Projecting Irregular Vehicle Positions on Tracks

Finn Peter Evers

Bachelor's Thesis
September 2024

Real-Time and Embedded Systems Group
Department of Computer Science
Kiel University

Advised by
Dr. Ing. Alexander Schulz-Rosengarten

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Weiterhin erkläre ich, dass die digitale Fassung dieser Arbeit, die dem Prüfungsamt per E-Mail zugegangen ist, der vorliegenden schriftlichen Fassung entspricht.

Kiel,

Abstract

Accurate positioning of vehicles on railway tracks remains a challenge and many different approaches for this issue have been developed [OBL+17]. On the other side, public transport passengers show high interest in the ability to see up-to-date transportation vehicle positions on interactive maps [CO07]. However, existing visualization solutions often depend on timetables for vehicles as well as reliable data being available for calculating and visualizing currently expected vehicle locations.

This thesis proposes a new algorithm that is able to visualize on-demand public transport and vehicles on rails without a schedule. With minimal and inaccurate Global Navigation Satellite System data only, available for vehicles on a predefined track, in order to produce a best-effort projection and near-future prediction for vehicles on the track. Through quality evaluation of received signals, utilization of all previously received vehicle positions for vehicles, and consideration of constraints for vehicles and the track, the algorithm aims to produce a visualization reflecting the most plausible positions of vehicles on the given track. It also provides an estimate on how accurate the displayed positions are. Furthermore, it uses all available data for all vehicles on the track for the purpose of producing a near-future real-time movement prediction for vehicles, considering the positions of other vehicles on the track, the structure of the given track, and previous data readings in the process. The resulting vehicle projection, its predicted inaccuracy, and a vehicle movement prediction can then be visualized on a map, with the possibility of displaying the prediction as if the vehicle position was updated in real-time. A preliminary evaluation of the developed implementation shows varying results for the accuracy of calculated projections and predictions.

Acknowledgements

Firstly, I would like to thank Prof. Dr. Reinhard von Hanxleden as the head of the Real-Time and Embedded Systems Group for the possibility to work on this topic and take a part in the REAKT DATA project.

Furthermore, I am grateful for the help of my supervisor Dr.-Ing. Alexander Schulz-Rosengarten, whose guidance helped me a lot during writing of the thesis.

Lastly, I want to thank Merlin Felix, Tokessa Hamann and Yorik Hansen for the feedback, help and good company during the entirety of writing the thesis.

Contents

List of Figures

List of Tables

List of Acronyms

API Application Programming Interface

CRS coordinate reference system

EPSG European Petroleum Survey Group Geodesy

GNSS Global Navigation Satellite System

GPS Global Positioning System

HTTP Hypertext Transport Protocol

ID identifier

OSM OpenStreetMap

WGS World Geodetic System

Chapter 1

Introduction

Many methods applicable for monitoring trains on railway networks using the Global Navigation Satellite System (GNSS) have been developed by researchers [OBL+17], especially in the context of the European Train Control System [LS14]. Still, GNSS data is never perfectly reliable on its own. Positions are rarely perfectly accurate, signal quality decreases heavily in non open-sky environments, and positional updates can even be lost during transit in such scenarios [LS14; KH17]. Integrity of GNSS signals in urban transport remains an unsolved task [ZMB+18]. Consequently, utilizing GNSS information for vehicle localization always raises challenges and is often dealt with by utilizing other information from sensors added to vehicles or railway tracks.

Meanwhile, with the ongoing digitalization of everyday life and the easy availability of information using the internet, the demand for up-to-date and accurate information is at an all-time high and predicted to rise even further[1]. This includes the wish for real-time information for public transit operations, where a study even suggest that better up-to-date information availability increases the usage of transit services [BMW15]. Other findings show a high interest of passengers to see real-time locations of transport vehicles [CO07].

1.1 Related Work

While real-time information about current train schedules and delays is generally available in various ways, access to real-time locations of vehicles remains rather limited. Few providers publish this information accurately visualized in real-time on digital maps for the public. For example, the German network provider DB InfraGO AG only provides a paid service available for registered railway partners and unavailable for public use[2]. For the British and French networks, the company Signalbox Technologies Limited provides interactive maps for displaying live train positions[3]. Yet, the service provided by Signalbox has a few limitations: It does not ensure that trains are actually displayed upon their corresponding train tracks, but rather linearly interpolates between a historic train GNSS position and a predicted future position of the vehicle. This leads to train sometimes being displayed as if they were off the tracks as seen in Figure 1.1 and sometimes even trains changing their driving direction from one second to another in case the prediction changed. Furthermore, the prediction does not

[1] https://www.statista.com/statistics/871513/worldwide-data-created

[2] https://www.dbinfrago.com/web/schienennetz/leistungen/neben-und-zusatzleistungen-und-services/unsere-nebenleistungen/db-livemaps-10909360

[3] https://map.signalbox.io and https://fr.map.signalbox.io/ respectively

Figure 1.1. Visible Inaccuracy in Projection by Signalbox Technologies Limited. Source: `https://map.signalbox.io`

take the physical constraints of railway tracks into account. For example, the constraints of single track rails, which only one train can pass at any given time, are completely ignored. That can cause visualizations where trains seemingly collide, as they are displayed driving on the same track in opposite directions. Lastly, vehicle positions are only updated every second and not continuously. Thus, while being based upon live data, the visualization lacks realism, as many of the displayed states do not reflect the reality of the state on the tracks.

Brosi et al. solved many of the problems described above in their developed approach: Utilizing scheduled timetable information and real-time updates where available, they display approximated and continuously updated positions of vehicles on their planned paths [BBS14]. Vehicles currently in motion are also displayed as such, since their positions are updated fluently on the map. However, their method falls short for any public transportation offerings on a given track without schedule and/or fully pre-planned path due to the fact that Brosi et al. intentionally decided against incorporating GNSS data into their method [BBS14]. Furthermore, while their visualization is rather continuous, vehicles start jumping on higher map zoom levels and the vehicle speed is modelled as constant without any vehicle acceleration or deceleration.

Especially the first issue renders their method inapplicable for monitoring on-demand public transport offerings or maintenance vehicles, as these do not follow any predefined schedule. With on-demand offerings expected to become more popular in the future [SC18], researching a method to visualize vehicle positions as accurate as possible seems a relevant topic.

1.2 The REAKT project

One example for a project researching on-demand public transport is the *REAKT DATA* project[4]. *REAKT DATA* is part of the *REAKT* initiative which deals with the topic of reactivating shut down railway tracks through the use of autonomous vehicles. The initiative utilizes the disused railway track between Malente and Lütjenburg as a real-live laboratory for research purposes. Currently, tourists can drive on the rails with trolleys for any distance at any speed. Whenever they are not moving, trolleys can be lifted out of the tracks in order to let other vehicles pass.

These trolleys are also used to collect data for the *REAKT DATA* project as they are each equipped with a GNSS-sensor. These sensors attempt to send GNSS updates in regular intervals containing information about the vehicle position as well as the vehicle heading and speed if available. However, the transmitted data in its raw state is unreliable: Positions are rarely perfectly accurate as they often indicate that trolleys are driving several meters off the track. Furthermore, some updates are lost during transit, resulting in varying intervals between received updates, often of several minutes in length. Thus, for visualizing

This thesis was developed primarily for the *REAKT* track and data from the *REAKT DATA* project served as the basis for testing and evaluation of the developed algorithm.

1.3 Problem Statement

For on-demand public transport, accurate predictions of vehicle position and speed are particularly interesting. With no timetable, positioning information of vehicles provide the primary source of truth for departure times.

The primary issue with GNSS-positions is the accuracy, continuity, and reliability of the data: Trackers only send their location periodically and some of the transmitted data is lost during transit, leading to even larger time gaps in the data stream. On top of that, practically no positions sent by GNSS trackers are perfectly accurate, as many received spatial coordinates suggest the vehicle is a few to more than ten meters away from the track. After all, it can be assumed that vehicles will not leave the track in normal operation, thus any position not on the tracks is inaccurate to some degree. Concluding, using GNSS data alone for monitoring vehicles provides periodic snapshots of the situation on the tracks at best. Yet, inaccurately reported coordinates result in no accurate monitoring at all.

To overcome the challenges of lossy and inaccurate data, especially in the context of the *REAKT DATA* project, the goal is to specify and implement an algorithm which estimates the vehicle solely upon the available GNSS data, requiring no more information from vehicles at all. The data is assumed to be inaccurate in position and updates are expected to be heavily varying in interval, on average multiple minutes apart. It is assumed that GNSS sensor readings can be unambiguously matched to vehicles and that the interval between updates for a given vehicle can be precisely measured. Furthermore, updates received from GNSS

[4] `https://reakt.sh`

sensors might have information regarding vehicle heading and speed, however, this is not required for the algorithm to work. With that in mind, the algorithm should aim to provide a best-effort approximation of current vehicle positions by weighting the different information available about the vehicles and the track itself.

1.4 Outline

The following chapter, Chapter 2, explains how the different described problems can be solved. Chapter 3 dives into the implementation of the proposed concepts. Afterward, Chapter 4 hosts a short evaluation of the developed implementation. At last, Chapter 5 summarizes the findings and gives an outlook for possible future work.

Chapter 2

Concepts

The primary goal for the developed algorithm is to move from individually received, partially unreliable, and in their raw form unrelated position updates for each vehicle to a plausible, connected state model for all vehicles on the track. For the modelling of the track state, all available information is first evaluated and weighted. Overall, the most recent GNSS signal and all signals of the current trip for each vehicle are considered to be the most important information, whereas the position of other vehicles on the track and statistical information collected about historical trips on each track are also considered relevant, yet only in combination with the most recent position information. Through the weighting and combination of these different factors, a best-effort approximation is computed by the algorithm and can then be visualized in real-time for end-users on their devices.

2.1 Track Acquisition

In order to detect, evaluate, and correct vehicle updates that are not on the associated tracks, geographical and possibly other information about the track is required. This information needs to be accurate, as the attempted correction for positions off tracks can only be as good as the track data utilized to rectify these positional errors.

A good and open data source to acquire information about any track is OpenStreetMap (OSM)[1]: OSM is a crowdsourced initiative, aiming to provide an open, free and editable map of the entire world [HW08]. It provides accurate geographic data with overall high worldwide coverage, especially in urban areas [NZ14]. With contributors continually reviewing and updating the available information, OSM has become a reliable and open source for global map data [ZZ10; MM+17].

Within OSM, information is primarily stored in nodes, ways, and relations. Every node describes a single point on earth and has geospatial information in form of *latitude* and *longitude* attached to it. Furthermore, every way consists of two or more ordered nodes and every relation contains one or more nodes, ways or other relations[2]. Moreover, all of these elements can have tags consisting of a key and a value for that key attached to them. Tags are the primary method to classify objects and convey metadata in OSM, with over 96 000 distinct keys currently in use[3]. For example, a node accompanied by the key-value-pair

[1] https://www.openstreetmap.org/about

[2] https://wiki.openstreetmap.org/wiki/Elements

[3] https://wiki.openstreetmap.org/wiki/Tags

railway=level_crossing indicates that at the position of that node, a road is crossing railway tracks on the same level.

Following the reviews from Ziestra and Zimpf [ZZ10] as well as Mooney et al. [MM+17], OSM is deemed a good and accurate source of reference for track information in this thesis. Thus, geographical and other information about the vehicles track are acquired from OSM by extracting and filtering the available data for the given track as needed. The final, filtered information then contains a set of all nodes in OSM for the given track connected to at least one other node in the set, as unconnected nodes do not carry any information for the extracted track, all geographic positions for these extracted nodes, details about which of the extracted nodes are connected via tracks in real life and finally the OSM key-value-pairs associated to each extracted node, as these are needed at a later step for track segmentation. The extracted information about the geographical track layout as well as its special characteristics at certain location serves as the primary source of reference throughout this thesis.

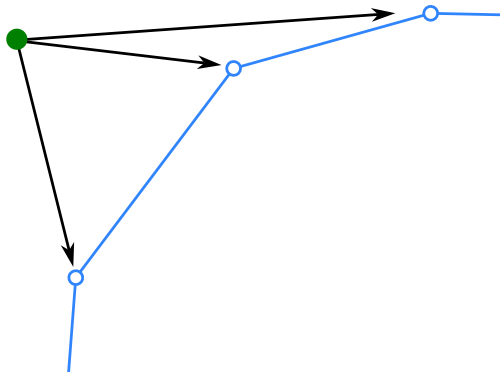
2.2 Correction of Inaccurate Position Updates

The best and primary resource for modelling the positions of vehicles on the track are the received position updates from the GNSS trackers on the vehicles, as this is the only available information directly tied to the vehicles. Yet, practically all received positions sent by the GNSS trackers will not be perfectly accurate and deviate slightly from the vehicles actual position [LS14]. In an effort to correct the deviation in the received position from the actual position, the extracted track information from OSM can be utilized: As rail vehicles will not leave the track under normal circumstances, the first effort to be made in order to correct these inaccuracies is to calculate the nearest position on the tracks for the received update. This point can then serve as a basis for further evaluation and other considerations.

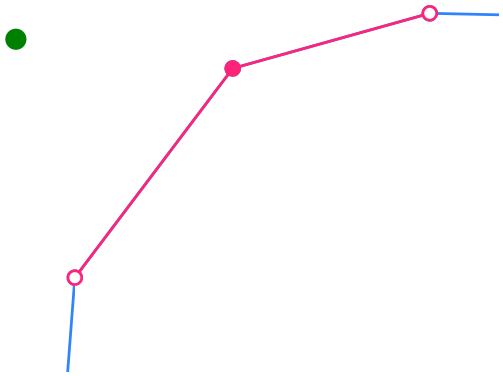
The basic procedure for calculating this point seen in Figure 2.1 can be described as follows: First, we have to find the spatially closest point to the given, inaccurate position update in all the given points from OSM. Then, for each edge connected to the closest point we have to calculate the closest location on that edge to the given update position. Lastly, we check which location and edge is the closest in all calculated points on connected edges based upon the distance between the determined location and update position.

2.2.1 Finding the Nearest Edge for the Given Track

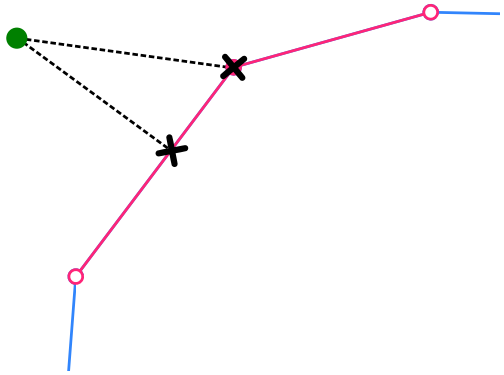
To find the closest point in all points from OSM, we first have to ensure the given update coordinates and the points from OSM follow the same coordinate reference system (CRS), as only points of the same cartographic system are comparable. OSM uses the World Geodetic System (WGS) 84 coordinate system, which is an ellipsoidal two-dimensional coordinate system and the current standard for geodesics and cartography. This is also the default CRS used by the Global Positioning System (GPS) [SD16], thus, for GPS-trackers, no coordinate transformation is required. Yet, other GNSS-sensors use different CRS [SD16], in which case a



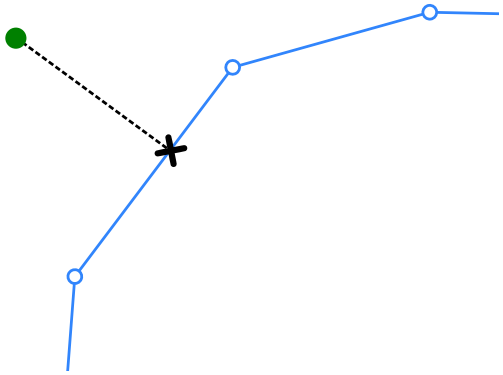
(a) Distance calculated from the update location to all points.



(b) Determined closest point with connected edges highlighted.



(c) Calculated closest locations on edges connected to the closest point.



(d) Determined closest location on the track for the given update.

Figure 2.1. Process of determining the nearest point on the track for a given update.

coordinate space transformation to the WGS 84 representation would be required.

With all points now in the same CRS, we can now search for the spatially closest point in the set of points from OSM. As WGS is an ellipsoidal system, we now have to calculate all geodesic lines between the given update point and all points from OSM. After all distances have been calculated, the shortest calculated geodesic is the most interesting one, as this yields the closest point on the track to the given location from the update.

2.2.2 Calculate Closest Point on Track for the Given Position

In order to now determine the spatially closest point on the track, we can utilize that with the closest point from OSM in the track known, the point must lie on one of the edges connected to the point. As every point after the extraction is part of at least one edge, the nearest point to the update must lie on one of these edges, since otherwise the received update point and the previously acquired point would not be the spatially closest.

