Demand-Responsive Transport to ensure accessibility, availability and reliability of rural public transport

REGIONAL WORKSHOP ON OPEN DATA DEVELOPMENT FOR PUBLIC AUTHORITIES
Output I2.4

Aalborg University
Dept. of Computer Science

Authors:
Kristian Torp
Magnus N. Hansen

31/12/2021
Project nr. #R101
(Interreg Baltic Sea Region)

Table of Content

1 Summary

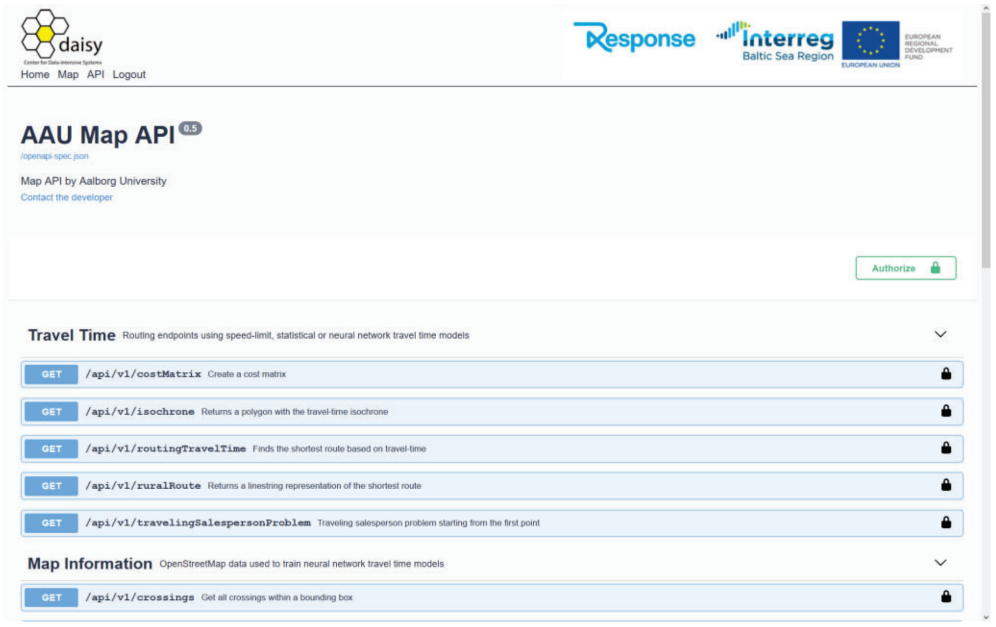
This report describes how the data warehouse developed for GPS data in report O2.1 of the RESPONSE project can be used to show how data can be used to solve common challenges in the transport sector. The focus is on the two Key Performance Indicators (KPIs) travel-time and fuel consumption. In addition, there are several additional information that can be extracted at the segment level and a route level.

The report focuses on using the data accessible via the RESTful API developed in the RESPONSE report O2.1. The full API is publicly available via the website https://mapapi.cs.aau.dk. To be able to use the data extracted from the RESTful API several different output formats are supported, including Microsoft Excel and the very widely used CSV file format.

2 API Overview

The landing (or main) page of the RESTful API is shown in Figure 1. This web page is built using only open-source software products such as the PostgreSQL database [1] for data storage, the Python [2] programming language for application logic, and the RESTFul [3] API documentation tool Swagger UI [4].

The API provides a web-based graphical user interface to use the developed RESTful API while at the same time allowing IT professionals to query the API using two standard tools for querying websites (using a URL via the HTTPS protocol or the `curl` [5] [6]command-line tool available on all major operating systems, i.e., Linux, macOS, and Windows).



AAU Map API 0.5

/openapi-spec.json

Map API by Aalborg University

Contact the developer

Travel Time Routing endpoints using speed-limit, statistical or neural network travel time models

GET `/api/v1/costMatrix` Create a cost matrix

GET `/api/v1/isochrone` Returns a polygon with the travel-time isochrone

GET `/api/v1/routingTravelTime` Finds the shortest route based on travel-time

GET `/api/v1/ruralRoute` Returns a linestring representation of the shortest route

GET `/api/v1/travelingSalespersonProblem` Traveling salesperson problem starting from the first point

Map Information OpenStreetMap data used to train neural network travel time models

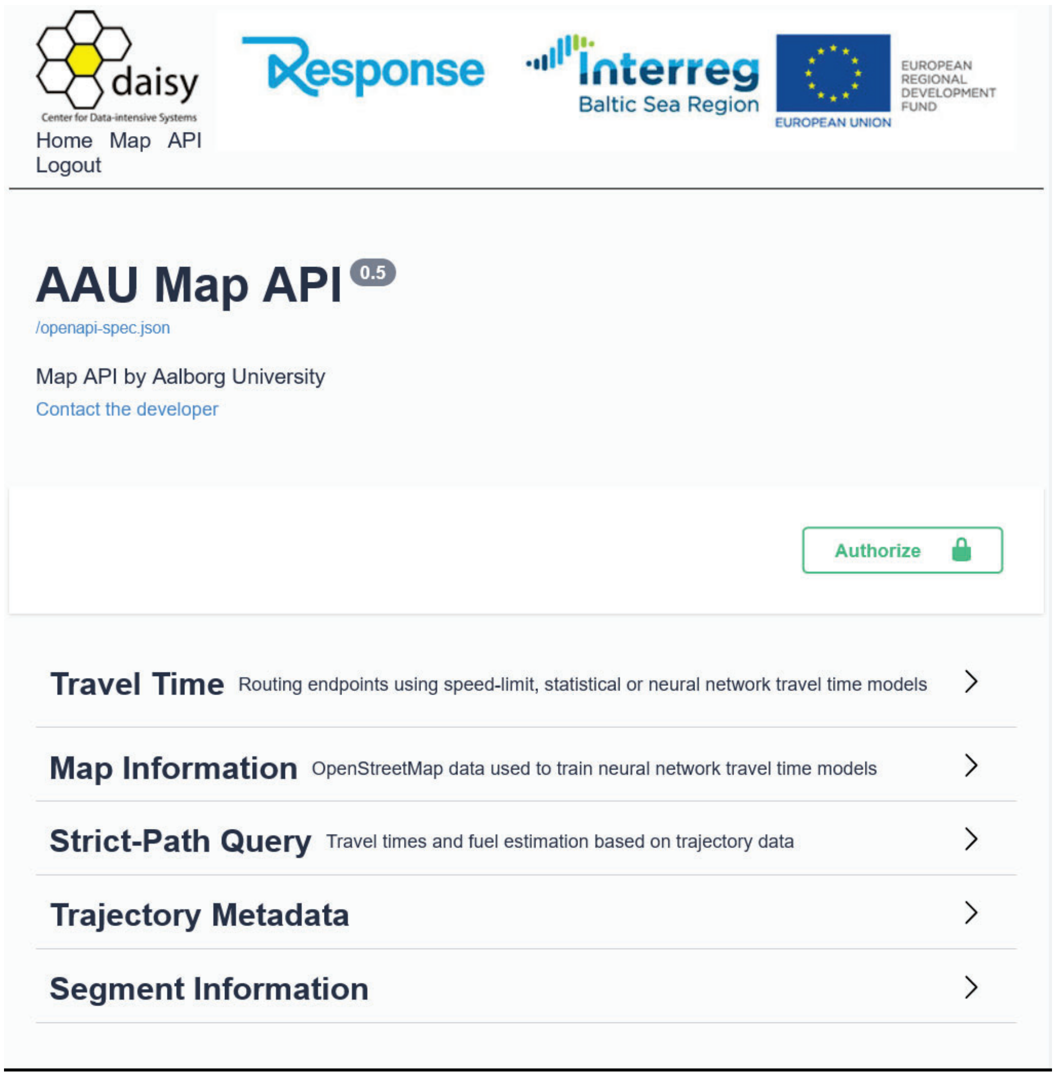
GET `/api/v1/crossings` Get all crossings within a bounding box

Figure 1 RESTful API Front Page (https://mapapi.cs.aau.dk)

The RESTful API is accessible from most, if not all web browsers, such as Apple Safari, Google Chrome, Microsoft Edge, and Mozilla Firefox.

