs/RainbowColorMap_VisViewpoints.pdf).

6. External software dependencies should be minimized.

   We have all had the experience of trying to install a software package only to discover that we are missing some crucial dependency. That dependency has two others. Finally, deep down, the entire house of code requires a never-released bootleg version of some library that noone has used since the days of the PDP/11.

   Obviously we want to prevent this.

   However, there is an even more compelling reason to minimize dependencies. There are computing environments where software installation is governed by administrative and legal constraints rather than technical ones. In such an environment it can take months just to install a more recent version of an already-approved package, let alone an entirely new one. By minimizing our external dependencies and sticking to stable versions we make it easier to use Tracktable in any environment of interest.

"Those are my principles, and if you don't like them... well, I have others."

Groucho Marx

1.6 Contacts

For questions, bug reports or contributions contact:

- Andrew T. Wilson <atwilso@sandia.gov (atwilso@sandia.gov)>

1.7 Credits

1.7.1 Tracktable

Tracktable is written by Andy Wilson and Danny Rintoul at Sandia National Laboratories in Albuquerque, New Mexico.

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

1.7.2 Third-Party Software

We include databases of cities (from MaxMind (http://www.maxmind.com)), time zone boundaries (from Eric Muller) and airports (from partow.net (http://www.partow.net/miscellaneous/airportdatabase)) in Tracktable.

Their licenses are reproduced in full in the file `LICENSE.html` in the root directory of the Tracktable distribution. In this section we reproduce those parts of the licenses that are required to be present in documentation.

MaxMind City Database

This product includes data created by MaxMind, available from *http://www.maxmind.com/*.

1.8 Things To Do

Todo

Document response files here.

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/examples.rst, line 209.)

Todo

In a future release there will be a detailed explanation of the differences between the C++ and Python interfaces and how to go back and forth between them.

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/reference.rst, line 6.)

Todo

Clean up table of contents. Look at tracktable.render.rst for an example.

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/reference.rst, line 55.)

Todo

Write an explanatory page for the Python domain module.

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/reference.rst, line 58.)

Todo

Code example for annotations

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/user_guide.rst, line 414.)

Todo

We haven't described how to set up a map projection for the Cartesian domain.

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/user_guide.rst, line 544.)

Todo

Add tracktable.script_helpers.argument_groups to the documentation

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/user_guide.rst, line 841.)

Todo

Document response files in full

(The original entry is located in /Users/atwilso/projects/tracktable/devel/Documentation/user_guide.rst, line 849.)

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

t

Symbols

A

B

C

D

E

U

W

X

Y

DISTRIBUTION:

1	MS 0359	D. Chavez, LDRD Office, 1911
1	MS 0899	Technical Library, 9536 (electronic copy)


Sandia National Laboratories