To calculate the minimum distance of the point to an edge, we have to compute the dropped perpendicular foot for the received point to that edge. However, this is difficult in an ellipsoidal CRS. Thus, we have to transform the coordinates of the received update point, the node, and all to that node connected nodes into a cartesian two-dimensional CRS, e. g., the European Petroleum Survey Group Geodesy (EPSG):3857. In this coordinate system, we can then easily compute the perpendicular distance for all lines through the given edge nodes and the closest point on that line. After we have retrieved all the closest points on the lines, we have to clamp these points to the edges connected to the closest nodes, as there might be cases where the closest point on a line might not be on the given edge anymore - in that case, the closest point for that edge is the closest node, since otherwise the point would be on that edge. After we have retrieved all clamped closest points on edges between the closest node and connected nodes, we can then convert these closest points back to an ellipsoidal CRS and then find the point with the shortest geodesic to the received update position in this set of points. This then yields us the closest point for the given update on the track, the closest edge, and the corresponding nodes for that event, and the distance of that update to the track, which is the length of the shortest geodesic. Here we also perform or first evaluation of received updates: Should the update's distance to toe track exceed a predefined distance $d_{\max}$ to the track, we consider the update to be too degraded in quality based on that reading. The update for this is thus discarded at this step and neither handled further nor considered for any evaluation described in the following sections.

2.3 Improving Outlier Detection based upon Local Historic Inaccuracy

For a given inaccurate update, we now can calculate the nearest point on the track and the distance of that update to the track, which gives a general idea on how inaccurate the update possibly is. However, one issue remains: The positional inaccuracy, which is two-

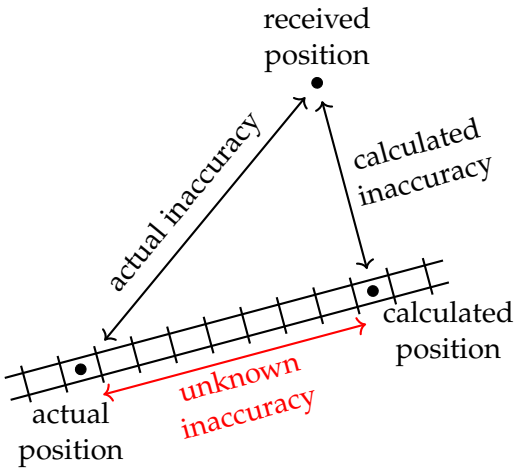


Figure 2.2. Unknown vehicle inaccuracy.

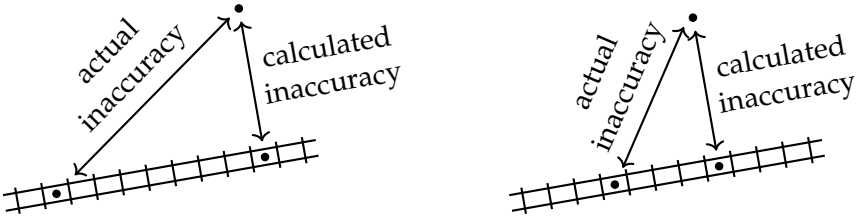


Figure 2.3. Example for two updates with equal calculated inaccuracy, yet different actual inaccuracy.

dimensional in nature, was reduced to a one-dimensional distance to the track. Each update has an inaccuracy away from the track which was just calculated and an unknown inaccuracy along the track as highlighted in Figure 2.2. Following the Pythagorean theorem, the actual inaccuracy for a given vehicle update is always equal or greater to the calculated inaccuracy. Yet, with the limited information available, there is no way to know how far the projected point is off along the track compared to the real vehicle position. Thus, looking only at the distance to the track for a given update does not suffice to evaluate a given GNSS sensor reading, since two updates with the same calculated distance to the track might be vastly different in accuracy, as shown in Figure 2.3.

A better evaluation can be performed by considering more than just the current received and corrected update: GNSS signal accuracy is dependent upon the real-world surroundings for the tracker, e. g., signals sent from within a forest leads to a decrease in accuracy [LS14]. Thus, it can be assumed that multiple received, spatially close signals overall suffered from the same decrease in signal quality. Furthermore, it is statistically likely that for a stationary vehicle on the track, received positions would differ due to the inaccuracies and be distributed within some radius around that vehicle. Therefore, the calculated inaccuracies for some points

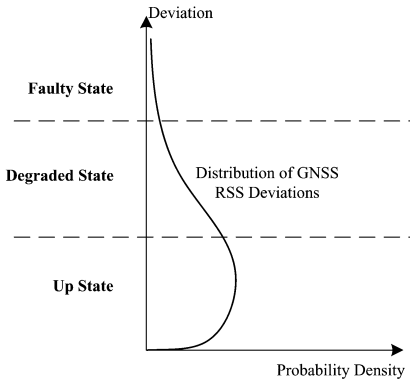


Figure 2.4. Expected deviation distribution for calculated update distances from the track. Adapted from [LS14].

is almost equal to the actual inaccuracy whereas for other points, the calculated distance to the track might be much smaller than the actual distance to the vehicle. This issue is illustrated in Figure 2.3. An expected distribution for the deviation of the absolute distance inaccuracies from updates to the track in that case is visible in Figure 2.4.

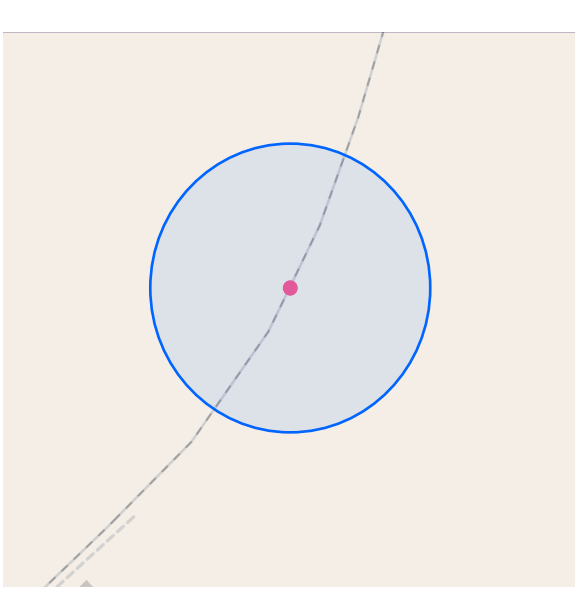
With this in mind, we can now better evaluate the accuracy of a given update position. First, we will store all updates, their projected points, and the computed distance of that update to the track. Then, given the projected position of a new update on the track, we can check for these historic update projections and their corresponding calculated inaccuracy within a predefined radius r_{dist} around that update. To check whether a historic point is within that radius, the distance from the given update projection to previous projection can be computed using the great-circle-distance between the two positions. We check within a radius of the projected point instead of the received, inaccurate point for two reasons:

1. Received update positions, even after being initially projected onto the track, will rarely overlap exactly. However, as previously outlined, any GNSS signal inaccuracy will be noticeable in an area around the given point and thus also be measurable in that area.

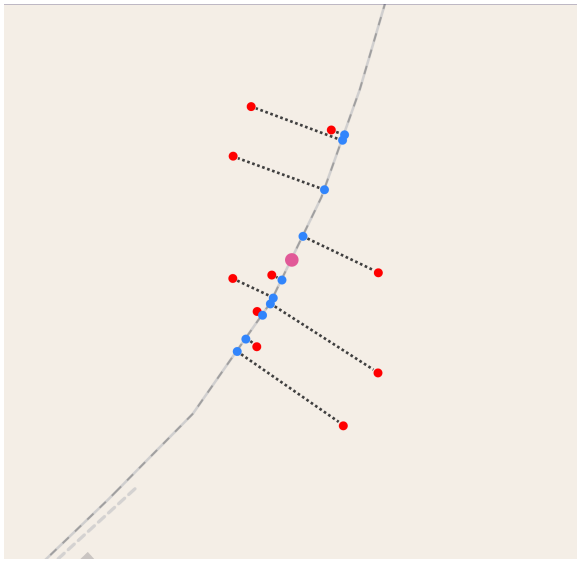
2. If we looked for previous updates around the inaccurate update positions, we might find no spatially close previous updates due to the inaccuracies, especially for large outliers where the distance to the track is larger than the specified radius r_{dist}. Utilizing the positions projected onto the track instead ensures that spatially close previous updates and their inaccuracies are utilized in the following calculation.

This gives a set of tuples P_{near} of close, historic points and their calculated distance to the track as seen in Figure 2.5. For further evaluation, the distances are especially interesting: We can now compute the average distance for the sample set of received coordinates to the track in that area using the corresponding formula

$$\mu_{hist} = \frac{\sum_{(p_{proj},\delta_{tr})\in P_{near}} \delta_{tr}}{|P|}$$



(a) Search area with radius r_{dist} around a point projected on the track.



(b) Found historic raw points, historic projected points, and their distances to the track around the projected point.

Figure 2.5. Search area and historic point information around a projected point.

and the standard deviation with the formula

$$\sigma_{hist} = \sqrt{\frac{1}{|P|-1} \sum_{(p_{proj},\delta_{tr})\in P_{near}} (\delta_{tr} - \mu_{hist})^2}$$

The latter formula can only be used for $|P| \geqslant 2$, so only if at least two previous points were spatially close. For this case, it even makes sense to ensure that $|P|$ is larger than a defined threshold p_{min}, so that at least p_{min} spatially close previous updates are required for utilizing the historical data. A set P with few members can occur for the following reasons:

- ▻ Few previous updates are available.
- ▻ The position was received in an area where usually no signal is available for GNSS trackers to send their location.

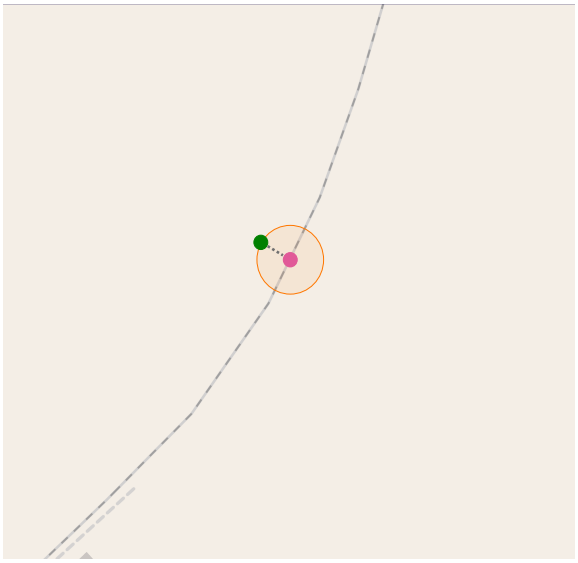
In both cases, with little information available, utilizing the values calculated earlier is prone to errors, as with few to no previous values available, the calculated average is prone to outliers and therefore unreliable. Thus, in a scenario where $|P| < p_{\min}$, we fall back to not using historical data in that area, as this might introduce more errors. Instead, we fall back to calculating the global historical average for all available previous data points in order to obtain measurements to compare the point to. In the case that $|P_{global}| < p_{\min}$, we instead use a predefined distance d_{manual} for the following comparison. However, both cases rarely occur in practice, as after some time, enough data should generally be available for this evaluation.

We then calculate an inaccuracy distance based upon this data to compare the gap between the raw and projected update position to. This distance is calculated with $d_{hist} = \mu_{hist} + c_{hist} \cdot \sigma_{hist}$, where c_{hist} is a constant factor defined beforehand. This information now provides us with a way to evaluate updates: First, we can check whether the calculated gap of the given update to the track is smaller or larger than the just calculated distance. In the case that it is smaller, we can assume that the update's inaccuracy after projection on the track is most likely closer to the calculated inaccuracy, given that the inaccuracy along the track remains unknown, thus we take that the first distance a reference for the given update. This distance is expected to more accurately reflect the actual inaccuracy. Furthermore, by taking the larger discrepancy, it is more likely that the actual vehicle position is somewhere in the determined inaccuracy range compared to the calculated one-dimensional error for the given update. The difference of inaccuracies for this case can be seen in Figure 2.6. Should the received signal's distance be larger than this distance in that area, we can assume the newly calculated distance to be a better measurement for assuming the inaccuracy, as the signal seems to be of degraded quality. This provides us with a better way of estimating the quality of received signals, as with one degree of the inaccuracy unknown, utilizing the higher distance calculated based upon the historic data is expected to be closer to the real inaccuracy than the one-dimensional determined error for the update. Furthermore, the inaccuracy determined here can be utilized at a later stage to display to users how accurate the current update is expected to be. Whilst this inaccuracy based upon the historic data will be larger than the calculated distance to the track, this ensures that do not communicate updates as too accurate to users: The exact error for the given position remains unknown, thus displaying a larger inaccuracy then initially can help in conveying this uncertainty to clients. We call this chosen distance δ_{acc}.

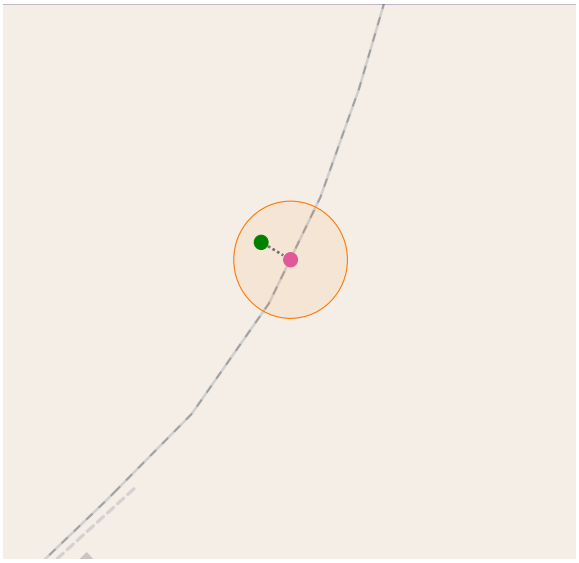
2.4 Modelling Vehicles on Track Segments

While the aforementioned steps now provide us with an actual vehicle position on the tracks and methods to evaluate the update accuracy, we are still considering vehicle updates only individually. Thus, the position of other vehicles is currently not considered further. However, it can be assumed that vehicles will not change their order once they are driving on the tracks. This leads to the following scenarios, given that we primarily consider on-demand vehicles and vehicles without a schedule:

1. Given two vehicles where one is directly following the other vehicle and both are driving in the same direction on the track. For example, this might happen if two maintenance vehicles are driving from their depot towards the construction site at the start of their shift or if a group of tourists occupies two trolleys on the *REAKT* track and proceeds to drive in a convoy. If the vehicle directly behind now sends an update which lead to the assumption it has overtaken the other vehicle based upon the previous update from the leading vehicle, we can assume that in reality the other vehicle also progressed its position, as otherwise the vehicles would have changed their order on the track.



(a) Update inaccuracy prior to considering historic data.



(b) Refined update inaccuracy after considering historic data.

Figure 2.6. Difference of update accuracy evaluation before and after considering historical data.

2. Given two vehicles driving in opposite directions on the same track and driving towards each other. This scenario might e. g., if two maintenance vehicles are working spatially close at the construction site or if two groups of tourists where one group started driving in Malente and the other in Lütjenburg meet halfway on the *REAKT* track. If one of the vehicles sends coordinates that indicate it has overtaken or is spatially close to the other vehicle based upon its last coordinate reading, we can assume that at least one of the vehicles has come to a halt or will be about to stop. This is due to the constraint that most vehicles cannot easily change order on a single railway track, as most vehicles cannot leave the rails at all. In the case that at least one of the vehicles can leave the track, this vehicle will have to come to a full stop in order to do so, e. g., trolleys on the *REAKT* track have to be stopped and lifted out of the tracks and road-rail vehicles would also be required to stop in order to change wheels.

Furthermore, we can assume that vehicles on the same path but not spatially close to each other will not be affected by updates of other vehicles as well as spatially close vehicles on different tracks will also not affect each other - in both cases, despite the vehicles moving the other vehicle will most likely not be affected by that movement because of distance and track constraints respectively.

However, so far we have no proper way of knowing where a vehicle is along the track - we only project the vehicle onto the closest position on the track. We do not consider the direction or speed the vehicle is driving at and how the vehicle would progress along the path considering this information. Thus, the next step is to not only correct the inaccuracies of updates, but also model an accurate state of the vehicles on the track where we can easily

track the expected progress of vehicles and check for spatially close and possibly affected vehicles from received sensor updates. With this model, we then can easily check for spatially close vehicles, their progress along the track, compare the available data, and then decide on how to update vehicle predictions.