2.1 API Subparts

The RESTful API is split into several functional subparts based on the information that they provide access to. The splitting of the API also makes it possible to restrict access to the subparts. Further, the smaller subparts make the API easier to understand by the users. The overview of the API subparts is shown in Figure 2. Here the subparts are illustrated by collapsing the information on the individual API endpoints shown in Figure 1.


Home Map API
Logout

AAU Map API 0.5
/openapi-spec.json

Map API by Aalborg University
Contact the developer

Authorize

Travel Time Routing endpoints using speed-limit, statistical or neural network travel time models

Map Information OpenStreetMap data used to train neural network travel time models

Strict-Path Query Travel times and fuel estimation based on trajectory data

Trajectory Metadata

Segment Information

Figure 2 The RESTful API Subparts on the

The five parts of the API are the following

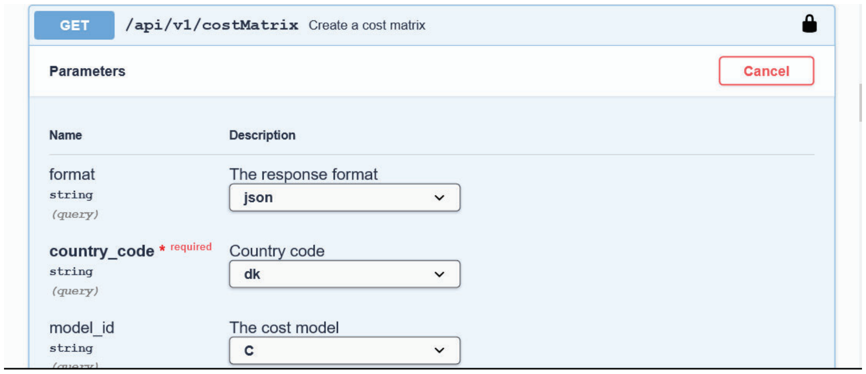
- **Travel time** (public) provides access to various routing information, e.g., point-to-point travel time and isochrone information.
- **Map Information** (login) provides access to the underlying OpenStreetMap (OSM) [7] digital map in general.
- **Strict-path queries** (login) provide access to GPDR complaint information on GPS data
- **Trajectory metadata** (login) provides access to details of individual trajectories.
- **Segment information** (login) provides access to the details of the digital map at the most detailed level, which is at the segment level.

The travel-time subpart of the RESTful API is available to the public. The four other subparts are only available to API users that are logged in. This is indicated by the (public) and (login) notations, respectively. Each subpart is explained in more detail later in this report.

2.2 Output Formats

For the data extracted from the RESTful API to be useful in the organization that uses the API, the data must be available in several different formats. In this section, we provided an overview of these output formats.

The user can specify the output format by using the *format* parameter shown as the first parameter in Figure 3. Here the output format is the widely used JSON format. The user can change the output format by picking a from a dropdown list (not shown in Figure 3).


GET /api/v1/costMatrix Create a cost matrix

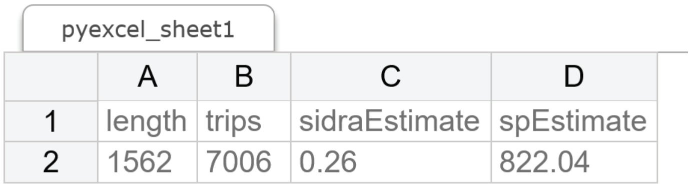
Parameters

Name	Description
format string (query)	The response format: json
country_code * required string (query)	Country code: dk
model_id string (query)	The cost model: c

Figure 3 Specifying the Output Format

The API supports four different output formats. Not all formats are relevant for all API endpoints. The output formats supported are the following.

- **JSON** [8] is a widely used data format in a web context. If the output produced is a geometry, e.g., a point, linestring, or a polygon, the specialized GeoJSON format is used. The GeoJSON format makes it very simple to visualize the geometries extracted in a broad range of both open-source and commercial GIS software products.
- **CSV** [6] is a Comma-Separated Values text file. The CSV format is widely used for data exchange and makes it simple to import the data extracted from the API into existing databases.
- **XSLX** is the Microsoft Excel format. This makes it possible for API users to experiment with the visualization of the data. The Excel format is very widely used for data analysis in many public and private organizations.
- **Handsontable** is a format for displaying tabular data directly in a web browser. An example is shown in Figure 4 where the result of a RESTful API call is shown directly in a web browser window. This format makes it convenient to experiment with the API.


pyexcel_sheet1

	A	B	C	D
1	length	trips	sidraEstimate	spEstimate
2	1562	7006	0.26	822.04

Figure 4 The handontable Output Format shown in Web Browser

2.3 Level of Information

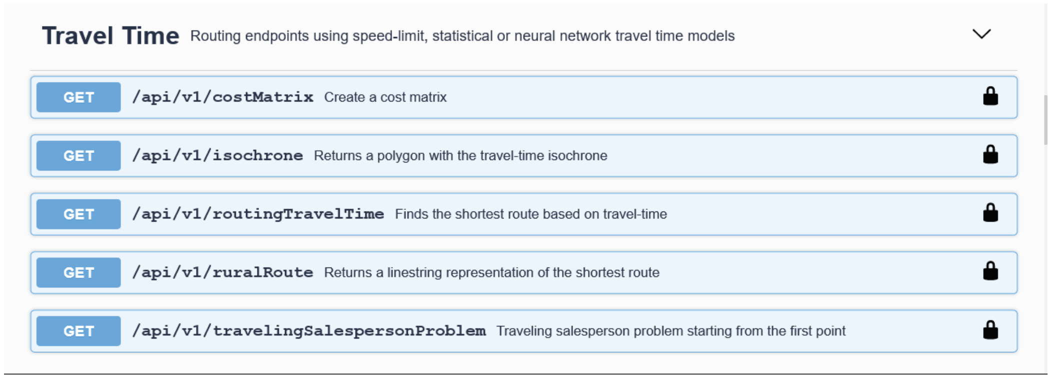
A huge challenge in designing an API that is based on GPS data is the GDPR rules. A user of the API has an interest in getting as much information as possible while the GDPR rules clearly state that individuals have a right to privacy.

To provide as much relevant information to a user, the API can provide information at multiple levels.

- At a **GPS point** level, there is access to the first and last GPS record on a single segment, if there are at least 11 or more vehicles. However, a user cannot combine GPS points from multiple segments as the temporal information is provided minimum at a 15 minutes interval.
- At a **segment level**, the travel time and fuel consumption estimates are available. This makes it possible to build additional services on top of the API. Also because the API can be accessed via a programming language, i.e., extraction can be automated.
- At a **route level**, the travel time and fuel consumption estimates are available. The benefit is that a user can get a combined result for the travel on multiple, consecutive road segments. This is convenient as the user does not need to combine the multiple segment-level queries. It is also considered more accurate as turns are taken into consideration. This is not possible at the segment level.