2.4.1 Combining Track Nodes and Edges into Larger Segments

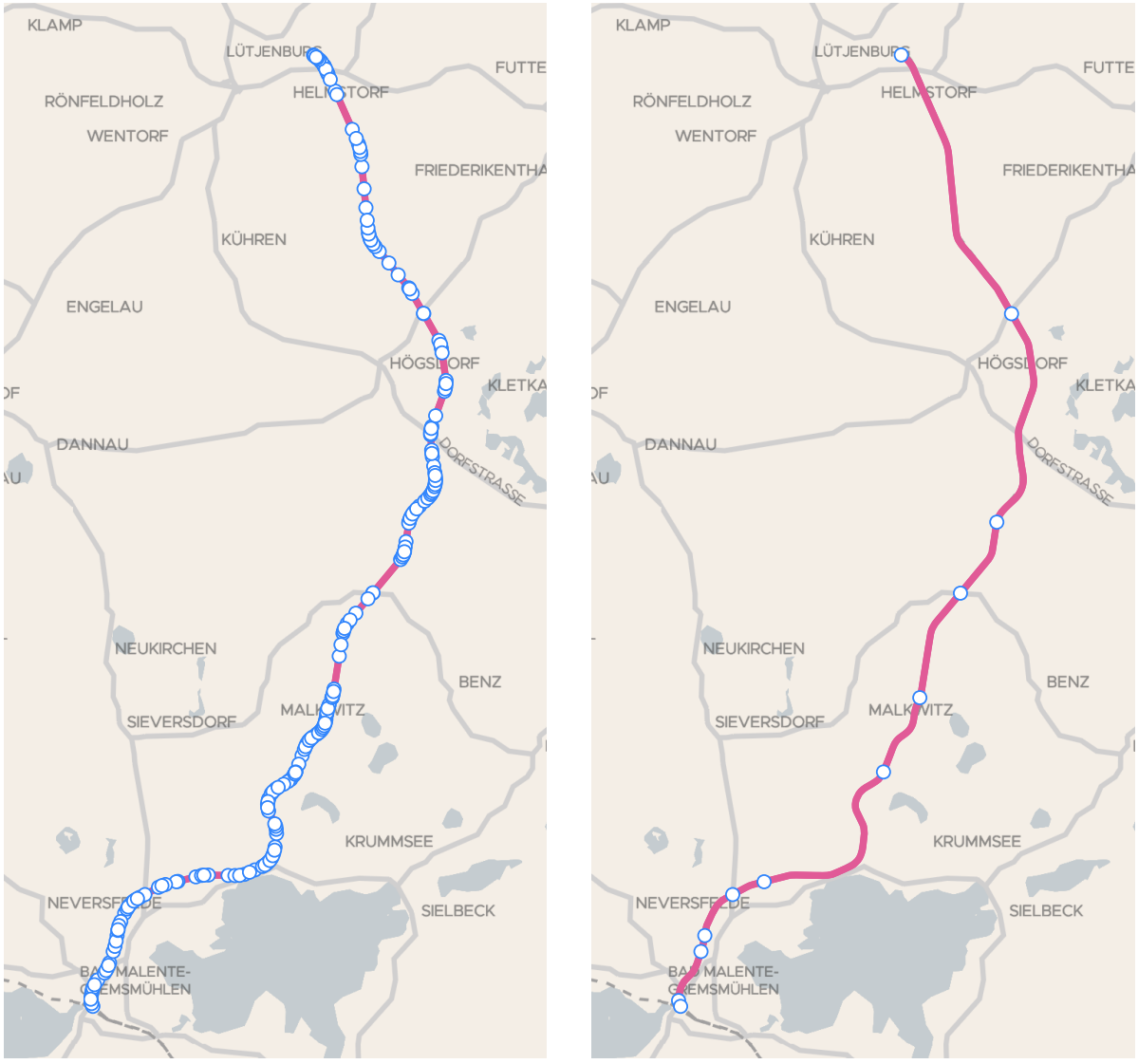
Currently, for the calculation of the spatially closest point, we look at each node and edge extracted from OSM individually. Yet, vehicles do not drive along singular edges, but more rather paths along multiple edges, as the edges extracted from OSM only describe a direct, straight path exists from node a to node b and nothing else. Additionally, vehicles might be on the same path and spatially close, yet not on the same edge. Thus, in order to better determine which vehicles share a path and how far progressed along that path vehicles are, we remove uninteresting nodes and combine edges connected to removed nodes into segments so that we can model vehicles along these larger parts and not singular edges. Still, we have to keep the geographical information of nodes and edges for correcting the coordinate inaccuracies of updates.

For deeming a node interesting or uninteresting, criteria are needed for classifying the nodes. Generally speaking, nodes are relevant if any of the following conditions is met:

- They have any amount of edges connected to them other than two.
- They have a key-value pair attached to them that seems interesting.

Nodes with any amount of edges other than two are interesting as these provide information about the track layout, such as the ending of a track or railway switches. As vehicles can join different track parts depending on the switch configuration, we should also separate segments at such interesting points. Furthermore, we might have additional interesting points with just two edges connected to them, e. g., stations for railway networks, railway crossings in the context the trolleys in the *REAKT DATA* project or depots for maintenance vehicles on tracks. As vehicles will not simply drive past these points, it seems reasonable to end track segments at such nodes. The state of the track beyond that node for a vehicle approaching that point is barely relevant as long as the vehicle does not cross the position of the node. Once it does cross this point, the state can be considered, yet splitting segments reduces the need for checking whether vehicles could affect one another at such interesting points.

With this in mind, we can now combine edges into larger segments and form a simplified graph of the network. For all uninteresting nodes, we remove these and join their two connected edges into one. Furthermore, during the joining of edges, we remember which node was removed in order for these edges to be joined, and keep this information ordered during the removal, so that upon completion, we know which node belongs to which edge and which other nodes were removed before and after a given uninteresting node for that segment. After removal, we thus know for two connected interesting nodes which uninteresting nodes geographically lie in between them. This leaves us with a simplified graph consisting of only



(a) Track before segmentation and simplification.

(b) Track after segmentation and simplification.

Figure 2.7. Comparison of the Malente-Lütjenburg track before and after simplification.

(a) Projected position on track without reference to track.

(b) Position along segment with distance along track (blue) from selected start node (red).

Figure 2.8. Difference between a raw projection and a calculated position along segment.

interesting points whilst keeping the possibility of identifying corresponding edges via given nodes. The effect of this simplification can be seen in Figure 2.7.

2.4.2 Connecting Vehicle Positions to Segments

Converting the projected positions for vehicles into distances along a given path is now somewhat an easy task. As we previously already computed the two nodes the vehicle projection is in between, we can now search for a segment both nodes are in. As segments contain at least two nodes, since segments consist of at least one previously not joined edge, and as the computed nodes are connected via an edge, such a segment definitely exists. Once the segment is found, we can compute the distance of the vehicle along that segment. We take one of the previously determined nodes for the projection and gather along the determined segment all the nodes coming before that node utilizing the information retained in the last step. This search includes the interesting nodes at the start and end of segments. The direction we start searching the segment is irrelevant at this point, we only have to remember from which node we started the search. We again retain the order of found nodes. Then, for all nodes found, we calculate the geographical distance from the given node coordinates to the next found node position. For the last found node, which has no next node, we instead compute the distance from that node to the node we performed the search with. Then, we sum up all the calculated distances to get the distance from the start node to the node we searched along the edge with. In the case that the start node is the node we searched with, this distance is obviously 0.

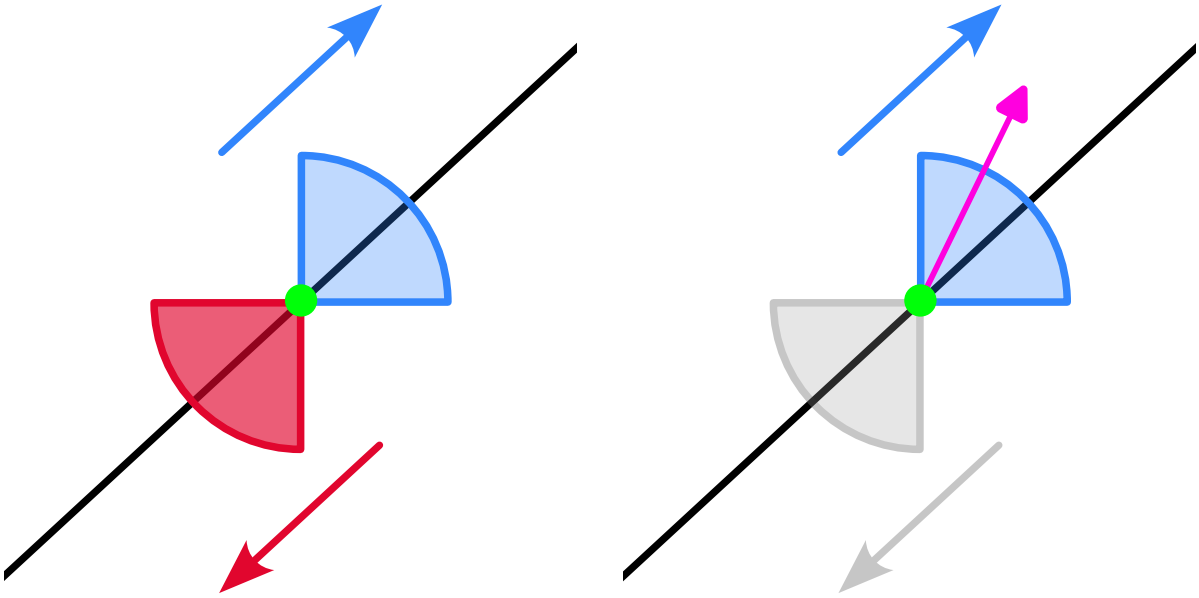
Finally, we only have to compute the distance of the vehicle along the previously determined edge and add or subtract that to the previously determined length. For computing

the distance along the edge, we use the geographical distance between the node coordinates and the computed point in WGS 84 projection. Then, we check if the node of the determined nodes for the update we did not search for was the last found node in the search. If that is the case, we subtract the distance from the summed distance as we already added the entire length of the edge the vehicle is on to the progressed length. If the node we did not search for was not the last found node, we simply add the calculated distance from the node to the vehicle to the previous sum, as the length of that edge was not yet considered. The resulting summed distance is then the distance of the vehicle along that segment from the previously chosen start node for that segment as seen in Figure 2.8. With this information, we now know which track segments vehicles are on and can determine how far the vehicles are progressed along that segment.

2.4.3 Directional Vehicle Movement along Segments

The progress of vehicles along a given track with a start node can be retrieved, however, we currently do not know whether the vehicle is actually driving away or towards the decided start. So far, the directional heading of a vehicle is not considered nor known. Yet, this information is required in order to know whether vehicles are driving in the same direction or opposing directions. To determine the direction a vehicle is driving along its assigned segment, we have to differentiate between two cases: The case that vehicle heading and speed are available and the case that no such information is available. For the first case, we utilize the heading information for the vehicle $\mathcal{H}_v$ and compute the directional heading $\mathcal{H}_v$ of the edge the update position was projected upon by computing the heading from one node to the other node. For this calculation, the choice of node order is irrelevant, it only has to be remembered which node was chosen as the first node. We can then compare the two headings: If $|\mathcal{H}_v - \mathcal{H}_e| \leqslant 45^\circ$, we can now assume the vehicle is driving from the first chosen node to the other node along that edge. In the case that $135^\circ \leqslant |\mathcal{H}_v - \mathcal{H}_e| \leqslant 225^\circ$, we know that the vehicle is driving along this edge from the second node to the chosen node. We allow a window of 90° for the tolerance of readings due to possible inaccuracies in updates. A visualization for this concept and how it would apply for a given vehicle heading can be found in Figure 2.9. In all other cases, we assume a too inaccurate sensor reading, as vehicle and edge heading do not seem to match, thus the information is discarded, and we handle the update as if no heading information was available. Heading information of the vehicle may still be determined at a later stage through the use of other information available. In the case that a received speed reading is negative, we change the previously determined node to the other available node and swap the speed reading to a positive reading, so that the vehicle is driving forwards along the track again. This removes unneeded complexity from modelling the directional movement, as the vehicle's direction along the segment is now known by only considering the resolved node order.

With the vehicle heading now known, we can check with the path computed in the previous step whether the vehicle is actually progressing away from the assumed starting node or whether it is driving away from the other node available for that segment. We can



(a) Concept for determining vehicle heading with tolerance windows for each direction shown.

(b) Example for inferred vehicle direction (blue) based upon a given vehicle heading (pink).

Figure 2.9. Determining vehicle direction along a given track segment.

easily check this by validating that the node the vehicle is driving away from for the current edge appears in the calculated node path before the node of the current edge the vehicle is driving forwards to. Thus, if the node the vehicle is driving towards to was not in the calculated node graph or was the taken node from the previous step and the node the vehicle is driving away from was the last found node, the heading of the vehicle was randomly correct and the chosen starting node of the segment and computed distance along the segment were correct. In the case the above condition was not met, the vehicles progress along the segment is starting from the wrong node of the segment, thus we have to swap the start node and then recompute the progressed distance. Recomputing the progressed distance can easily be done by computing the distance of the entire segment and subtracting the progressed distance from that length. With this now done, we have directional headings for the vehicles on the track.

For the case where no information about the vehicle's heading is available, we cannot infer any direction from the information available for the vehicle alone. In this case, we assume the vehicle is idle and not moving for now.

Yet, in both cases, we now have corrected the vehicle position reading, assigned the vehicle to a track segment, know how far the vehicle has progressed along that segment and, for the case that the heading is known, determined the direction the vehicle is driving. All this enables us to move from an individual GNSS sensor reading towards a model of the

sending vehicle which we can compare and set in relation to other vehicle models on the track. Furthermore, we can now apply this method to not just one, but all vehicles we have received a position reading for. Lastly, this enables us to track the progress a vehicle has made in the case of a newly received update, which we will now take a closer look at.

2.5 Handling New Updates for Known Vehicle

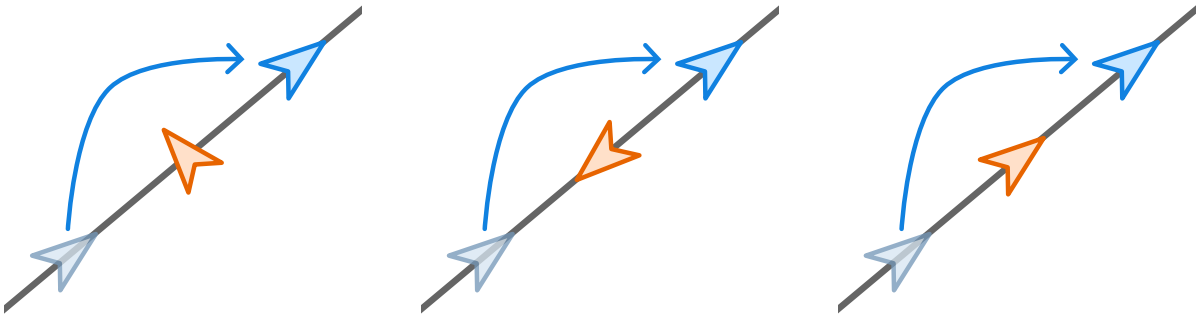
So far, we have only looked at the first given GNSS position update for any vehicle. However, in practice we will receive updates over and over for a given vehicle and want to update its position accordingly. While these have to be processed as previously described, we can infer additional information from the previously determined position for that vehicle. Given the previously calculated position along the corresponding segment and the newly projected position along its segment. From that information, we can determine the distance the vehicle travelled and the path the vehicle took in between updates.

Combining the information from the two updates, we have a point the vehicle was previously at and a position the vehicle is at now. First, we can check whether the vehicle has moved at all: If the geospatial distance between the two points is smaller than any of the two δ_{acc}s calculated in Section 2.3, we assume the vehicle has not moved and is currently not moving, as the distance traversed is most likely just an inaccuracy of sensor readings. In all other cases, we calculate the distance and path taken by the vehicle. Should both readings still lie on the same track segment, the travelled distance can be computing by first ensuring that both readings have the same start node and then calculating the absolute difference of the progress of both data points. The conversion for the case of different start nodes on the same segment was described in Section 2.4.3. Should the two coordinates from the readings not lie on the same segment, we determine the path the vehicle took along the track utilizing the previous projected position on the track and the newly projected position, which segments it passed and compute the distance travelled. The distance travelled can be determined by summing the remaining length of the previous segment the vehicle travelled on, the length of all passed segments and the progressed distance of the vehicle's current segment using the start node which is a part of the last previously passed segment. The latter is important as the vehicle might have recently changed directions based on sensor readings and thus might have a different point of reference as a start on the current segment.

With this, we now know how far the vehicle travelled in the given time interval, the path it took during that time interval and, lastly, can compute the average speed of the vehicle from the previous position to the current position by dividing the travelled distance by the duration of the passed time interval.

2.5.1 Progressing Adjacent Vehicle Projections on Position Updates

Utilizing the newly available information, we are now able to enforce the constraint that the order of vehicles on the track will not change under normal circumstances and consider



(a) Passing of a non-moving vehicle.

(b) Passing of a vehicle moving in the opposite direction.

(c) Passing of a vehicle moving in the same direction.

Figure 2.10. Different possibilities for a vehicle update causing the passing of another vehicle.

this information for the vehicle projections accordingly. As the path the vehicle progresses is now determined, we can check for any vehicles currently known to be present anywhere on that path. In the case that no vehicles were seemingly passed, we do not have to update any other vehicle projections. However, for the case that other vehicles are present on the path the vehicle has passed, we have to differentiate between the three scenarios that can occur and check for the applicable scenario whether we have to update other vehicle projections. The three different possible situations are shown in Figure 2.10. To handle these, we proceed as follows: First, we check whether the vehicle passed any vehicles along its progressed path. If no vehicle was passed, no progressing of other vehicle projections is needed. Otherwise, in the case of a passed vehicle, we check whether the passed vehicle was driving towards or away from the current vehicle or if the vehicle was currently idle.

We handle the different cases as follows:

- In the case that the other vehicle was idle and thus not moving, we first check whether the length of the geodesic between both vehicle projections is smaller than the summed inaccuracies, of both vehicle projection. If this is the case, this implies the updates were inaccurate, and we expect the indicated passing to just be an inaccuracy, restore the ordering of the vehicles and use the position of the more accurate projection of both vehicles as a reference point. We then move the vehicle with the more inaccurate reading in front of the other vehicle, respecting the order of vehicles on the track. In the case that the distance between vehicles is larger than their summed inaccuracies, we have to differentiate whether the passed vehicle can be passed on a single track whenever it is not moving or whether it cannot be passed. Supposing the other vehicle cannot be passed, we move the other vehicle in front of the currently updated vehicle, as we expect the other vehicle to have started moving again based on the data available.

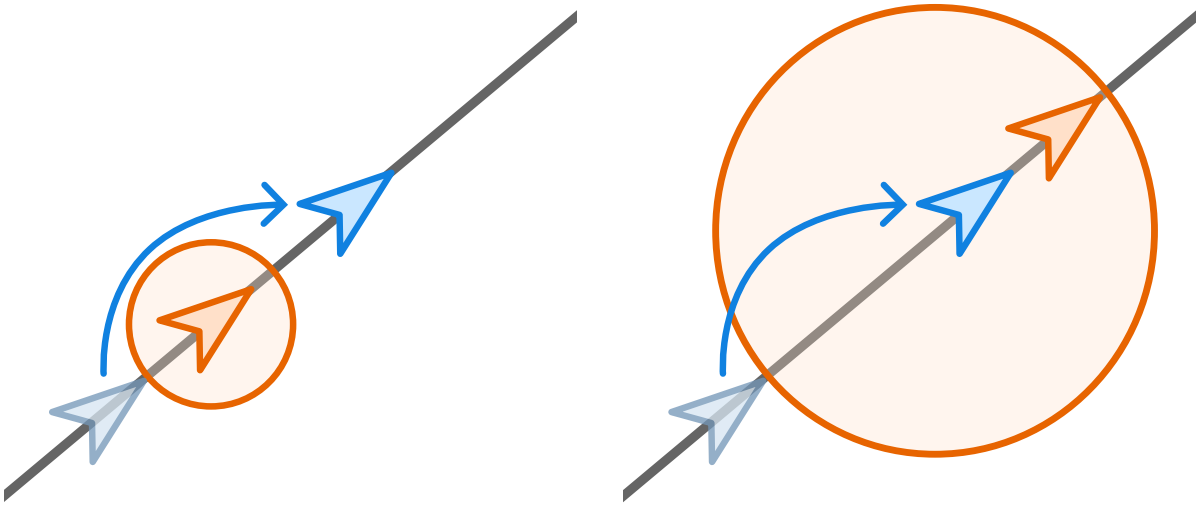
  If it can be passed, we expect the vehicles to just have passed each other, as the idle vehicle most likely let the other vehicle pass, so the passed vehicle is not updated.

▷ In the case that the other vehicle is moving towards the current vehicle, we again check at first whether the change of order could just be an inaccuracy of the GNSS sensor readings based upon the provided inaccuracies for both vehicle projections. Should that be the case, we recompute the vehicle projections following the behavior described in the previous case and are done for this update. Otherwise, we differentiate based upon whether the other vehicle can be passed whenever it is idle or not: In the case that the other vehicle can be passed, we assume that it came to a full stop some time after the last update and let the current vehicle pass. Thus, we change the order of the vehicles on the track and can leave both projection positions as-is, since there is no need to update any of the projections or their corresponding inaccuracies. Is the other vehicle known as not being able to let other vehicles pass, we assume for this case that the vehicle changed the direction along the track sometime since its last update. Thus, we update its position to be just ahead of the currently updated vehicle. For this, we introduce a predefined distance of d_{min} between vehicles as a gap to use for this scenario. We now can update the passed vehicles position using this distance, which ensures the order of vehicles on the track is retained even after this update. However, as the approach was primarily tested for vehicles that can leave the track, the assumption made here requires validation and evaluation in the future.

▷ For the last case where a vehicle has passed another vehicle which is currently expected to be driving in the same direction, we do not differentiate between passable and non-passable vehicles and first restore the previous ordering. Then, we check whether the length of the geodesic between both projections is smaller than the sum of computed inaccuracies as with the other cases. Should the length be smaller, we attempt to move the updated vehicle to a position within its calculated inaccuracy where it would stay behind the other vehicle. Otherwise, we also move the seemingly passed vehicle to a new position within its calculated inaccuracy where the previous ordering would be restored. Such two positions exist as otherwise the inaccuracies of the vehicles would not overlap.

In the event that their inaccuracies along the track do not overlap, we instead move the passed vehicle according to its previous speed readings for the passed time frame since the last received update for that vehicle, and calculate a passed distance on the associated track segment for the given time frame. For the case that the calculated distance does not suffice to place the vehicle anywhere within the inaccuracy along the track of the updated vehicle, we instead compute the minimum distance required to place the passed vehicle in this inaccuracy range. Then, we check whether this would move the passed vehicle from the track segment it is currently expected to be on. Should this be applicable and the next segment after the current segment not be uniquely identifiable, we only move the passed vehicle to the end of the current segment and not further, as we do not know the exact path the passed vehicle might have taken, and leave the updated vehicle projection as is.

For all other cases, we progress the passed vehicle by the distance previously determined and, in the event that the progressed distance should not restore the previous vehicle ordering, move the updated vehicle behind the passed vehicle. Due to the previous



(a) Vehicle inaccuracy before position push due to other vehicle.

(b) Vehicle inaccuracy after position push due to other vehicle.

Figure 2.11. Vehicle inaccuracy change after position update caused by other vehicle.

constraints, this should still place the updated vehicle somewhere within its calculated inaccuracy range.

In all cases where we moved the other vehicle without knowing its actual current position solely due to the likelihood of the event, we have to consider that the projection of the passed vehicle leads to a high inaccuracy in the resulting vehicle position, especially for vehicles that can leave the track. In an effort to do so, we recalculate their δ_{acc} to indicate that the inaccuracy spans the entire area from the previously received update inaccuracy for that vehicle to the newly projected position if the vehicle can be passed on the track. This is illustrated in Figure 2.11. For vehicles that cannot be passed and thus have to have moved, the new inaccuracy for the vehicle positioning ranges from the newly projected position to the furthest possible inaccuracy for the updated vehicle, as the vehicle will be some place in front of the other vehicle. This ensures in both cases that the uncertainty of this estimate, while being partly based on evidence, is considered in follow-up calculations and can be communicated to users at a later stage. This can lead to very large inaccuracies for vehicle positions in the case that a passed vehicle was moved very far, however, the uncertainty of this position is at least known and can be communicated to users at a later stage. It has to be noted that this is still a lower-bound estimate: As the position was recalculated based on the last speed-reading, this might lead to inaccuracies once the vehicle accelerated afterward, however, this issue is currently not further considered. In any case, the uncertainty is recalculated once a new GNSS sender update for the pushed vehicle is received and its position is updated accordingly.

We can then repeat these cases as needed in the case that the updated vehicle passed multiple other vehicles in between updates or that the progressing of other vehicles caused new vehicles to be passed.

2.6 Removing Outdated Vehicles from the Model

In the case that we receive no GNSS sensor reading for a given vehicle in a very long time, the previously calculated path taken along the track could trigger a lot of vehicle updates whilst being inaccurate, since we do not know how exactly the vehicle behaved in between the updates and other vehicles might have had more frequent updates which enabled us to estimate their behavior accurately. Furthermore, there is no way to validate whether the currently projected position for a vehicle is still accurate after a long time interval and the possibility for erroneous projections is very high. This is especially true in scenarios where vehicles can leave the track or be passed by other vehicles on a single track. Thus, we establish a maximum time interval $t_{\max}$ after which we remove vehicles from the simulation in case no new GNSS update was received for that vehicle in the set time interval in order to prevent this behavior.

2.7 Predicting Vehicle Movement for the Near Future

At this point, we have computed a best-effort projection for each vehicle on the track, having considered historic updates, other vehicle positions on the track and locations on the vehicles segment that cause special behavior. However, with all this information available, we can attempt to go one step further and predict the vehicle movement for the near future. As we already know the vehicle's current speed, its heading, the segment it is on and, as long as we have received signals from them, the vehicles currently it, we have a chance to predict a realistic movement for vehicles in the near future. However, in order to predict the vehicle movement, we have to define the behavior we expect of vehicles in order for the simulation to be as realistic as possible. We know that vehicles will follow the laws of physics and thus accelerate and decelerate instead of abruptly coming to a stop or continue driving.

Furthermore, we assume that

- ▻ vehicles follow a definable path along the network for the near future;
- ▻ vehicles will not suddenly change direction on the track;
- ▻ all vehicles will drive in a similar pattern at a given position, e. g., they all slow down at a specific position due to a slope there;
- ▻ vehicles driving towards each other will gracefully come to a full stop;
- ▻ vehicles following each other will not overtake one another;

▻ vehicles will gracefully come to a stop at locations with a defined special behavior.

While we already implicitly and explicitly took some of the mentioned assumptions for granted earlier, the approach chosen earlier based these assumptions primarily on evidence in form of newly received vehicle location updates that would contradict one another. However, for the prediction, we base our vehicle movement on nothing but previously available information in order to predict the movement for the near future. This will lead to errors and situations where the prediction was totally off, as just one wrong assumption breaks the modelling process. Nevertheless, we assume here that the preconditions are met in the majority of times - whether this theory stands can be evaluated against a given track at a later time. In any case, the uncertainty of the predictions should be communicated to users so that they do not take the displayed positions as certain.

The remaining procedure is as follows:

1. First, we predict the path the vehicle will take along the track given a certain direction along the track.
2. Then, we try to estimate the vehicle speed and change of speed along that path.
3. Afterward, we check for any obstacles, such as other vehicles or locations with defined special behavior, compute how these might affect the vehicle and specify how the prediction will behave on such events.

Finally, we define how the algorithm shall recompute predictions in the case of newly received updates for that vehicle.

2.7.1 Predicting the Near-Future Vehicle Path

The following part is highly speculative and primarily works for a network consisting of only one track, thus, this will need more work in the future. Yet, the advantage of this step is that vehicles could move beyond switches for more complex networks during the vehicle movement prediction. With a near-future path predicted or determined based on other evidence for the vehicle, a decision can be made which path a vehicle will take at a certain switch. With the restriction that only GNSS information is known, however, this decision cannot be made properly yet.

As we want to move from a projection to a vehicle path prediction, we have to assume that the vehicle might progress beyond the segment it is currently on. Yet, at the current time, we only assign vehicles to one path segment, thus it is unknown which path the vehicle will take after leaving the current segment. In order to find a path the vehicle is most likely going to take, we search for the closest node in the direction the vehicle is currently heading which satisfies at least one of a set of manually specified criteria for this classification task. This node is then the determined expected endpoint for this vehicle. This might be for example a node which is classified as a railway stop in the context of public transport or a depot or construction site in the context of maintenance vehicles on tracks.

Once this endpoint is found, we compute the path the vehicle has to take from its current segment to this found node. This yields us a path we predict the vehicle will probably take for the near future considering its current heading on the track.

At this point, it has to be mentioned again that this method of determining the expected vehicle path is highly speculative and will yield faulty results for larger and more complex networks. In more complex scenarios, one might have to classify possible endpoints depending on the direction of travel on the track, whether another vehicle is already heading towards this node, or utilize additional information available. However, specifying the exact behavior in these scenarios went beyond the scope of this thesis. The method in its current state works well for the *REAKT DATA* track because there is only one track.

2.7.2 Vehicle Speed Approximation over Time

Having a guess for the future path for the vehicle, we still need to model how we expect the vehicle to progress along that determined path. Whereas the projection only estimates the vehicle's position for one given point in time, the prediction aims to provide a position reading for a vehicle at any given point in time. This means we have to approximate the vehicle's speed, as this will primarily affect how the vehicle progresses along the given path.

This is initially not too difficult: In the case the GNSS sensor indicates the vehicle is not moving, or we previously determined the vehicle is currently not moving, we also predict the vehicle to not be moving, as the given evidence suggest just that. Otherwise, if the update has speed-information attached to it, we just use this information, as this is the most accurate data about the speed of the vehicle we have. Should this information not be available, we can use the calculated speed from Section 2.5. Using this reading can lead to issues though, as this calculated the average speed of the vehicle for the passed distance during the time interval between position updates and does not actually reflect the current speed. For example, if the vehicle came to a full stop during the time interval, the real velocity would be much higher than the previously calculated velocity. Yet, this measurement is still better than no reading at all.

One issue still remains: While we now determined a speed estimate for the vehicle, this value is currently constant which means we predict the vehicle to move with a constant speed along the given path. This is highly unrealistic, as due to the nature of the track, e. g., hills or parts where vehicles have to slow down, the vehicle will change its speed over time. Modelling it with a constant velocity despite these characteristics of the track will result in very unrealistic predictions. In an effort to improve this, we now will try to model the acceleration and deceleration of the vehicle along its given path over time using historic and current data available for the track.

Modelling Vehicle Acceleration over Time using Historic Speed Information

For estimating the vehicle acceleration over time, we consider previous speed readings from GNSS sensors if available. Furthermore, we can use the previously calculated average speeds

as described in Section 2.5. Yet, while the sensor readings provide a measured speed at a given point, the calculated average velocity rather describes the speed for an entire given distance along possibly multiple segment and not at a specific point. This makes it difficult to combine data from both datasets in the current state.

In an attempt to resolve this issue, we can attempt to assign the calculated average speed a point on the traversed part of the track where this reading is possibly most accurate. The chosen way for this thesis to solve this issue is to take the middle point of the used track part, as this seems to be the most likely place for the calculated average speed to be similar to the actual vehicle speed. However, this method could be improved in the future.

With historic information about vehicles speeds along the track now available, we can query this information similar to the approach described in Section 2.3: Given the currently expected position P of the vehicle and its heading, we can look for previous speed readings around this position which also had the same heading along the track segment as current the vehicle in a radius $r_{velocity}$ we choose beforehand. We again use a radius here due to the fact that we will not receive sensor positions at the exact same places. Furthermore, the direction along the track is considered as e. g., vehicles might progress slower along a part of the track in one direction compared to the opposite direction due to driving uphill in one and downhill in the other direction. This provides us with an estimate of the average velocity v_{avg_P} of previous vehicles for the given point. Using this average velocity, we can now compare the previously determined speed for the vehicle to the average reading by dividing the determined speed with the estimating average velocity: $c = \frac{v_{vehicle}}{v_{avg_P}}$ The result c for this calculation lets us know how fast the vehicle is currently progressing compared to previous vehicles in this area. In cases where $c < 1$, the vehicle is progressing slower than average, whereas for cases $c > 1$, the vehicle would be progressing faster.

Knowing the current relation between the vehicle's speed and the average speed of all vehicles for the current position, we use this information to predict the change of speed of the vehicle along its predicted path: Given the predicted path, we progress the vehicle's position along that path by $d = r_{velocity} * 2$ and calculate the corresponding geographical coordinates of the point on the track. This can be done by reversing the method described in Section 2.4.2. For this point P_1 on the track, we again determine the average speed $v_{avg_{P_1}}$ of vehicles heading in the same direction. With this new average, we then try to estimate the vehicle's speed at that position. Given the previously determined c, we compute the expected vehicle velocity at that point with the formula $v_{vehicle_{P_1}} = c \cdot v_{avg_{P_1}}$. This predicts the vehicle speed based upon the historic information as well as the difference of the current vehicle compared to the average vehicle velocity. We then determine how we need to accelerate the vehicle over the distance of d from the currently predicted speed to reach the determined speed by determining a function $a(t)$ that accelerates the vehicle as described over the given distance. This can be for example done linearly or by using more advanced acceleration models for the vehicle. Afterward, we calculate the time $t_{prog_{P_1}}$ it takes the vehicle to travel the distance of d considering the initial velocity at the start of the distance as well as the defined acceleration behavior. With the simulated progressed distance along the track, the newly calculated speed, the determined

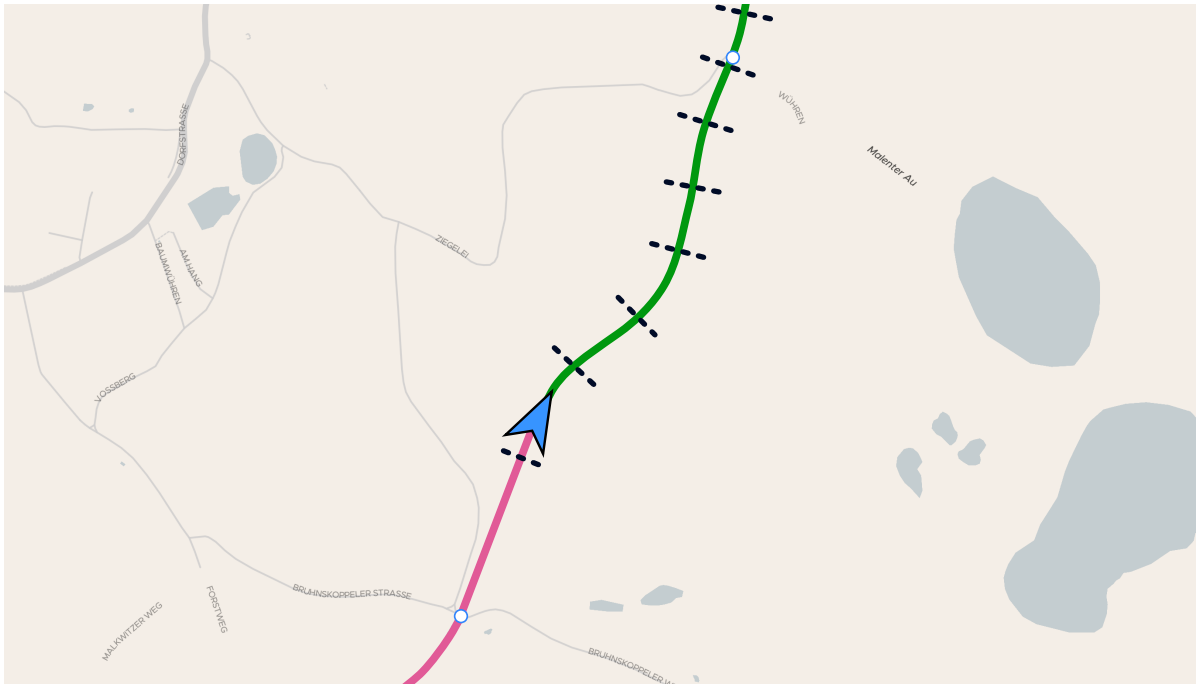


Figure 2.12. Predicted path segmentation for vehicle acceleration approximation.

acceleration for that segment along the path as well as the calculated time it takes to travel the distance along that segment, we repeat these steps until

- the vehicle would reach or exceed the end of its predicted path with the next simulated progress d or
- the sum of calculated travel times t_{prog} exceeds the time $t_{\max}$, as after this time, the vehicle is removed from the simulation as described in Section 2.6.