3 Travel Time RESTful API Endpoints

The travel-time API endpoints provide access to high-level services that are widely used in the transportation industry. These services are all freely available via the internet as they do not return any personal information and therefore are fully GDPR compliant. The services cover all of Denmark, the Faroe Islands, and Sweden. An overview of the travel-time RESTful API endpoints is shown in Figure 5.



Travel Time Routing endpoints using speed-limit, statistical or neural network travel time models

GET /api/v1/costMatrix Create a cost matrix

GET /api/v1/isochrone Returns a polygon with the travel-time isochrone

GET /api/v1/routingTravelTime Finds the shortest route based on travel-time

GET /api/v1/ruralRoute Returns a linestring representation of the shortest route

GET /api/v1/travelingSalespersonProblem Traveling salesperson problem starting from the first point

Figure 5 Travel Time RESTful API Endpoints

The result of using the *routingTravelTime* and *ruralRoute* endpoints is shown in Figure 6. Here a route starts at the needled labeled 1 the route ends at the Central Train Station, in Karlstad, Sweden. The addresses are visited in the order specified, i.e., first 1, then 2, then 3, and so on. The travel time for the entire route is shown in Figure 6.

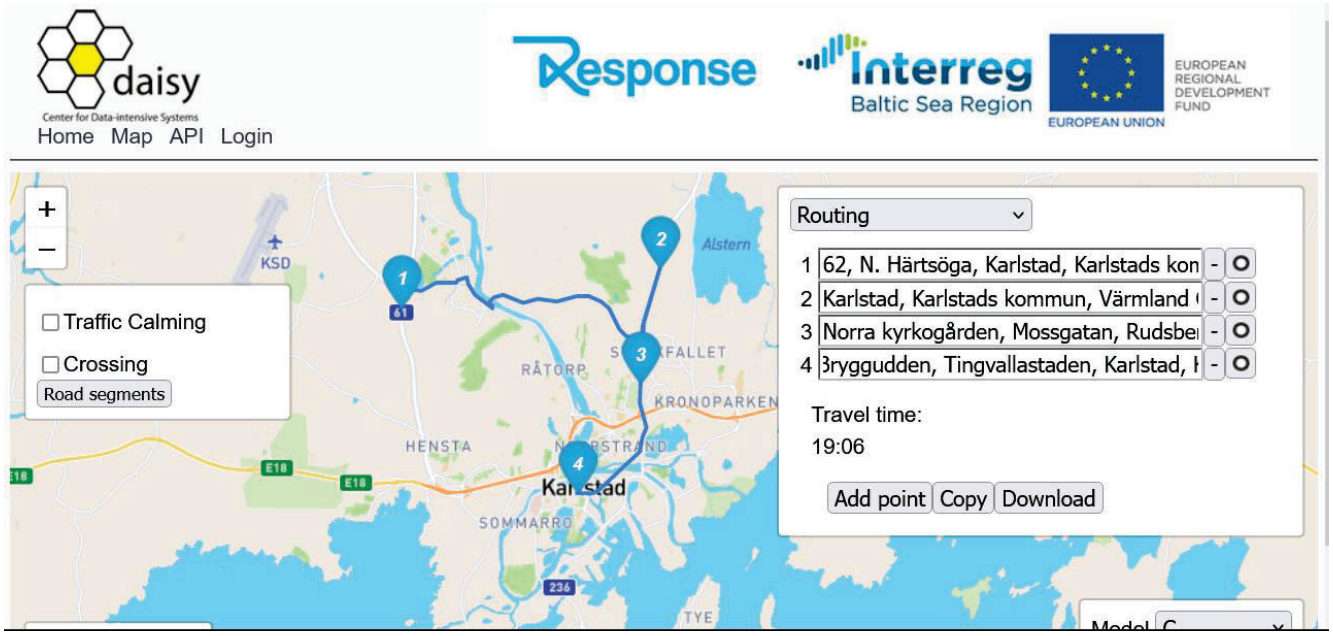

daisy
Center for Data-intensive Systems
Home Map API Login
Traffic Calming
Crossing
Road segments
Routing
1 62, N. Härtsöga, Karlstad, Karlstads kon
2 Karlstad, Karlstads kommun, Värmland
3 Norra kyrkogården, Mossgatan, Rudsbe
4 Bryggudden, Tingvallastaden, Karlstad,
Travel time:
19:06
Add point
Copy
Download
Karlstad
Model C

Figure 6 Result using the two Routing RESTful API Endpoints

The result of using the *isochrone* endpoint is shown in Figure 7. The isochrone center is in the Swedish town of Kumla. Each ring in isochrone represents 10 minutes of travel time, e.g., the yellow parts can be reached in 10 minutes, the green parts in 20 minutes, the red parts in 30 minutes, and the blue parts in 40 minutes.