The result of this repetition for a predicted vehicle path is illustrated in Figure 2.12. After we stopped repeating those steps, we can combine the calculated acceleration behaviors into one function which then provides us with a predicted estimate for the velocity change of the vehicle over time along its predicted segment. This is again speculative, as this prediction is not based upon evidence for the current trip, however, through the use of the historical information, the goal is to provide a better estimate on how the vehicle will move along the track. Yet, this approach was neither implemented nor evaluated, thus, it would have to be tested and possibly improved in the future.

Recognizing Obstacles in the Vehicle Path Prediction

While we now obtained a more accurate prediction for the vehicle's speed based upon historic data, we so far did not consider the current data available about other vehicles on the track

or special circumstances known for some locations on the vehicle's path, e. g., that vehicles will come to a full stop at the aforementioned stations or depots and not simply drive past these. In order to classify such special locations, we require classifiers for these to be specified in advance. While this information might also be partly available from historic data, current positions of other vehicles is not, thus considering these constraints in a separate step was deemed reasonable.

With this given, we check along the predicted vehicle path for other vehicles or special locations as described before. If neither a vehicle nor a node is found, we leave the prediction as-is, as with no other information available, we have no other way of enhancing the estimated vehicle trajectory. Otherwise, we find the spatially closest object to the current vehicle in the found locations and vehicles. Should the nearest obstacle be a vehicle which is currently not moving and deemed to be passable whenever it is not moving, we ignore this object whilst remembering that it was ignored and instead check for the next closest obstacle in the path. Besides this case, we primarily will meet the three scenarios shown in Figure 2.13, on which we continue as follows:

- If the object is a vehicle driving in the same direction as the vehicle, we check whether the current vehicle would currently pass the other vehicle at any point within the time interval $t_{\max}$ considering the determined acceleration for both vehicles. We only have to check for an intersection during this interval, since after this time the vehicles are removed from the simulation as described in Section 2.6. The passing of another vehicle only occurs if the current vehicle is predicted to drive faster than the other vehicle. In the case that the vehicle would pass the other vehicle, we instead decelerate the current vehicle prediction in such a way that the vehicles keep the predefined minimum distance of $d_{\min}$ as introduced in Section 2.5.1. For the deceleration of the vehicle, we try to decelerate the vehicle with a maximum deceleration value of $a_{\max}$ in a way that the current vehicle matches the other vehicle's speed one reaching the minimum distance $d_{\min}$ to the other vehicle. Should that not be possible in the time frame before the two vehicles seemingly would collide, we instead assume the vehicle came to an emergency stop for the prediction ignoring all other readings, as we deem an emergency stop more likely than the case of vehicles crashing and thus stop the movement prediction. Otherwise, we match the speed behavior of the current vehicle to that of the vehicle ahead of it, assuming that the vehicles will drive in convoy from that point on.

  Once this is done or should the vehicles not pass each other according to the current predictions, we additionally check whether the predicted path of both vehicles deviates after any given point within the time frame $t_{\max}$. Should that be the case, we again check for obstacles in the vehicle's path, excluding the already handled vehicle in that search.

- In the case that the object is a vehicle driving towards the current vehicle or is an idle vehicle which cannot be passed, we calculate the point in time and position when the vehicles are expected to meet each other. Should this point in time again exceed the time window $t_{\max}$, we do not need to update any of the predictions as described before.

Otherwise, with this point now known, we utilize the maximum deceleration $a_{\max}$ again and update the acceleration modelling for both vehicles to decelerate with this value and fully stop in front of each other with the minimum distance $d_{\min}$ between them. This ensures that the vehicles will stop in front of each other and do not seemingly crash during rendering of the prediction.

▷ For the last scenario where the next obstacle is a special location the vehicle will not pass, we again first check whether the vehicle would reach this point in the time frame $t_{\max}$. In the case that it does, we decelerate the vehicle in a way that it comes to a full stop using the previously defined maximum deceleration $a_{\max}$ at a distance d_{node} towards the node the vehicle shall keep. The distance is also defined beforehand. This might be needed e. g., for trolleys driving towards level crossing, as they will not come to a stop on these crossings but rather a short distance in front of them as well as cases where two trains are on the same platform in a given station and the special location is the center of that platform, to which both trains will keep a certain distance. Once we have updated the vehicle acceleration model to reflect this consideration, we again update any vehicle behind the current vehicle if available and needed. This is in an effort to ensure that the updated velocity behavior for this vehicle is also considered for the prediction of other vehicles.

For all cases, no matter whether a prediction was influenced or not, we remember for all vehicles found in the currently updated vehicle's path that this vehicle is in some way in the path of that other vehicle. This is done to ensure that on any prediction update following in the future, we can cascade that update through the simulation and update other predictions according to the newly available information. Whilst at the current time, other predictions might not be influenced by the recalculation of this prediction, this can change quickly: In the case that a vehicle appears in any other vehicle's path, all vehicles behind that vehicle would also have to come to a stop in order to keep up an accurate prediction. By keeping track of which vehicles can influence the position of other vehicles, we can attempt to keep up a best-effort prediction of not just the updated, but all vehicles on the current track.

However, much of this is only a concept and was not properly implemented nor evaluated in this thesis. Thus, some of the above assumptions should be evaluated and possibly refined in future work.

2.7.3 Prediction Behavior upon New Vehicle Position Reading

Whenever a new GNSS sensor reading for an already known vehicle with an existing projection and prediction is received, we update the vehicle projection based upon that data as described in Section 2.5. In the case of predictions, we first calculate the current position we predict the vehicle to be at and whether we expect the vehicle to currently be moving. Then, we check for the following cases:

▷ The vehicle projection changed direction on the track.

(a) Vehicles driving in same direction.

(b) Vehicles driving towards each other.

(c) Vehicle driving towards special node on track.

Figure 2.13. Different types of obstacles for vehicle prediction.

- ▻ The projection indicates the vehicle is moving whereas we predicted the vehicle is not currently driving.
- ▻ The projection indicates the vehicle is idle whereas we predicted the vehicle to currently be moving.
- ▻ The length of the geodesic between the projected and the currently predicted vehicle position is larger than the spatial inaccuracy determined in Section 2.3.

If any of the above cases is met, we discard the calculated prediction, as it was inaccurate and recompute a new prediction based upon the available data using the described steps. We also ensure other vehicles' predictions affected by the outdated prediction are updated as previously described in an effort to ensure all predictions are up-to-date at any given time. Should no condition apply, we consider the prediction accurate and continue using it for estimating the vehicles position.

2.8 Vehicle Visualization with Calculated Projection and Prediction

With all the available information, it is now possible to display both the projection and the prediction for each vehicle. For visualizing the projection of a vehicle, the client needs to receive the projected position, the vehicle heading if available as well as the expected

inaccuracy distance for that location. Using this information, we can visualize the current projected position for the vehicle, the direction along the track utilizing the calculated heading. In order to display this, the vehicle can for example be displayed as an arrow similarly to how most navigational devices show vehicle bearings. Lastly the inaccuracy range for the vehicle projection can be shown by either highlighting the entire range on the track the corresponding vehicle might be on or drawing a circle that includes the entire inaccuracy range on the track for that vehicle. With this, users can easily see how accurate the currently shown position for the vehicle. For the visualization of the continuous prediction, the client must be informed about the geospatial layout of the expected path, the currently progressed distance of the path as well as the speed and determined acceleration-model in order to display the predicted position continuously. Then, the predicted vehicle position can be updated in real-time in an effort to show a steady movement for the corresponding vehicle. From the provided speed and direction along the track, clients can also update and visualize the vehicle heading repeatedly, as the vehicle heading on the tracks will only be influenced by the direction of the rails the vehicle is driving upon. Lastly, clients must be informed in the event that any of the previously calculated data changed due to updates or that vehicles were removed from the simulation due to no available data for a too long time period. With this information, clients are able to interactively show vehicle positions on a map.

Chapter 3

Implementation

In this chapter, the implementation for the described algorithm and a basic client-side visualization for the projections and predictions is described. The implementation follows a standard client-server architecture where the projections of position update and the vehicle movement prediction is calculated server-side and then send to registered clients. Clients then only have to visualize the received information about vehicles on a map and update these as needed in the event of new updates received from the server.

Python[1] was chosen as a primary programming language for the server-side implementation. Other tools in the *REAKT DATA* project are also implemented in Python, which, in addition to the design approach chosen, eases the integration of this solution into the project in the future.

Due to time and complexity reasons, the acceleration and deceleration of vehicles for the prediction based upon historic data was not implemented. Furthermore, the obstacle detection for vehicles as well as some other constraints were not tested thoroughly, which can cause the rendering to not follow all laid out constraints in some cases.

3.1 Overview on the Backend Architecture

A graphical overview for the developed backend architecture is found in Figure 3.1 and can be further described as follows:

- **Position Update Generator**: This component is responsible for providing position updates for all vehicles on the given track. During development, this may be a mock-up, whereas for a production environment this component should provide the position updates for vehicles. The raw, inaccurate vehicle positions are put into the update queue to be handled by the Projection Algorithm.

- **Rail Network Manager**: This component provides and manages the tracks vehicles can drive upon. It is responsible for extracting the information from OSM and proving the information for other components. The primary purpose of the rail network manager is to enable other components to work with extracted network, e. g., another component can receive the nearest point on the track for a given inaccurate point by querying the rail network manager. Furthermore, the rail network manager can be used to store information

[1] https://www.python.org/

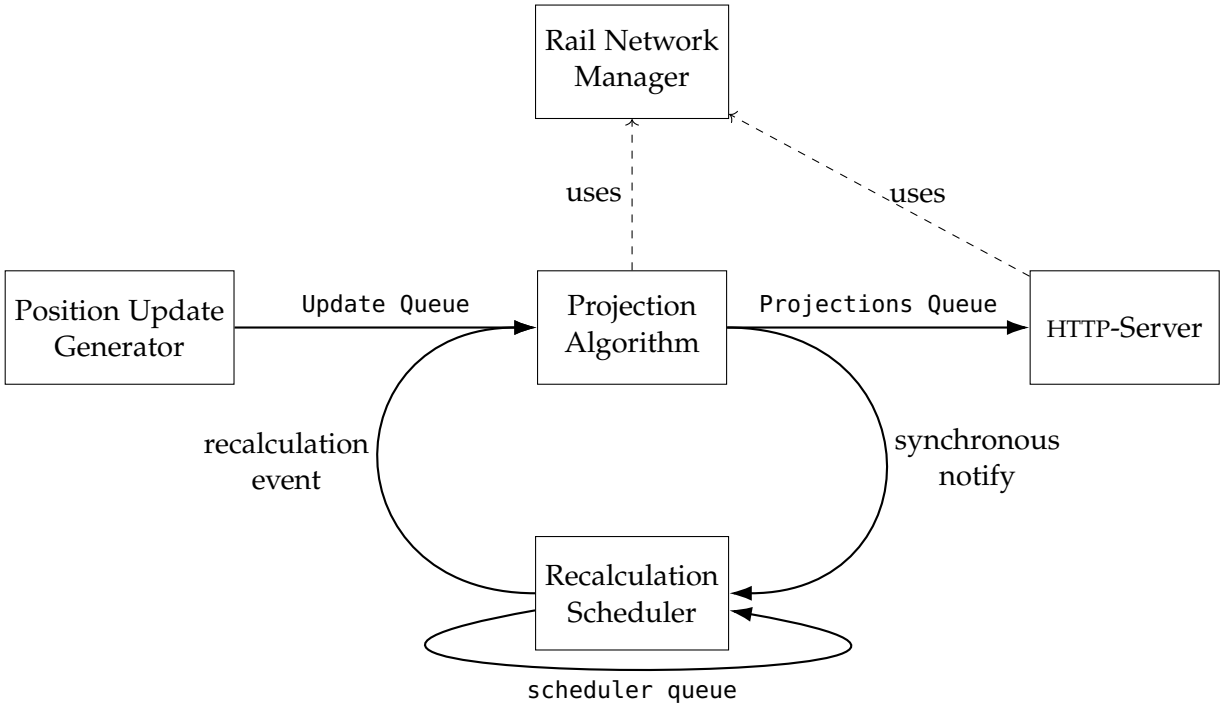


Figure 3.1. Architectural overview of the developed Python implementation.

about on which track segments vehicles are currently on. However, it is not responsible for checking that vehicles are actually ordered correctly on these segments.

- **Projection Algorithm**: This is the main component realizing the implementation of the proposed algorithm. The component handles all passed vehicle updates from the update queue, stores the vehicle states and projects and predicts vehicle movements with the available information. It is notified by vehicle position updates from the position update generator and periodical updates from the scheduler. Then, by using the available information, it calculates new positions and predictions as needed and sends out the new information using the attached queue.

- **Recalculation Scheduler**: The scheduler is only used by the projection algorithm. It ensures the projection algorithm is periodically awakened again to handle special events, for example the removal of a vehicle that was not updated for a long time. The scheduler is directly invoked by the projection algorithm and then schedules itself to notify the projection algorithm after a given interval by firing special events into the main update queue.

- **HTTP-Server**: This is the server serving the calculated projections and predictions for all vehicles as well as geospatial information about the track for clients. It keeps a state of all

updates fired by the projection algorithm for clients connecting at a later time. Position updates are then served using a websocket clients can connect to.

The different components excluding the position generator will be described more in detail in the following sections.

This architectural design comes with the primary advantage that different components can be distributed on different systems, as communication is handled via queues, or that most components can be easily swapped or substituted in the future, since they only rely on the messages sent via the channels and their content. Additionally, through the use of queues, components are active only when they are needed, and remain inactive whenever no event to be handled is in their corresponding queue. One disadvantage is that with unidirectional communication between components, the server for clients has to keep track of vehicle position updates received by the projection algorithm in order to provide all available vehicles to newly connecting clients.

3.2 Network Modelling

As many of the calculations and described steps heavily depend upon the network extracted from OSM, developing methods to access thin information reliable and fast was deemed an important task. This task was primarily split into extracting the network information from OSM and then easy and fast utilization of the methods during program runtime.

3.2.1 Network Extraction

One crucial part is the extraction of the required data from OSM, as the extracted information serves as the basis for all distance measurements and modelling vehicles on the track. However, using all available data from OSM is not feasible, as for a small network, only information regarding these tracks is required. Thus, the available information has to be filtered and then put into a structure where the information can easily be accessed and further used.

For the initial data extraction, OSMnx [Boe24] was used. OSMnx is a tool primarily for analyzing networks and offers various methods for extracting a network graph from OSM. One of its advantages is that OSMnx gathers the data from OSM using the Overpass-Application Programming Interface (API) with no need for any file-downloads required. During the extraction process, filters can be specified that have to be met in order for the information to be returned. Furthermore, one can specify a list of OSM-tags that shall be extracted if available. Finally, OSMnx loads the queried information into a NetworkX MultiGraph with nodes and edges as tuples of two nodes already properly inserted. NetworkX [HSS08] is a graph library for Python frequently used for network modelling. Nodes and edges also already have relevant information such as geospatial coordinates the length of edges linked to them. OSMnx computes the edge length using the haversine formula.

Yet, simply extracting the graph using OSMnx does not work, primarily for two reasons:

- ▻ The Overpass-API does not support logical or-operations for queries. Thus, in the case this is needed, two networks have to be extracted and merged.
- ▻ OSMnx is primarily intended for network analysis, meanwhile the goal for the network modelled here is to provide fast access to geographical and geospatial information and vehicle modelling on the network.

Concluding, the network extracted by OSMnx needs to be further processed before being ready for use in the context of this thesis. This is done as follows. First, we extract (possibly) multiple networks from OSM using OSMNx by providing fitting Overpass-queries. Then, we merge the extracted graphs into one graph based upon nodes that are equal in both graphs. Afterward, all OSM-tags that have unneeded values for specified keys are removed. Additionally, some manually specified OSM nodes that are deemed uninteresting despite otherwise meeting ID have their tags removed. This in preparation for the following graph simplification, in which all nodes with no interesting OSM tags and exactly two edges connected to them are removed from the graph. Connected edges to removed nodes are merged, their lengths are summed, and the exact geographical layout of individual edges is merged into joined line segments consisting of more than two points. These steps are performed for all removed edges. We then store the computed information with the newly merged edges.

Parameters that can be specified for this extraction-process can be found in Table 3.1. With the extracted MultiGraph, we now store the geodesic information for each edge and the corresponding length in a geopandas [JBF+20] data frame. This has the advantage that later, the ordering of nodes in a vehicle path directly translates to directed edges, despite railway tracks being unidirectional by nature, and that this information regarding edges can be queried in a timely manner. Lastly, after storing the geographical information of the track in the data frame, the graph is converted from a Multigraph into a normal, undirected graph because tracks can be driven on in both directions. Thus, modelling each track only as one track helps at later stages in modelling the vehicles consistently on the track.

3.2.2 Querying the Network

The most common task for the network is to assess for given inaccurate update coordinates which point is the closest on the track. As this information is required quite frequently, it should be computed rather quickly.

In order to achieve this, the loaded geographical information from OSM is first converted from the WGS 84 representation into a cartesian 2D projection using the pyproj and PROJ - libraries [ERW+24]. The chosen cartesian projection for this task is EPSG:3035, which is a CRS with high distance accuracy for Europe[2]. This eliminates the need to calculate the geodesics first for determining the closest edges in this implementation. Following the projection into cartesian space, the previously joined edges are converted into Shapely-geometries.

[2] https://epsg.io/3035

Table 3.1. Configuration options for the extraction and modelling of the network.

Field	Description
`bbox`	The bbox in which to acquire the network. Format: (left, bottom, right, top) coordinates in WGS 84 (EPSG:4326).
`custom_filters`	Custom filters for the query against overpass.
`useful_tags_node`	An optional list with additional tags to include for nodes in the graph.
`tags_to_remove`	Tags which are not of value and should be removed before graph simplification.
`node_attrs_include`	List of node attributes to merge the graph on.
`exclude_nodes`	Manual OSM node IDs to exclude during path simplification.
`filename`	The filename to save the collected graph to.
`poi_filter`	Determines points of interest / endpoints for vehicles.
`poi_behavior`	Behavior specifier for approaching nodes.

Shapely [GWV+24] is a Python-library for working with geometries in a cartesian CRS. It utilizes the popular and fast open-source library GEOS [GEO21] for geometrical computations. Furthermore, Shapely can easily determine the nearest point along a given joined line segment for a given point using the built-in `nearest_points`-method, thus, we can already use the merged segments for computing the closed point. Now, we only have to efficiently find the nearest segment for a given point. This is done by putting all track segments into an STR-tree [LLE97], which is a sort of R-tree [Gut84] created by using the Sort-Tile-Recursive algorithm. This tree can not be changed, however, this is not an issue, as we do not modify the track after extraction and simplification. With this tree, we can easily and efficiently find the nearest segment in the network for a given vehicle position after converting the coordinates into the cartesian CRS EPSG:3035 and then determine the nearest point along that segment using Shapely. The found point can then be converted back into WGS 84 using PROJ and the distance of the update to the track is then equal to the haversine distance from the location update to the nearest point on the track with both points in WGS 84.

Determining the progress of vehicles along the projected edge is done by calculating the normalized distance of vehicles along that edge and multiplying this distance with the actual edge length. The calculated distance is then directional starting from one of the two edge nodes, which can be swapped and recalculated in the case that the vehicle is driving the opposite direction along the edge. For calculating the taken path of a vehicle and the predicted vehicle path to end nodes, Dijkstra's algorithm [Dij59] is used to find the path between points and the predicted path to the next interesting node respectively.

Lastly, the network is able to store which vehicles are currently on which edge and can add vehicles, swap vehicle positions and remove them as needed. The network is not responsible for the right ordering of vehicles along the track, as this is the responsibility of the caller to ensure that - the network only provides the basis for caching which vehicle is where and is

unaware which type of vehicles it is storing.

3.3 Algorithm Implementation

The implementation of the algorithm is outsourced to a singular thread which waits for new updates to be sent to it using its associated queue. Created, updated or deleted vehicle predictions and projections are then handled to the server responsible for distributing the updates to connected clients. This separation of concerns enables better maintainability and easier swapping of singular components. The available definable parameters described in Chapter 2 can be found in Table 3.2.

Table 3.2. Configuration Options for Position Projection Algorithm

Configuration Option	Value	Description
`MAX_DEGRADED_DISTANCE`	200	Maximum distance for considering update quality
`STD_DEV_FACTOR`	1.0	Factor to multiply standard deviation for historic evaluation (c_{hist})
`VEHICLE_REMOVAL_INTERVAL`	1200	Time (in seconds) to remove vehicle if no updates received
`NEEDED_DATAPOINTS`	5	Minimal number of results needed for evaluation
`HISTORIC_QUERY_RADIUS`	20	Query radius for historic queries
`SET_MEDIAN_DISTANCE`	10.51	Manually set median for updates with insufficient data
`SET_STD_DEV_DISTANCE`	14.31	Manually set standard deviation for updates with insufficient data

3.3.1 Vehicle Update Processing

On a new update sent via the queue, the update is processed as described in the concept. The update is first projected and assigned a point on the track using the modelled network, then it is compared to previous updates for the vehicle if available and afterward adjacent vehicles are updated accordingly. These updates are then also stored in the network and the update processor ensures the right ordering of vehicles on the network edges. Finally, after the thread computed all changes and saved then, the update processor sends an update into its outgoing pipe to inform the server about the updated vehicle projection and prediction.

3.3.2 Storing and Accessing Historical Data

Historic updates take a core role in evaluating received updates and for the estimation of the future vehicle speed for sections. Therefore, updates have to be stored for later use and

results for spatial querying on data, e. g., for calculating the average deviation around a point, should optimally be available rather fast for good performance. In order to achieve this, PostGIS [3] was chosen as a database, which utilized Postgres [4] as a base database and provides extensions for working with geodesic data out of the box. Additionally, PostGIS supports indexing on geospatial data, which enables fast querying for spatially close points. All this enables us to store the points and needed information easily and additionally query spatially close points, their corresponding average distance/speed and the standard deviation for these values efficiently on the database level, as Postgres supports gathering this information natively.

For easily accessing this information from PostGIS in Python, SQLAlchemy [Bay12] in combination with geoalchemy [BLB+24]. SQLAlchemy is a database object relation mapper and enables us to comfortably store and retrieve the stored historically information from theoretically any database. Geoalchemy is a SQLAlchemy extension and adds support for storing Shapely geometry types as geographic information in the database. These tools in combination enable easy and fast storing and access of previous updates for evaluation and predictions.

3.3.3 Scheduling Vehicle Transitions and Vehicle Removals

As described in Section 2.6, vehicles are to be deleted after a specified amount of time. As the main update thread always waits on new updates to be passed into its queue in order to save resources, it needs to be activated in the case that a vehicle is due for removal. For this purpose, a scheduler was developed which can be directly invoked by the main handler to schedule such updates. The scheduler ensures that after the defined time by the main-handler, the main thread is woken again by passing an update message with the vehicle to remove in the queue of the main update thread. Furthermore, it ensures that only the last scheduled event for each vehicle is considered. The main update handler can also use the scheduler to notify itself about moments where a prediction changes edges in order to register the vehicle on the new, predicted edge.

3.3.4 Vehicle Acceleration Modelling

Modelling vehicle acceleration is a somewhat complicated task. As long as vehicle acceleration is not considered, we can compute the passed distance for a vehicle with a speed reading v between two time readings t_0 and t_1 with the standard formula $s_{passed} = v \cdot \delta_t$, where $\delta_t = t_1 - t_0$ is the passed time between the two time readings. However, once we consider a possible acceleration, the velocity for a given vehicle is no longer constant for the time interval but rather dependent on the time interval t. During that time interval, the vehicle might have accelerated or decelerated, thus the acceleration has to be considered for the speed used in the calculation. Furthermore, the acceleration is also not constant but rather changes over

[3] https://postgis.net

[4] https://postgres.org

time as well, as the vehicle will not be accelerating or decelerating indefinitely, but only to a certain speed and continue to drive at that given speed. Concluding, both acceleration and speed need to be modelled as functions for accurately representing the vehicle movement.

We model the acceleration as follows:

$$a(t) = \begin{cases} x & \text{if the vehicle shall be accelerated by } x\frac{m}{s^2} \text{ at the given time} \\ 0 & \text{otherwise} \end{cases}$$

and thus the velocity can be computed via

$$v(\delta_t) = v(t_0) + \int_{t_0}^{t_1} a(t)dt$$

Consequently, we do not model the acceleration as a continuous function, but rather let it jump between values - whilst this does not truly reflect the real change of velocity for the vehicle, this vastly eases follow-up calculations. Furthermore, the acceleration behavior is still a guess at best, thus the slight inaccuracy introduced by this simplification was deemed negligible.

With this, we can now model the acceleration behavior based on these assumptions as a list of tuples (a, t) where a is the acceleration for the vehicle and t is a time interval over which the acceleration will be applied to the vehicle. The value for a can be either positive or negative, which allows for acceleration or deceleration as needed. Once a vehicle has been accelerated by a for the time t, we move to the next tuple in the list or, should the list be empty, just continue using the last calculated speed for calculating the vehicle's distance as needed. For example, should a vehicle's acceleration be described by the list $[(2, 10), (0, 20), (-3, 10)]$, we would first accelerate the vehicle by $2\frac{m}{s^2}$ for 10 seconds, then continue with the last calculated speed for 20 seconds and then decelerate the vehicle for 10 seconds by $-3\frac{m}{s^2}$. Afterward, the vehicle would be expected to continue driving with the last calculated speed.

This enables us to model acceleration for the vehicle with approximating functions and also use this model to check how vehicles would behave when meeting other vehicles on the track. For example, we can calculate when two opposing vehicles would meet on a given track segment given their current acceleration model. Sample code which computes when two vehicles will meet each other using this model of acceleration can be found in the appendix in Listing A.1. As seen in the listing, the calculations are rather complex, as for both vehicles, we have to determine the point in the future where, according to the model, they are expected to meet. For this, we have to accelerate both vehicles according to their models and check whether during the current acceleration they would have seemingly collided. Once the interval is found, we have to calculate the exact point in time when they would collide. The vehicle acceleration behavior can then be refined and recalculated based on the found point in time when the vehicles would intersect. It can also be determined whether two vehicles would not meet, in which case the models would not be updated.

Yet, acceleration modelling remains a complex task and thus was also not fully implemented in the approach. Since all calculations of progress for the vehicle now heavily depend on the time frame, as the correlation between speed and time is not linear anymore, a lot

more effort is required to approximate acceleration behaviors. Nevertheless, some aspects make use of the acceleration, e. g., stopping before interesting nodes, acceleration behavior can be defined and calculated for vehicles and also be accurately visualized for users on the client-side map.

3.3.5 HTTP- and Websocket Server

In order to make the generated projections available for multiple clients, a server is needed. This server is build with FastAPI [Ram] and distributes the received updates from the algorithm to registered clients using a websocket connection for sending the updates. It sends a `NEW` - message in the case of a newly registered update with the projection- and prediction-data attached, an `UPDATE` - message in case that a projection and/or prediction has changed with the information on what has changed and, lastly, in the event that a vehicle was removed from the prediction, a `DELETE` - message with means to identify the vehicle slated for removal. Furthermore, it keeps a record of all currently available vehicle projections and predictions for clients connecting at a later time. Then, in the event of a new client connecting, it sends them an initial `STATE` - message, containing a map of all currently known vehicles, where their identifier (ID) is the key and the value is their current projection- and prediction-data. This ensures all connected clients have access to the same information regardless of their time of connecting.

Finally, the server is responsible for translating the edges of the vehicle's expected path prediction from the internal representation to something clients can get over the HTTP. This is needed due to the fact that we work differently with the concept of direction on the side of the server compared to clients: Whilst the server can differentiate the direction of vehicles, clients instead just naively display and update the progress of a vehicle along a track, thus they need already directed edges for the visualization. Additionally, this translation layer enables us to arbitrarily change the network without the need to change client code, as clients dynamically fetch information about the geographical network layout from the server as needed, removing the need to ship the entire geospatial network layout to clients.

3.4 Client-Side Visualization

With vehicle positions and predictions now available, the last task for the implementation is to visualize this information appropriately on a map for the user in their browser. This task can primarily be split into three larger issues: Handling all incoming messages from the server, setting the vehicles expected path upon a newly received update and continuously re-rendering predicted vehicle positions on the map. Once these are solved, the user is able to see the lastly projections positions of all vehicles and the predictions for their current position on the map.

3.4.1 Used Technologies

For the implementation of the frontend visualization, TypeScript[5] was chosen as a programming language. Map rendering is performed by MapLibre GL JS [6], which has the advantage that it uses WebGL for map and other layer rendering. For working with geospatial data in TypeScript, the `GeoJSON` - standard [BDD+16] was used. Lastly, turf [7] is used to work with geospatial objects in TypeScript.

3.4.2 Vehicle Track Acquisition

As described in Section 3.3.5, the server does not send all track coordinates with each update, but rather a list of ordered IDs of edges to construct the vehicle path from. Thus, before the client can visualize the vehicle driving along a path, it has to construct the path from the available information.

This is done by requesting the raw coordinates for each edge by their ID from the servers API, concatenating their coordinates into one list, and finally creating a line string from the accumulated coordinates. As the server provides the track IDs and corresponding coordinates in order, no further logic is required for this client-side: A simple concatenation is sufficient. Furthermore, track coordinates for track parts and created line-strings are cached for future use, so the client only has to perform minimal requests. The created line string then replaces the raw track IDs for the vehicle and provides a path the vehicle can be animated along.

3.4.3 Server Message Handling

Before any vehicles can be rendered, the data has to be first queried from the server. Thus, the first step for clients is to connect to servers using the aforementioned websocket. Once clients are connected to the server, they will handle the four messages as follows.

- **`STATE` - message**: Upon receiving this message, all vehicle projections and their predictions are added to the visualization, the predictions are progressed from the time they were initially registered to the current time, and the animation for predicting vehicle movements is started.
- **`NEW` - message**: Upon receiving this message, add the new prediction and projection to the map.
- **`UPDATE` - message**: Upon receiving this message, replace the current projection with the new projection and update the prediction.
- **`DELETE` - message**: Upon receiving this message, delete the corresponding vehicle projection and prediction from the visualization.

[5] `https://www.typescriptlang.org/`
[6] `https://maplibre.org`
[7] `https://turfjs.org`

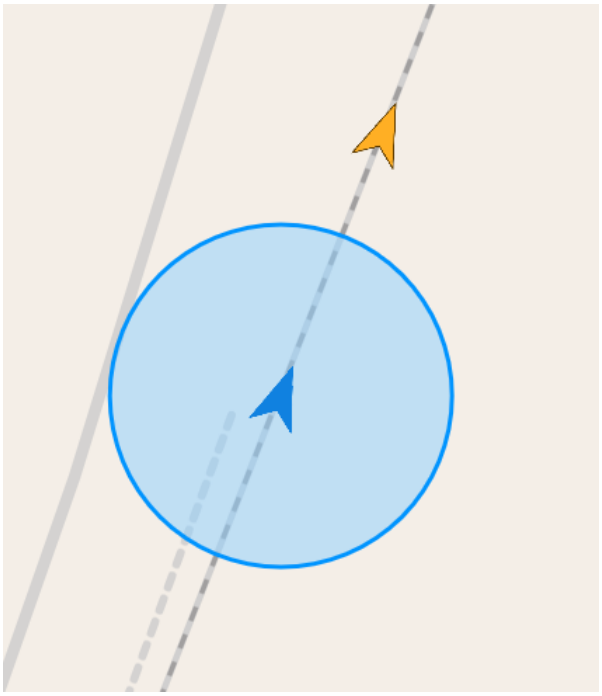


Figure 3.2. Visualization of vehicle projections and predictions in the developed implementation.

Additionally, we back up the initial speed of the vehicle and duplicate the time information for each acceleration interval. This information is later used for elimination of floating point errors during the vehicle prediction rendering. With the messages now handled, the last task that remains is to visualize the vehicles projections and predictions on the map.

3.4.4 Vehicle Rendering

For rendering the vehicles projections and predictions, we first add a layer to the map that can display this information. This layer requires a data source from which it can render objects. We use a `GeoJSON` data source, as `GeoJSON` is the preferred way for working with geospatial data in JavaScript. Furthermore, we ensure that projections and predictions are rendered differently, so users can easily distinguish between the two.

Displaying the projections is a rather simple task: As the coordinates for the projection are already given by the server, we just add these to the `GeoJSON` source and highlight that these are the projections. Furthermore, we display the possible inaccuracy for the projection by drawing a circle which covers the entire span of the inaccuracy along the track for that projection. A circle was chosen as this was deemed to be easier to see rather than highlighting the track. The projected position as well as the circular highlighted inaccuracy is highlighted as blue in Figure 3.2.