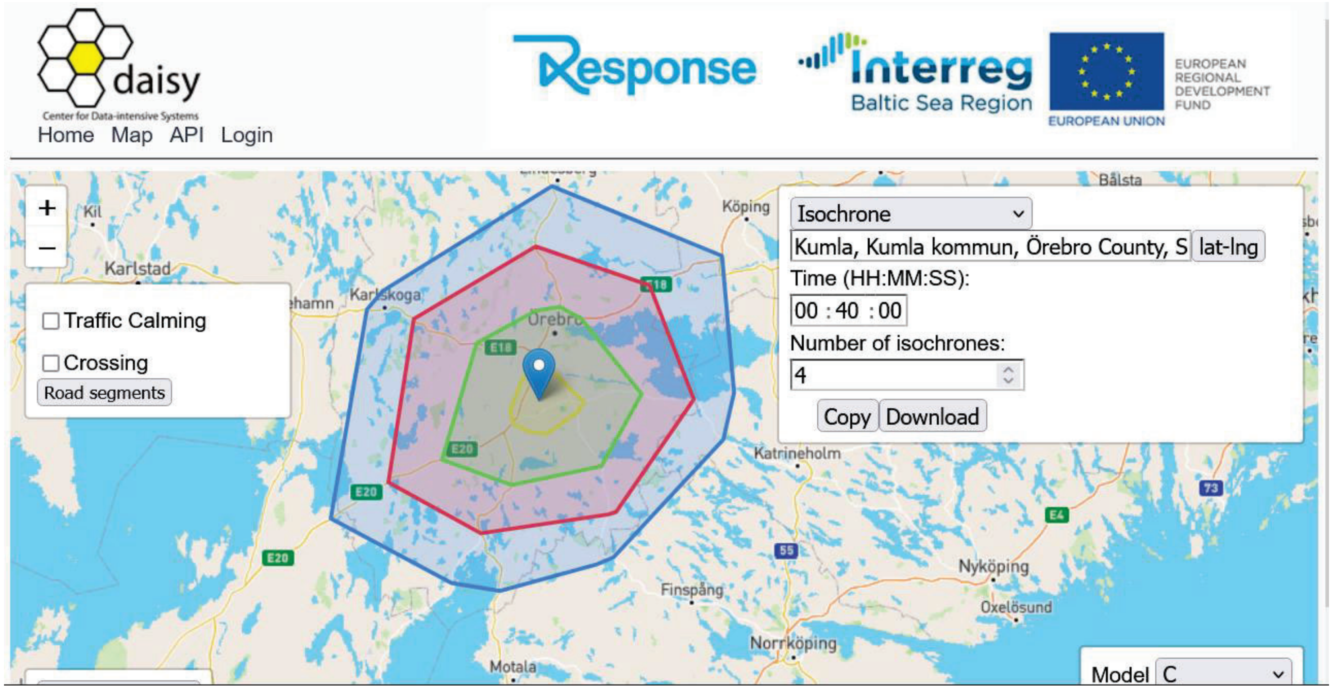
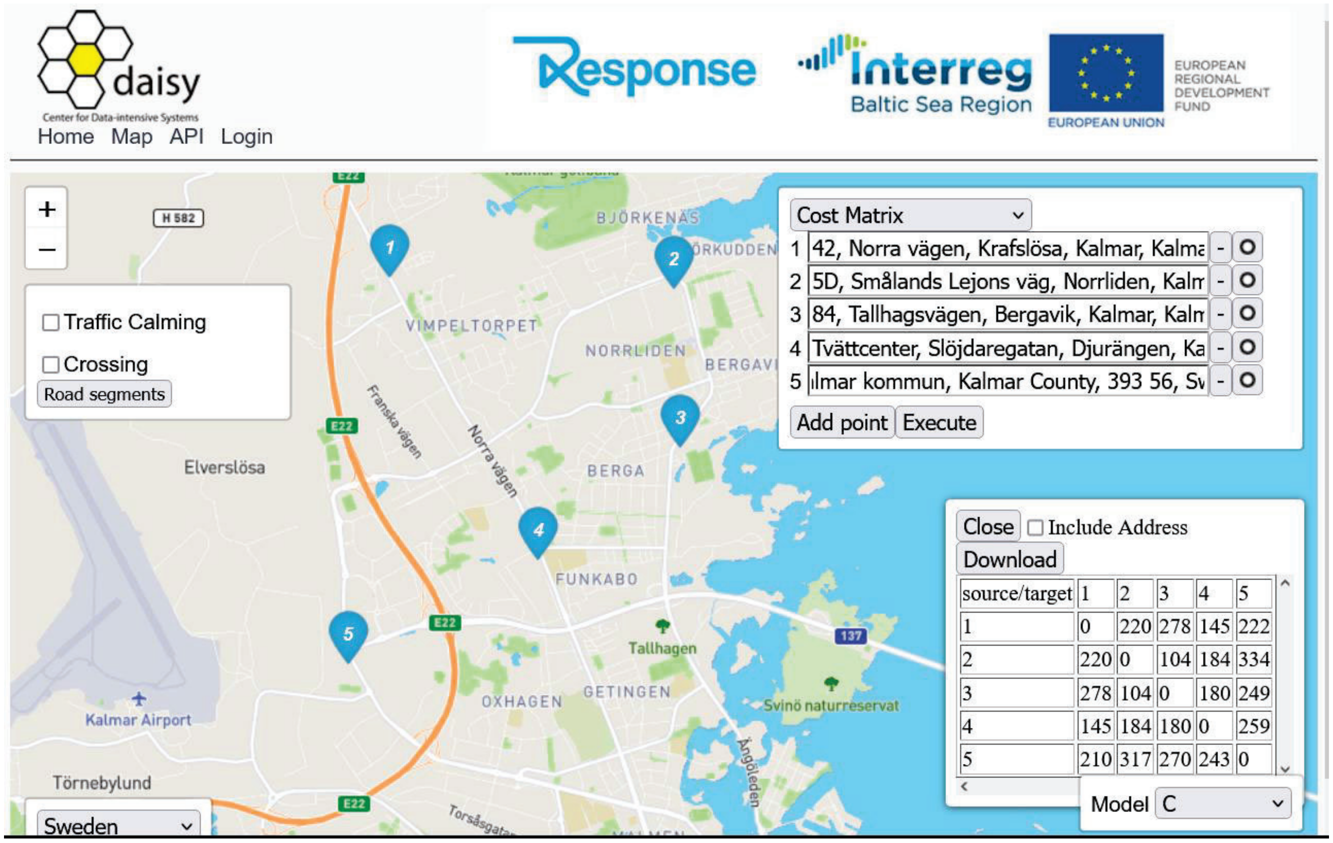


Figure 7 Result using the isochrone RESTful API Endpoint

The result of using the *costMatrix* endpoint is shown in Figure 8. The table at the bottom left shows the travel time between the numbered needles on the map. A cost matrix is a convenient way to present a larger number of travel times between multiple destinations. The map shows the area near Kalmar, Sweden.



1 42, Norra vägen, Krafslösa, Kalmar, Kalma
2 5D, Smålands Lejons väg, Norrliden, Kalm
3 84, Tallhagsvägen, Bergavik, Kalmar, Kalm
4 Tvättcenter, Slöjdaregatan, Djurängen, Ka
5 ılmar kommun, Kalmar County, 393 56, Sv

source/target	1	2	3	4	5
1	0	220	278	145	222
2	220	0	104	184	334
3	278	104	0	180	249
4	145	184	180	0	259
5	210	317	270	243	0

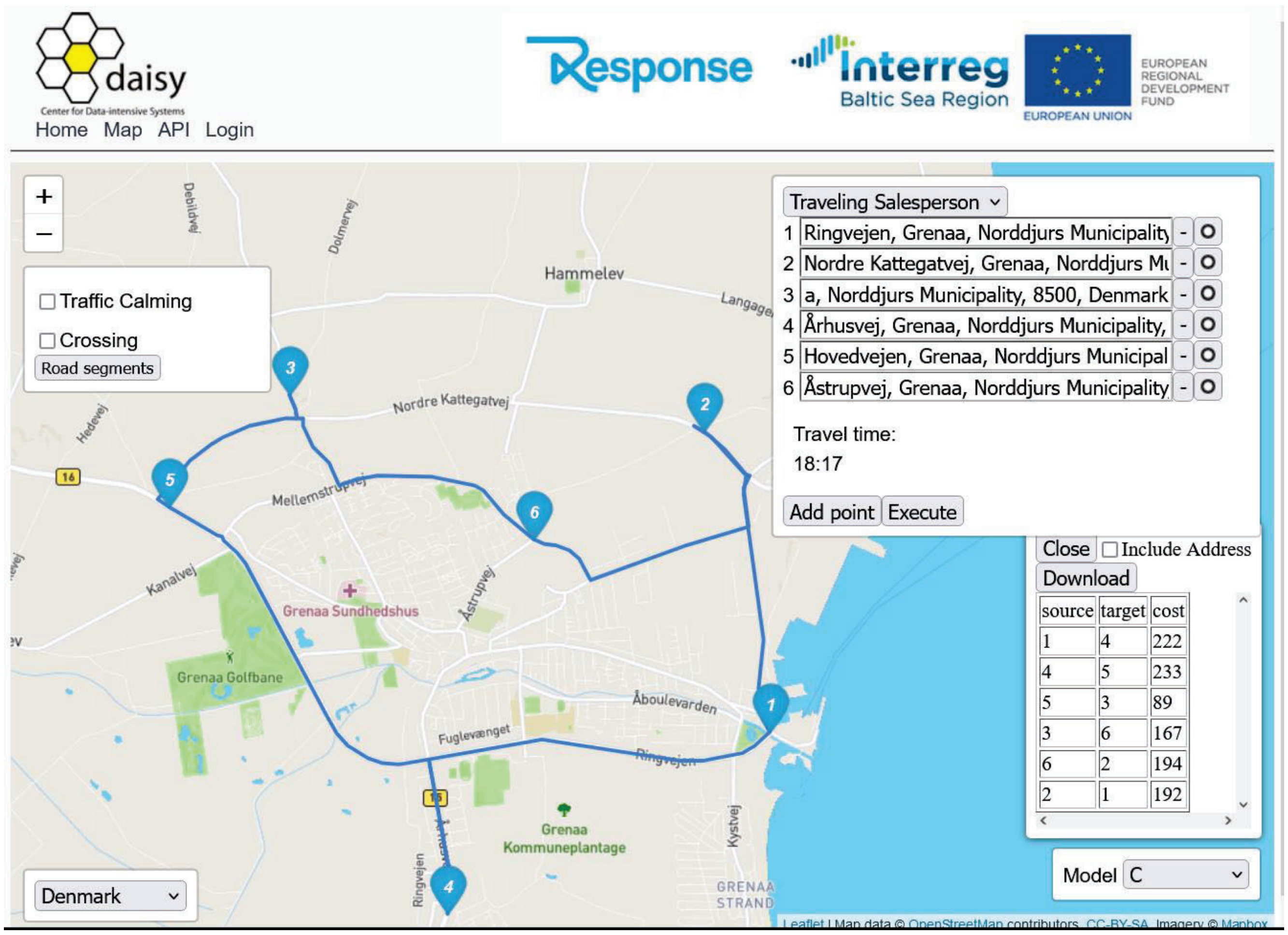
Figure 8 Result using the costMatrix RESTful Endpoint