Whereas the visualization of the projections only needs to be computed once, the vehicle positions for the predictions have to be recalculated every frame, as the goal is to visualize

these fluently for clients on their devices. In order to update vehicles every frame for the user, we use the native browser API `requestAnimationFrame` to call the method for updating all vehicle positions, which calls the given method every frame the browser re-renders the window. The API passes the provided method the progressed time in milliseconds as an argument, which we can then use to progress the vehicle predictions for the given time interval. The code for progressing the vehicle positions can be found in Listing 3.1. For the given time interval, we first check whether the vehicle needs to be accelerated or decelerated given the predicted acceleration information. This is directly checked at the start. Should that be the case, we additionally check whether the predicted acceleration would stop during the given time interval. In that case, we update the vehicle acceleration, the corresponding speed as well as the progressed position according to the remaining time for this acceleration interval and then proceed with the next acceleration model if available. Otherwise, should the vehicle continue to be accelerated at a constant value over the entire given time interval, we register that we accelerated the vehicle according to that information and determine the new speed and passed distance with that information. As we do not model the acceleration as continuous but rather constant for given intervals, the following calculation for the speed determination after the given interval can be done linearly. The progressed distance is then updated based upon the speed change over the interval and the length of that interval. In the event that the vehicle is not to be accelerated or that the given time interval was longer than the velocity model of the vehicle, we instead calculate for the vehicle how many meters it has progressed since the last update and add this to the position. Finally, we convert the progress of the vehicle along its path into coordinates for showing the vehicle position on the map using turf's `along` method, calculate a bearing from the direction of the vehicle along the track and add the calculated position and heading to the `GeoJSON` source. This is done for every known vehicle for each frame in an effort to produce a continuous visualization. Constraints such as vehicles stopping in front of each other are expected to have been provided by the backend through the models for the vehicles' accelerations. The resulting visualization for this approach can also be seen in Figure 3.2 where the projected position is highlighted in orange.

Listing 3.1. TypeScript-code used to calculate the vehicle's position after driving for a given time interval.

```
let timeToProgress = intervalToProgressVehiclesForInSeconds();
// check if the vehicle shall be (de-)accelerated
while (vehicle.acceleration?.length > 0) {
    // get info about the acceleration. This is a list
    // [accelerationInMs, remainingTimeToAccelerate, totalAccelerationTime]
    const acc = vehicle.acceleration[0];
    // reduce the time for this acceleration interval
    acc[1] -= timeToProgress;
    // if the time exceeded the remaining time to accelerate, apply
    // acceleration only for this interval
    if (acc[1] < 0) {
        // accelerate and move the vehicle for the given interval
        const prevSpeed = vehicle.speed;
        vehicle.prevSpeed = vehicle.speed = vehicle.prevSpeed + acc[0] * acc[2];
        vehicle.position += ((vehicle.speed + prevSpeed) / 2) * (acc[1] + timeToProgress);
        // remove the progressed time from the time to progress
        timeToProgress = -acc[1];
        // delete the information for the finished acceleration and move
        // to the possible next acceleration
        vehicle.acceleration.shift();
        continue;
    }
    // if the remaining time to accelerate the vehicle exceeds the time
    // to progress, just accelerate and move the interval for the given time interval
    const prevSpeed = vehicle.speed;
    vehicle.speed += acc[0] * timeToProgress;
    vehicle.position += ((prevSpeed + vehicle.speed) / 2) * timeToProgress;
    break;
}
// if no vehicle acceleration information is available (anymore),
// continue to move vehicle with constant speed
if (vehicle.acceleration?.length == 0) {
    vehicle.position += vehicle.speed * timeToProgress;
}
// calculate the progress of the vehicle along its expected path
const newPos = along(vehicle.path, vehicle.position, {
    units: "meters",
});
```

Chapter 4

Evaluation

In this chapter, I perform a short analysis of the developed algorithm and implementation with data collected from the *REAKT DATA* project.

4.1 Analysis of Available Data

For the analysis of the data and algorithm, a dataset from *REAKT DATA* containing 66149 data points was used. These data points were collected from the GNSS sensors which the trolleys currently driving on the *REAKT* track are equipped with. From these points, 466 points were initially manually cleansed, as these had a distance of more than a thousand meters to the nearest point of the track and were found to not belong to vehicle data acquired from the track, but rather tests of sensors before these were attached to vehicles. This left a total of 65683 points for further analysis.

After bucketing the remaining points depending on their distance to the track as seen in Table 4.1 and Figure 4.1, it was decided to set the max distance to the track $d_{\max}$ to 200 m. This excluded 107 more updates from the evaluation, which is however less than 0.2% of available updates. While the distribution of raw update inaccuracies as shown in Figure 4.1 seemingly does not fit the expectation previously shown in Figure 2.4, this might again be in parts due to the fact that we are currently looking at the one-dimensional error for updates despite this error being two-dimensional by error. However, this cannot be validated with the currently available data, as the exact error for these updates remains unknown and thus

Table 4.1. Table of distances to track and corresponding signal count.

Distance to track (m)	Signal count
0-100	65261
100-200	315
200-300	61
300-400	21
400-500	13
500-600	4
600-700	2
700-800	1
800-900	4
900-1000	1

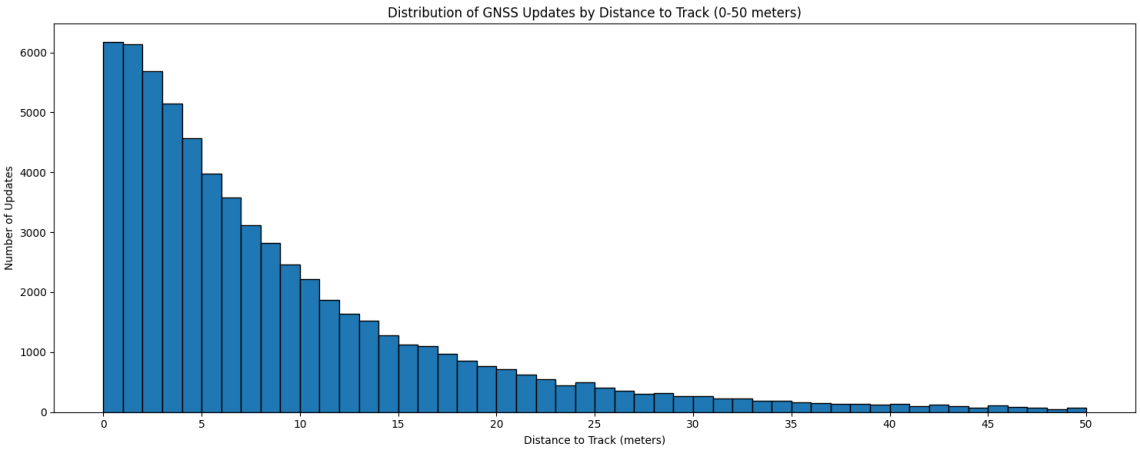


Figure 4.1. Amount of signals within distances to track.

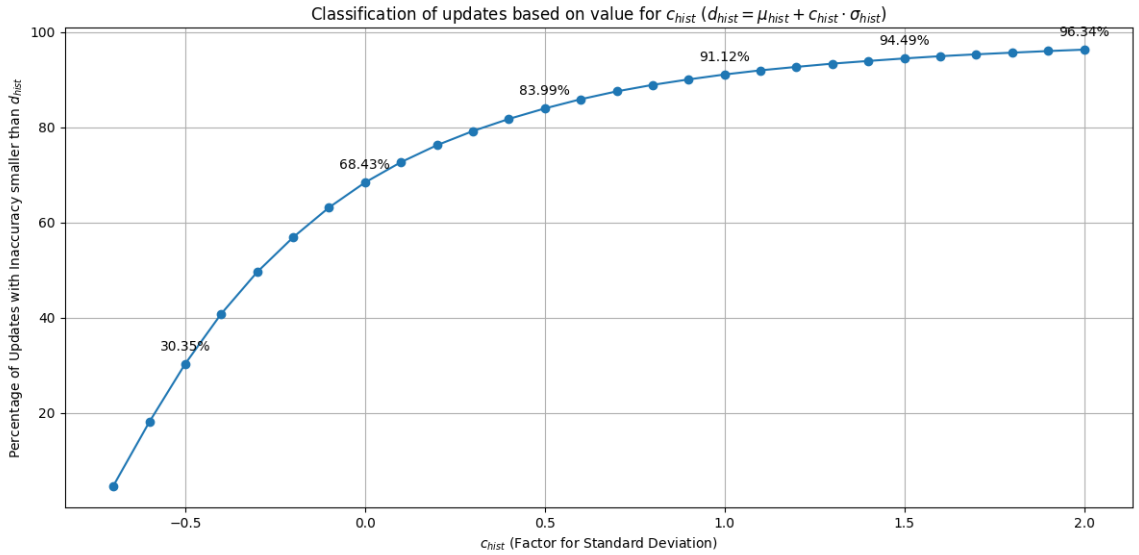


Figure 4.2. Influence of c_{hist} on update inaccuracy evaluation.

should be evaluated further at a later point where an exact point of reference for inaccurate updates is known.

For the remaining data, the global mean distance to the track and standard deviation of that distance was calculated: The global mean inaccuracy for the updates is $\mu_{hist} = 10.51m$ and the standard deviation of the inaccuracy is $\sigma_{hist} = 14.31m$.

Given these readings, an appropriate c_{hist} was determined by checking its influence on update classification as shown in Figure 4.2. This plot shows how many updates' δ_{acc} would be increased to consider an inaccuracy based upon historical data rather than using the calculated, one-dimensional distance to the track for this update. Based upon this data, it

was decided to settle for a $c_{hist} = 1.0$. This might seem like a large value, given that for this value, more than 90% of updates would have an increase in inaccuracy. However, as we are currently looking at μ_{hist} and σ_{hist} in a global scope, the real percentage once looking at a local scope is expected to be far smaller. Considering that the global inaccuracy is especially vulnerable to larger outliers, as all outliers are considered for the mean here, the expected influence on local calculations is expected to be far smaller. Furthermore, δ_{acc} is primarily used to enforce track constraints, check whether vehicles moved between updates as well as to communicate the updates' inaccuracy to users. In this context, having slightly larger values might even benefit the functionality of the algorithm overall.

Table 4.2. Hourly distribution of intervals between received updates for vehicles.

Time Interval	Count	Percentage	Percentage with 12h+ removed
0-1 hours	63006	96.11%	98.83%
1-2 hours	337	0.51%	0.53%
2-3 hours	168	0.26%	0.26%
3-4 hours	94	0.14%	0.15%
4-5 hours	55	0.08%	0.09%
5-6 hours	29	0.04%	0.05%
6-7 hours	20	0.03%	0.03%
7-8 hours	12	0.02%	0.02%
8-9 hours	3	0.00%	0.00%
9-10 hours	3	0.00%	0.00%
10-11 hours	8	0.01%	0.01%
11-12 hours	20	0.03%	0.03%
12+ hours	1804	2.75%	-

Lastly, it was checked with what frequency updates were on average received for all vehicles in order to determine when to remove vehicles from the simulation. For this, it was checked how much time passed in between updates. The initial result is shown in Table 4.2. However, due to the large amount updates with more than 12 hours between them, the resulting mean of time between updates was $\mu \approx 84min$ with a standard deviation of $\sigma \approx 10h$.

Following this result, another analysis was performed by excluding all updates with time gaps of more than 12 hours. A time that large between updates causes no accurate prediction to be possible whatsoever. The resulting mean and standard deviation were much smaller: This showed that on average, every 4 minutes a new update was received with a standard deviation of $\sigma \approx 6min$. Figure 4.3 shows that the majority of updates is received three to five minutes after the previous update. Utilizing this information as well as the cumulative percentages of the intervals as shown in Table 4.3, it was decided to set the time $t_{\max}$ after which vehicles are deleted to $t_{\max} = 20min$. This should cause vehicles to rarely be removed from the simulation and only in cases where the GNSS sender stopped working for a longer time interval. For these cases, an accurate projection is never possible. Yet, as just outlined,

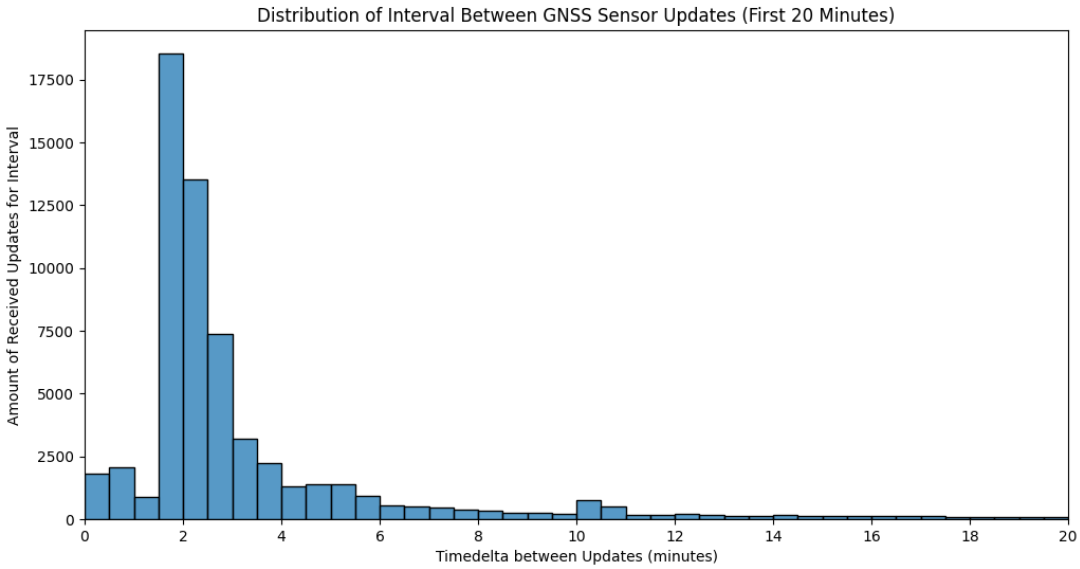


Figure 4.3. Amount of consecutive updates received within first twenty minutes.

the removal of vehicles will rarely occur and only in cases where it makes sense to remove the vehicle due to no data being available.

Table 4.3. Cumulative percentages for amount of updates received in first hour.

Time Interval (minutes)	Percentage of Updates	Cumulative Percentage
0-10	91.47%	91.47%
10-20	5.71%	97.18%
20-30	1.55%	98.73%
30-40	0.66%	99.39%
40-50	0.39%	99.77%
50-60	0.23%	100.00%

4.2 Preliminary Algorithm Evaluation

With these parameters, a preliminary analysis of the described approach could be performed.

As we set c_{hist} based on global, it was first checked how this would influence the calculated δ_{acc} upon local querying. For this, it was found that for 90% of updates, δ_{acc} is set based upon the historical data with an average increase of 16.10 m for update compared to their one-dimensional reading. Overall, in the cases where δ_{acc} was set based upon historical data, it

had a size of 24.5 m, so the size roughly tripled on average compared to the one-dimensional reading. Concluding, a smaller c_{hist} might provide better results in future applications.

Afterward, the accuracy for the pushing of vehicles as well as the prediction of vehicle positions was determined. Due to the fact that some parts of the predictions, especially the historic data usage, and some applications of track constrains are missing, this analysis was performed in one step, as the pushing of vehicles and the prediction of vehicle positions behave similar for that matter. For classifying whether a prediction or projection was accurate, it was checked whether the vehicle prediction was within the inaccuracy radius of a newly received update for that vehicle.

Utilizing the available data, it was found that on average, the prediction and the next received update had a distance of roughly 323 meters between them. Furthermore, it was found that roughly 42% of vehicles were predicted to be at the position their next update was received at. For these cases, the prediction on average calculated the vehicle position accurately for two minutes and 45 seconds before the next update validated the predicted movement. Furthermore, the standard deviation for the time between updates was $\sigma \approx 126s$, indicating that for some cases, the predicted position was accurately predicted much longer. However, within this percentage are many readings included where vehicles were not moving at all. By excluding remaining vehicles, still around 16% of predictions were accurate once the next update was received and the next update was received on average two and a half minutes after the update the approximation for the vehicle was based upon. The standard deviation of $\sigma \approx 109s$ indicates that also for this case, some movement predictions were accurately calculated for longer time intervals. Whilst this seems to show that the movement prediction is rather unreliable, this primarily applies to the approximation in its current state: Due to time reasons, historic data is currently not considered for the vehicle acceleration approximation and obstacle detection for the vehicle path is not properly calculated for all cases. This is clearly visible once looking at the average distance between assumed positions and updated positions, which is $\mu \approx 747m$ with a mean interval between updates of 9 minutes. We see here that due to the missing constraints, some calculations are far off from the reality on the track. Concluding, the algorithm might yield far better results for an implementation that honors the constraints laid out in Chapter 2.

The analysis performed here is not complete and can be extended in various ways. However, due to time reasons and many parts of the algorithm missing in the implementation. These aspects were not evaluated at all. A better and more thorough evaluation in the future where more of the described features are tested would be needed. This could help at determining issues with the described method in order to eliminate and solve false assumptions made about vehicles or updates. However, with the data and analysis available, it seems that the algorithm could help in approximating locations based upon sparse data.