The result of using the *travelingSalesPersonProblem* endpoint is shown in Figure 9

The goal of the traveling salesperson problem is to find the fastest route that starts at the needle label 1 and then visits all the other needles, labeled 2 to 6, and then returns to the address at needle 1 again. The map shows the area near the city of Grenå, Denmark.

The routes to take is shown in the table in the lower right corner in Figure 10. For this example, the first leg of the journey is from needle 1 to needle 4, second from needle 4 to needle 5, third from needle 5 to needle 3, and on.

The total travel time for visiting all needles is shown in Figure 10. Please note that the traveling salesperson is in a class of problems that only can be solved efficiently using heuristics.



1 Ringvejen, Grenaa, Norddjurs Municipality
2 Nordre Kattegatvej, Grenaa, Norddjurs Mu
3 a, Norddjurs Municipality, 8500, Denmark
4 Århusvej, Grenaa, Norddjurs Municipality,
5 Hovedvejen, Grenaa, Norddjurs Municipal
6 Åstrupvej, Grenaa, Norddjurs Municipality

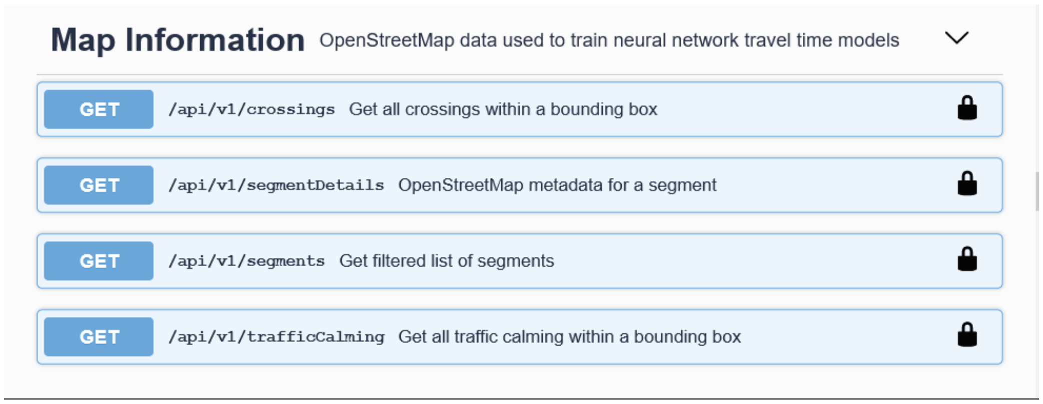
Travel time:
18:17

source	target	cost
1	4	222
4	5	233
5	3	89
3	6	167
6	2	194
2	1	192

Figure 9 The travelingSalesPersonProblem API Endpoint

Map Information RESTful API Endpoints

In the section, we describe the RESTful endpoints that provide access to the underlying map foundation. The purpose of these endpoints is simply access to basic map data that can provide useful insight to transportation authorities. A screenshot of the endpoints is shown in Figure 10.

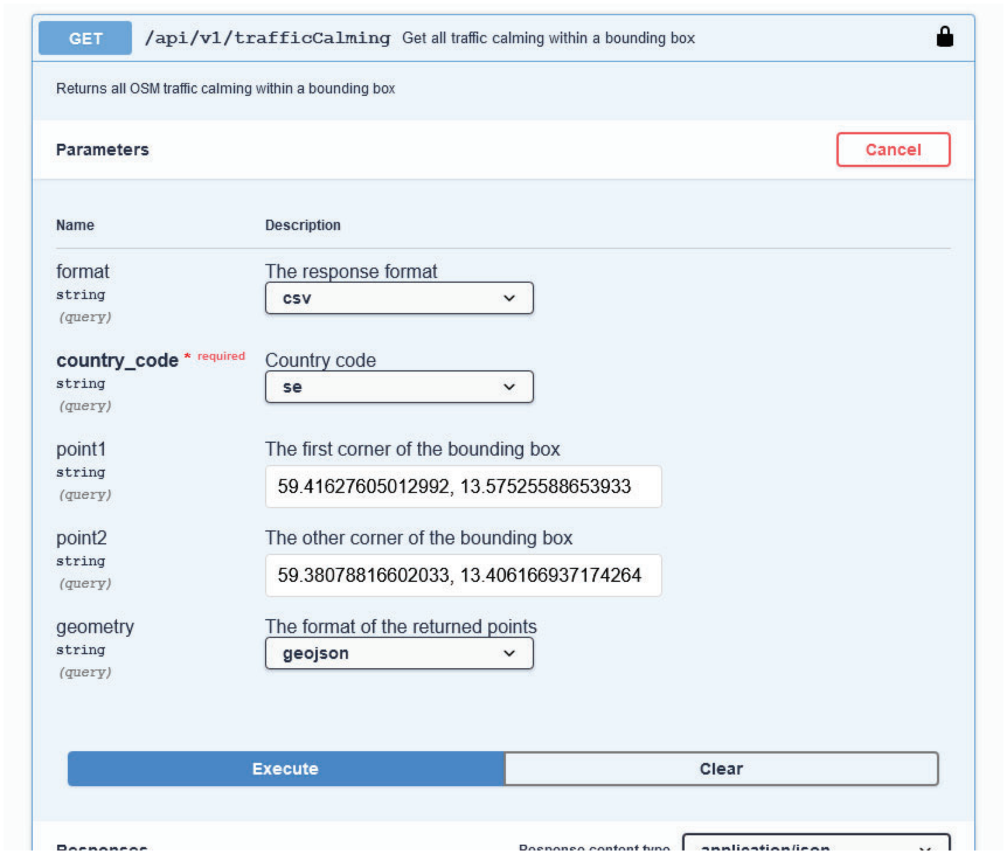


Map Information OpenStreetMap data used to train neural network travel time models

- GET `/api/v1/crossings` Get all crossings within a bounding box
- GET `/api/v1/segmentDetails` OpenStreetMap metadata for a segment
- GET `/api/v1/segments` Get filtered list of segments
- GET `/api/v1/trafficCalming` Get all traffic calming within a bounding box

Figure 10 Map Information RESTful API Endpoints

The screenshot shown in Figure 10 is a part of Figure 1. The user simply has to scroll down to access these endpoints.
All API endpoints are accessed similarly. An example of using the *trafficCalming* API endpoint is shown in Figure 11. In this example, all traffic calming on the road network in the Karlstad Area, Sweden.



GET `/api/v1/trafficCalming` Get all traffic calming within a bounding box

Returns all OSM traffic calming within a bounding box

Parameters

Name	Description
format string (query)	The response format: csv
country_code * required string (query)	Country code: se
point1 string (query)	The first corner of the bounding box: 59.41627605012992, 13.57525588653933
point2 string (query)	The other corner of the bounding box: 59.38078816602033, 13.406166937174264
geometry string (query)	The format of the returned points: geojson

Figure 11 Using the trafficCalming Endpoint

The result of the API endpoint call in Figure 11 is illustrated in Figure 12 using the PostgreSQL database. This shows that it is possible to integrate the data extracted from the RESTful API with other third-party software products such as the open-source software product QGIS [9] and pgAdmin [10].

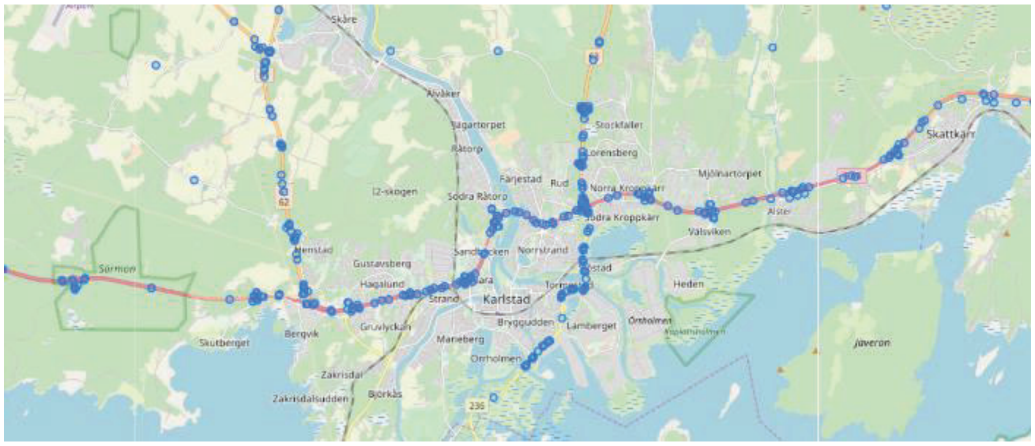
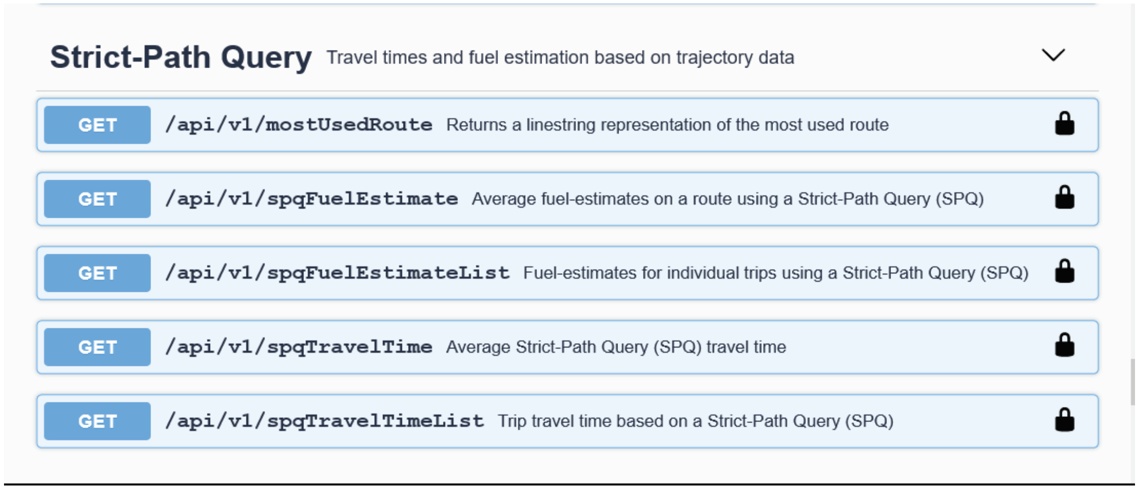


Figure 12 Result of the Traffic Calming API Endpoint Call

4 Strict-Path Query RESTful API Endpoints

An overview of the strict-path query subpart of the RESTful API endpoints is shown in Figure 13. This part of the API provides access to detailed travel-time information and fuel consumption estimates on the parts of the road network where GPS data is available. Please note that to fulfill the GDPR rules it is not possible to extract information such that it is possible to identify individual drivers.



Strict-Path Query Travel times and fuel estimation based on trajectory data

GET `/api/v1/mostUsedRoute` Returns a linestring representation of the most used route

GET `/api/v1/spqFuelEstimate` Average fuel-estimates on a route using a Strict-Path Query (SPQ)

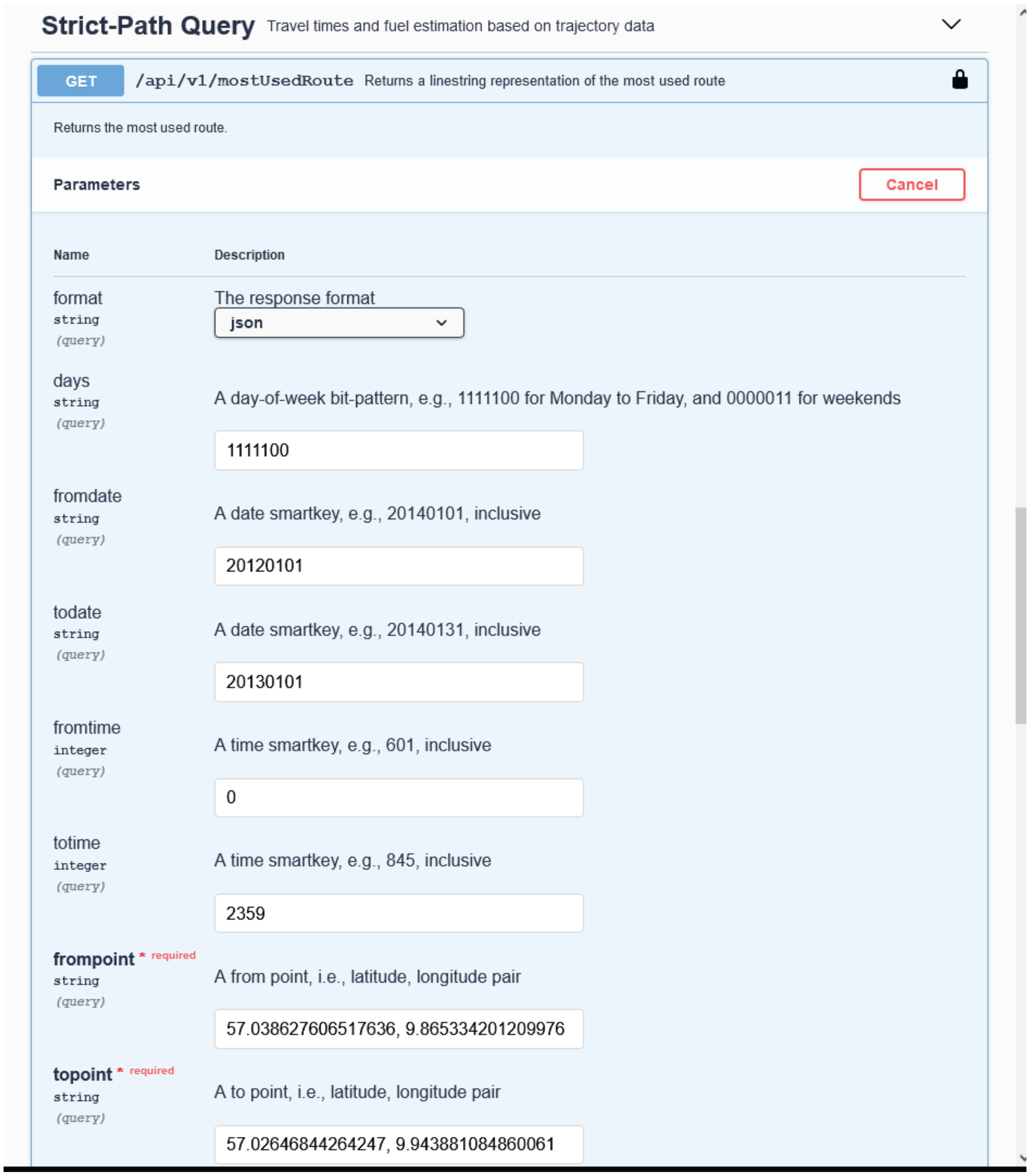
GET `/api/v1/spqFuelEstimateList` Fuel-estimates for individual trips using a Strict-Path Query (SPQ)