Chapter 5

Conclusion

This thesis proposed a new method to produce a best-effort projection and prediction for irregularly received inaccurate GNSS updates from vehicles without timetables on a given track. First, to eliminate the GNSS inaccuracies, the inaccurate position updates are placed on the specified track which was previously extracted from OpenStreetMap. Based upon the calculated distance from the update position to the track and its inaccuracy in comparison to historically received points in the update area, the obtained signal is then evaluated and further processed. For the purpose of preventing a change of order of projected vehicles, the algorithm then updates any adjacent vehicle projections based upon the update evaluation, the vehicle headings and estimated speed as needed. In an effort to provide a live projection of the vehicle, the algorithm then predicts the velocity and acceleration for each updated vehicle, estimating the acceleration based upon historical data and user configuration. Finally, the resulting projections for updated vehicles are then made available for clients, which can then visualize the vehicles on any given map continuously with minimal effort. The evaluation shows that the algorithm projection and prediction quality varies from accurate to unreliable results depending on the tested scenarios. However, this might be primarily due to limitations of the current implementation, as some elements of the concept were not implemented, and can thus be improved in the future.

Since many parts of the described algorithm can be customized or extended, the algorithm is easily adaptable towards different situations and applications.

5.1 Future Work

Due to the fact that the method was primarily tested with a single track with few vehicles in mind, there remains much room for improvements, especially in the context of larger and more complex networks.

5.1.1 Improve Point Inaccuracy Improvement

The inaccuracy of received update points is two-dimensional in nature, yet can only be calculated in one-dimensional space. There is currently no way to know how far the projection onto the track is off compared to the actual vehicle position on the track. While this issue is taken care of by utilizing the historic inaccuracy of position updates in the projected area and increasing the inaccuracy for most updates, as most points will be equally inaccurate away from the track as along the track on average, there remains room for improvement.

For example, small calculated inaccuracies could be taken more into account in areas where points are generally more inaccurate, as is it unlikely that suddenly an overly accurate update is received in such an area. Furthermore, point clustering algorithms could be considered for the retrieval of historic inaccuracies and comparison to these, as this would improve the reference value and exclude too large outliers to be considered in the historic evaluation. The improved inaccuracy detection could then improve the position projection.

5.1.2 Better Use of Historic Data for Singular Vehicles

In the described approach, primarily historic information for all vehicles is considered. However, for a given vehicle, more information might be extracted from its previous updates, even in cases where $t_{\max}$ was exceeded, and the vehicle removed from the model. This might be e. g., how fast a singular vehicle moves at maximum speed based on historic readings or a better approximation of the vehicle speed compared to historic speed for segment by utilizing past speed information of the vehicle's current trip. This could further enhance the projection as well as the prediction for vehicles.

5.1.3 Implementation Support for Complex Vehicle Tracks

While the general method is also suited for complex tracks, the implementation was primarily tested and developed against a single track network. Thus, some considerations for multi-track networks were neither considered nor implemented, as they could not be tested with the present track. This primarily includes:

- **Vehicle Path Prediction**: The current approach to determine the vehicle's path for the near future as described in Section 2.7.1 is rather naive and will in its current state not work well for more complex networks. Improvements in determining the most likely next endpoint would vastly improve the prediction quality, however this might require more information about vehicles to be available. Furthermore, the currently implemented method to calculate the expected vehicle path on the tracks is rather expensive and falls short for multiple possible endpoints for vehicles. For each change of vehicle direction, Dijkstra's algorithm is used to find the new distant vehicle endpoint based upon the vehicle heading and furthest distance from the vehicle. This however might lead to inaccurate assumptions, as the next most distance endpoint in the current endpoint might not always be the actual endpoint. Thus, some changes for a more accurate prediction might help with performance and accuracy.

- **Nearest Vehicle Path Segment**: In the case of multiple and spatially close tracks, issues might arise while query the current expected track segment for each vehicle:

  1. The spatially closest track segment is not the actual track segment the vehicle is currently driving on, thus at least one more spatially close track segment would have to be queried.

2. The spatially closest track segment is the actual track segment the vehicle is currently driving on, but it is not assumed based upon the predicted path of the vehicle.

These cases would both need to be considered for an accurate vehicle projection and prediction, yet it is hard to make an accurate assumption with only the vehicle position updates available. In some cases, this might to many erroneously made assumptions. Concluding, both issues would most likely require more information for each vehicle to be available.

5.1.4 Filtering of Frequent Updates

The developed algorithm was designed under the assumption that updates are only received every few minutes for each vehicle. Thus, in the scenario that the frequency of updates is much higher and many vehicles are registered, performance issues might arise due to the amount of needed computations. While the received data points are still interesting for the historic information, some updates should probably not be used for further computations in the algorithm. Furthermore, for many available data points, better caching for queries in the database should also be considered in order to improve performance.

5.1.5 Using R-Tree with Geodetic Distance for Nearest Track Querying

In the current implementation, the spatially closest edge is determined by querying an STR-tree with all edges projected into a Cartesian coordinate system with the update point projected into the same system. This might lead to issues, as in the flat projection distance relations can become distorted, resulting in possibly inaccurate distance calculations in Cartesian space and wrong closest edges being returned.

R-trees have peen proven to also work for geodetic distances [SZK13]. Using an R-tree suited for geodetic distance queries would always yield accurate results and eliminate the uncertainty of possibly inaccurate results.

5.1.6 Continuous Acceleration Modelling

In the current implementation, the acceleration functions used for position projection are not continuous - the acceleration jumps between intervals. For slightly more realistic acceleration or deceleration predictions, it might be worth to model the acceleration as continuous and change the implementation.

5.1.7 Visualization Improvements

The client-side visualization in its current state is mostly a proof-of-concept and can be improved in various places. This includes but is not limited to:

- ▻ Visualizing the age for all vehicle projections and predictions in an appealing way.
- ▻ Updating vehicle positions only in the currently displayed bounding box of the map.

- ▻ Limiting the frequency of redrawing vehicles based upon map zoom state and amount of displayed vehicles.
- ▻ Adding a time slider for fast-forwarding or rewinding the prediction state.

These improvements would increase accessibility and usability for users and can be implemented with changing close to none of the backend-algorithm.

Appendix A

Code to Determine Intersect Point of Vehicles

Listing A.1. Code used to compute when two vehicles would theoretically collide given their current distance, speed, acceleration information, and the length of the edge they are on.

```
    def compute_opposing_intersection(
        self,
        v1_current_speed: float,
        v1_acceleration: AccelerationData,
        v1_distance_along: float,
        v2_current_speed: float,
        v2_acceleration: AccelerationData,
        v2_distance_along: float,
        edge_length: float,
    ) -> tuple[TimeSeconds, float, float] | None:
        """
        Compute the intersection point of two opposing vehicles on an edge.

        This method calculates the time and position where two vehicles traveling
        in opposite directions on the same edge will meet, considering their
        current speeds, accelerations, and positions along the edge.

        AccelerationData is modelled as a list of tuples, where each tuple contains
        two elements: (acceleration_value, duration). The acceleration_value is in
        m/s^2, and the duration is in seconds. This allows for representing complex
        acceleration profiles with multiple phases of acceleration or deceleration.

        Args:
            v1_current_speed (float): Current speed of vehicle 1 in m/s.
            v1_acceleration (AccelerationData): Acceleration profile of vehicle 1.
            v1_distance_along (float): Distance of vehicle 1 along the edge in meters.
            v2_current_speed (float): Current speed of vehicle 2 in m/s.
            v2_acceleration (AccelerationData): Acceleration profile of vehicle 2.
            v2_distance_along (float): Distance of vehicle 2 along the edge in meters.
            edge_length (float): Total length of the edge in meters.

        Returns:
            tuple[TimeSeconds, float, float] | None: A tuple containing the time of
            intersection and the distances of both vehicles at the intersection point,
            or None if no intersection occurs.
        """
```

```
        # safety check for any interception
        if v1_distance_along + v2_distance_along > edge_length:
            return None
        # check whether vehicles can keep a distance
        if v1_distance_along + v2_distance_along > edge_length - self.vehicle_gap:
            # if they can, calculate with that gap
            edge_length -= self.vehicle_gap
        # start calculation
        sum_time = 0
        v1_i, v2_i = 0, 0
        v1_t_offset, v2_t_offset = 0, 0
        while True:
            # the latter time value is a more or less arbitraty high value
            # in case no more acceleration information is available
            v1_acc = (
                v1_acceleration[v1_i]
                if v1_i < len(v1_acceleration)
                else (0, edge_length / v1_current_speed)
            )
            v2_acc = (
                v2_acceleration[v2_i]
                if v2_i < len(v2_acceleration)
                else (0, edge_length / v2_current_speed)
            )
            v1_t, v2_t = v1_acc[1] - v1_t_offset, v2_acc[1] - v2_t_offset
            t = min(v1_t, v2_t)
            v1_speed_delta, v2_speed_delta = v1_acc[0] * t, v2_acc[0] * t
            v1_passed_dist = (v1_current_speed + v1_speed_delta / 2) * t
            v2_passed_dist = (v2_current_speed + v2_speed_delta / 2) * t

            if v1_passed_dist + v1_distance_along >= edge_length - (
                v2_passed_dist + v2_distance_along
            ):
                # intersection found in this interval
                # calculate the exact time of intersection
                # solve with quadratic formula
                # as it can be said:
                # v1_current_speed * t + 0.5 * v1_acc * t^2 + v2_current_speed * t + 0.5 *
                    v2_acc * t^2 = edge_length
                a = (v1_acc[0] + v2_acc[0]) / 2
                # in case vehicles are not accelerating any more, calculate linearly
                # as a != 0 for quadratic formula needed
                if a == 0:
                    intersect_time = (
                        edge_length - v1_distance_along - v2_distance_along
                    ) / (v1_current_speed + v2_current_speed)
                    return (
```

```
                sum_time + intersect_time,
                v1_passed_dist + v1_current_speed * intersect_time,
                v2_passed_dist + v2_current_speed * intersect_time,
            )
        # otherwise, continue with formula
        b = v1_current_speed + v2_current_speed
        c = -(edge_length - v1_distance_along - v2_distance_along)
        intersect_time = max(
            (-b + factor * sqrt(b**2 - 4 * a * c)) / (2 * a)
            for factor in (-1, 1)
        )
        return (
            sum_time + intersect_time,
            v1_passed_dist + v1_current_speed * intersect_time,
            v2_passed_dist + v2_current_speed * intersect_time,
        )

    # update passed dists and speeds
    v1_distance_along += v1_passed_dist
    v2_distance_along += v2_passed_dist
    v1_current_speed += v1_speed_delta
    v2_current_speed += v2_speed_delta
    if t == v1_t:
        v1_t_offset = 0
        v1_i += 1
        v2_t_offset += v2_t - t
    if t == v2_t:
        v2_t_offset = 0
        v2_i += 1
        # this will be 0 in case of v1_t == v2_t
        v1_t_offset += v1_t - t
    sum_time += t
```

Bibliography

[Bay12] Michael Bayer. "SQLAlchemy". In: *The Architecture of Open Source Applications Volume II: Structure, Scale, and a Few More Fearless Hacks*. Ed. by Amy Brown and Greg Wilson. aosabook.org, 2012. URL: http://aosabook.org/en/sqlalchemy.html.

[BBS14] Hannah Bast, Patrick Brosi, and Sabine Storandt. "Real-Time Movement Visualization of Public Transit Data". In: *Proceedings of the 22nd ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*. SIGSPATIAL '14. Dallas, Texas: Association for Computing Machinery, 2014, pp. 331–340. ISBN: 9781450331319. DOI: 10.1145/2666310.2666404. URL: https://doi.org/10.1145/2666310.2666404.

[BDD+16] H. Butler, M. Daly, A. Doyle, Sean Gillies, T. Schaub, and Stefan Hagen. *The GeoJSON Format*. RFC 7946. Aug. 2016. DOI: 10.17487/RFC7946. URL: https://www.rfc-editor.org/info/rfc7946.

[BLB+24] Adrien Berchet et al. *Geoalchemy/geoalchemy2: 0.15.2*. Version 0.15.2. July 2024. DOI: 10.5281/zenodo.12707773. URL: https://doi.org/10.5281/zenodo.12707773.

[BMW15] Candace Brakewood, Gregory S. Macfarlane, and Kari Watkins. "The impact of real-time information on bus ridership in New York City". In: *Transportation Research Part C: Emerging Technologies* 53 (2015), pp. 59–75. ISSN: 0968-090X. DOI: https://doi.org/10.1016/j.trc.2015.01.021. URL: https://www.sciencedirect.com/science/article/pii/S0968090X15000297.

[Boe24] Geoff Boeing. *Modeling and Analyzing Urban Networks and Amenities with OSMnx*. Working paper. 2024. URL: https://geoffboeing.com/publications/osmnx-paper/.

[CO07] Brian Caulfield and Margaret O'Mahony. "An Examination of the Public Transport Information Requirements of Users". In: *IEEE Transactions on Intelligent Transportation Systems* 8.1 (2007), pp. 21–30. DOI: 10.1109/TITS.2006.888620.

[Dij59] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[ERW+24] Gerald I. Evenden et al. *PROJ*. Version 9.4.0. Mar. 2024. DOI: 10.5281/zenodo.5884394. URL: https://github.com/OSGeo/PROJ/.

[GEO21] GEOS contributors. *GEOS computational geometry library*. Open Source Geospatial Foundation. 2021. DOI: 10.5281/zenodo.11396894. URL: https://libgeos.org/.

[Gut84] Antonin Guttman. "R-trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*. 1984, pp. 47–57.

[GWV+24] Sean Gillies, Casper van der Wel, Joris Van den Bossche, Mike W. Taves, Joshua Arnott, Brendan C. Ward, et al. *Shapely*. Version 2.0.6. Aug. 2024. DOI: 10.5281/zenodo.5597138. URL: https://github.com/shapely/shapely.

[HSS08] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. "Exploring Network Structure, Dynamics, and Function using NetworkX". In: *Proceedings of the 7th Python in Science Conference*. Ed. by Gaël Varoquaux, Travis Vaught, and Jarrod Millman. Pasadena, CA USA, 2008, pp. 11–15.

[HW08] Mordechai Haklay and Patrick Weber. "OpenStreetMap: User-Generated Street Maps". In: *IEEE Pervasive computing* 7.4 (2008), pp. 12–18.

[JBF+20] Kelsey Jordahl et al. *Geopandas/geopandas: v0.8.1*. Version v0.8.1. July 2020. DOI: 10.5281/zenodo.3946761. URL: https://doi.org/10.5281/zenodo.3946761.

[KH17] Elliott D Kaplan and Christopher Hegarty. *Understanding GPS/GNSS: Principles and Applications*. Artech house, 2017.

[LLE97] S.T. Leutenegger, M.A. Lopez, and J. Edgington. "STR: A Simple and Efficient Algorithm for R-Tree Packing". In: *Proceedings 13th International Conference on Data Engineering*. 1997, pp. 497–506. DOI: 10.1109/ICDE.1997.582015.

[LS14] Debiao Lu and Eckehard Schnieder. "Performance Evaluation of GNSS for Train Localization". In: *IEEE transactions on intelligent transportation systems* 16.2 (2014), pp. 1054–1059.

[MM+17] Peter Mooney, Marco Minghini, et al. "A Review of OpenStreetMap Data". In: *Mapping and the Citizen Sensor* (2017), pp. 37–59.

[NZ14] Pascal Neis and Dennis Zielstra. "Recent Developments and Future Trends in Volunteered Geographic Information Research: The Case of OpenStreetMap". In: *Future internet* 6.1 (2014), pp. 76–106.

[OBL+17] Jon Otegui, Alfonso Bahillo, Iban Lopetegi, and Luis Enrique Díez. "A Survey of Train Positioning Solutions". In: *IEEE Sensors Journal* 17.20 (2017), pp. 6788–6797.

[Ram] Sebastián Ramírez. *FastAPI*. URL: https://github.com/fastapi/fastapi.

[SC18] Susan Shaheen and Adam Cohen. "Is It Time for a Public Transit Renaissance?" In: *Journal of Public Transportation* 21.1 (2018). The Future of Public Transportation, pp. 67–81. ISSN: 1077-291X. DOI: https://doi.org/10.5038/2375-0901.21.1.8. URL: https://www.sciencedirect.com/science/article/pii/S1077291X22000534.

[SD16] Martina Szabova and Frantisek Duchon. "Survey of GNSS Coordinates Systems". In: *European Scientific Journal* 12.24 (2016), pp. 33–42.

[SZK13] Erich Schubert, Arthur Zimek, and Hans-Peter Kriegel. "Geodetic Distance Queries on R-Trees for Indexing Geographic Data". In: *Advances in Spatial and Temporal Databases: 13th International Symposium, SSTD 2013, Munich, Germany, August 21-23, 2013. Proceedings 13*. Springer. 2013, pp. 146–164.

[ZMB+18] Ni Zhu, Juliette Marais, David Bétaille, and Marion Berbineau. "GNSS Position Integrity in Urban Environments: A Review of Literature". In: *IEEE Transactions on Intelligent Transportation Systems* 19.9 (2018), pp. 2762–2778. DOI: 10.1109/TITS.2017.2766768.

[ZZ10] Dennis Zielstra and Alexander Zipf. "Quantitative Studies on the Data Quality of OpenStreetMap in Germany". In: *Proceedings of GIScience*. 2010, pp. 1–7.