GET `/api/v1/spqTravelTime` Average Strict-Path Query (SPQ) travel time

GET `/api/v1/spqTravelTimeList` Trip travel time based on a Strict-Path Query (SPQ)

Figure 13 Strict-Path Query RESTful API Endpoints

An example of using the *mostUsedRoute* endpoint is shown in Figure 14. The request returns the most used route between two points. These two points are provided by the user in the parameters *fromPoin* and *toPoint* at the bottom of Figure 15. The format is JSON specified in the *format* parameter. The user can specify several temporal filters using the *days*, *fromdate*, *todate*, *fromtime*, and *totime*.


Strict-Path Query Travel times and fuel estimation based on trajectory data

GET `/api/v1/mostUsedRoute` Returns a linestring representation of the most used route

Returns the most used route.

Parameters

Name	Description
format string (query)	The response format: json
days string (query)	A day-of-week bit-pattern, e.g., 1111100 for Monday to Friday, and 0000011 for weekends: 1111100
fromdate string (query)	A date smartkey, e.g., 20140101, inclusive: 20120101
todate string (query)	A date smartkey, e.g., 20140131, inclusive: 20130101
fromtime integer (query)	A time smartkey, e.g., 601, inclusive: 0
totime integer (query)	A time smartkey, e.g., 845, inclusive: 2359
frompoint * required string (query)	A from point, i.e., latitude, longitude pair: 57.038627606517636, 9.865334201209976
topoint * required string (query)	A to point, i.e., latitude, longitude pair: 57.02646844264247, 9.943881084860061

Figure 14 Using the mostUsedRoute Endpoint

The result of the request specified in Figure 14 is shown in Figure 15 using the QGIS GIS system. The result is a GeoJSON string that can be visualized in many GIS products.

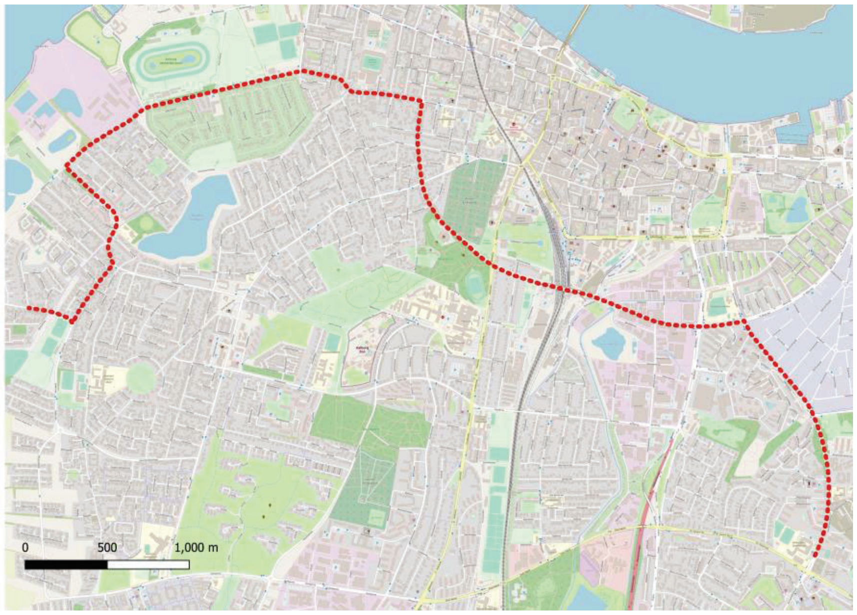
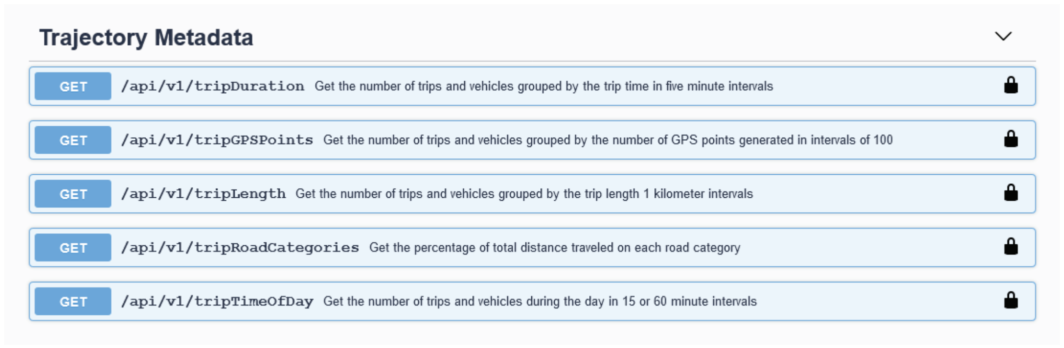


Figure 15 Result of mostUsedRoute Endpoint Displaying Most Used from Going Left to Right

5 Trajectory Metadata RESTful API Endpoints

An overview of the trajectory metadata API endpoints is shown in Figure 16. The endpoints provide information about trips (typically called trajectories).



Trajectory Metadata

GET `/api/v1/tripDuration` Get the number of trips and vehicles grouped by the trip time in five minute intervals

GET `/api/v1/tripGPSPoints` Get the number of trips and vehicles grouped by the number of GPS points generated in intervals of 100

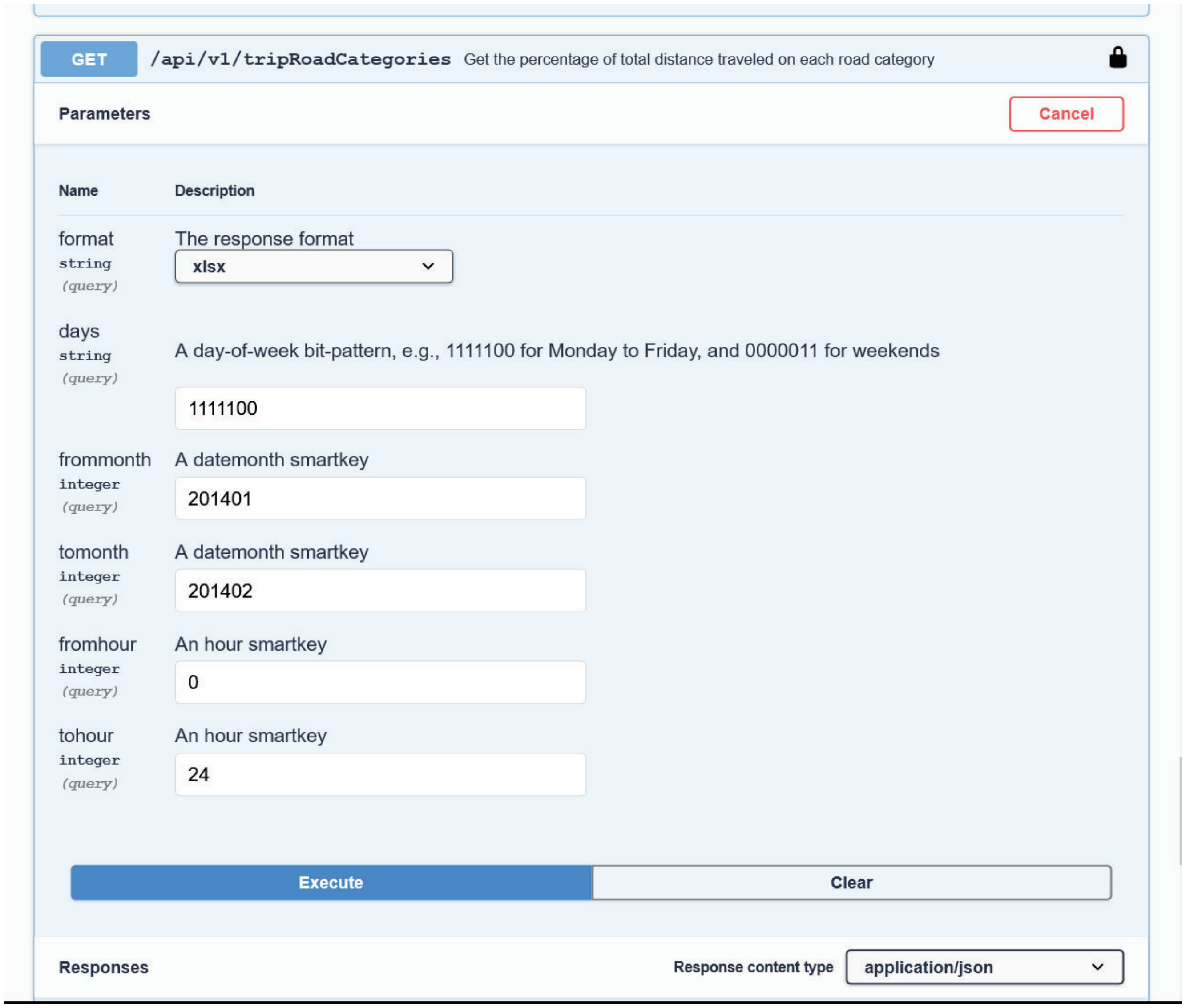
GET `/api/v1/tripLength` Get the number of trips and vehicles grouped by the trip length 1 kilometer intervals

GET `/api/v1/tripRoadCategories` Get the percentage of total distance traveled on each road category

GET `/api/v1/tripTimeOfDay` Get the number of trips and vehicles during the day in 15 or 60 minute intervals

Figure 16 Trajectory Metadata RESTful API Endpoints

An example of using the *tripRoadCategories* endpoint is shown in Figure 17. The endpoint returns the percentage usage of the road network, e.g., the percentages of the trips on motorways, trunk, or residential roads. The format is in this case the Excel format. The first five rows of the result of the request in Figure 16 are shown in Table 1. Because the Excel format allows the user to visualize the result in different fashions.



GET `/api/v1/tripRoadCategories` Get the percentage of total distance traveled on each road category

Parameters

Name	Description
format string (query)	The response format: xlsx
days string (query)	A day-of-week bit-pattern, e.g., 1111100 for Monday to Friday, and 0000011 for weekends: 1111100
frommonth integer (query)	A datemonth smartkey: 201401
tomonth integer (query)	A datemonth smartkey: 201402
fromhour integer (query)	An hour smartkey: 0
tohour integer (query)	An hour smartkey: 24

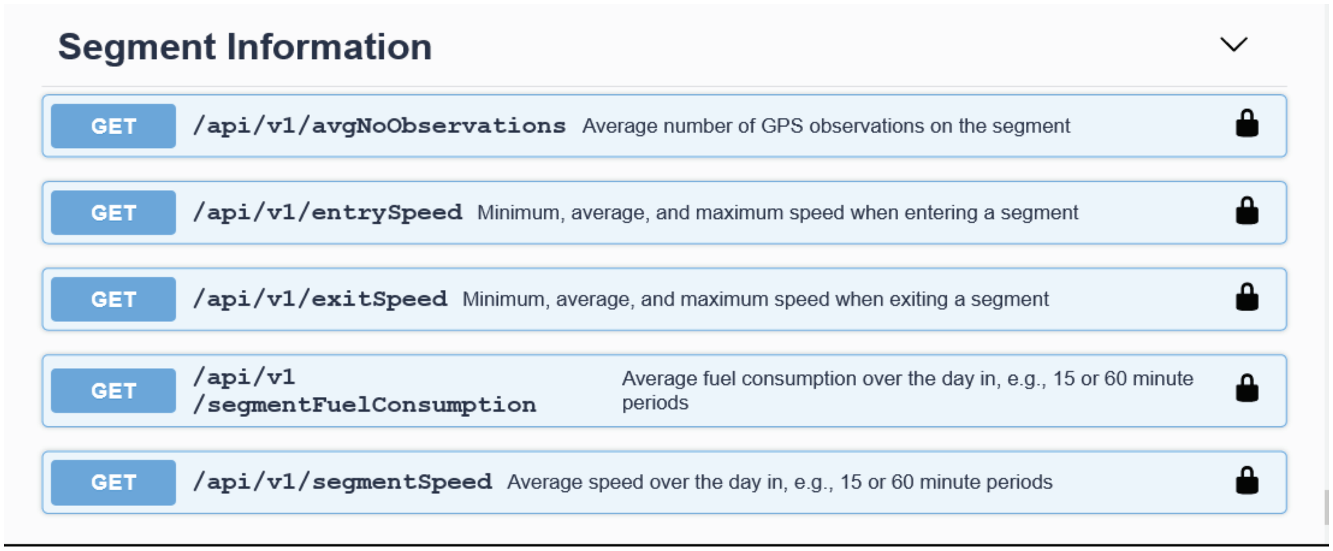
Figure 17 The tripRoadCategories *Endpoint*

Table 1 Usage of Road Network

date	motorway	trunk	primary	secondary	tertiary	residential	misc
20140101	26.4	2.08	16.37	23.65	14.45	7.31	9.73
20140102	31.45	2.33	13.2	23.63	14.67	6.63	8.09
20140103	36.47	1.69	12.64	22.38	13.27	5.91	7.65
20140106	40.33	2.22	11.67	19.82	13.05	5.45	7.45
20140107	35.01	2.23	12.86	22.23	13.82	5.58	8.27

6 Segment Information RESTful API Endpoints

The set of segment information endpoints is shown in Figure 18. The endpoints provide information related to road networks at the finest level of granularity of a single segment. This is typically the distance between two intersections. The endpoints focus on the two KPIs travel time and fuel consumption.



Segment Information

GET /api/v1/avgNoObservations Average number of GPS observations on the segment

GET /api/v1/entrySpeed Minimum, average, and maximum speed when entering a segment

GET /api/v1/exitSpeed Minimum, average, and maximum speed when exiting a segment

GET /api/v1/segmentFuelConsumption Average fuel consumption over the day in, e.g., 15 or 60 minute periods

GET /api/v1/segmentSpeed Average speed over the day in, e.g., 15 or 60 minute periods

Figure 18 Segment Information RESTful API Endpoints

As an example, the *segmentFuelConsumption* endpoint provides information on the fuel consumption on a segment. An example of this is shown in Figure 19. Similarly, the *segmentSpeed* endpoint provides information on the travel speed of a segment over the time of the day. An example is shown in Figure 20. The two examples, in Figure 19 and Figure 20 use the Excel output format. Directly from the file downloaded via the API, these graphs can be created.

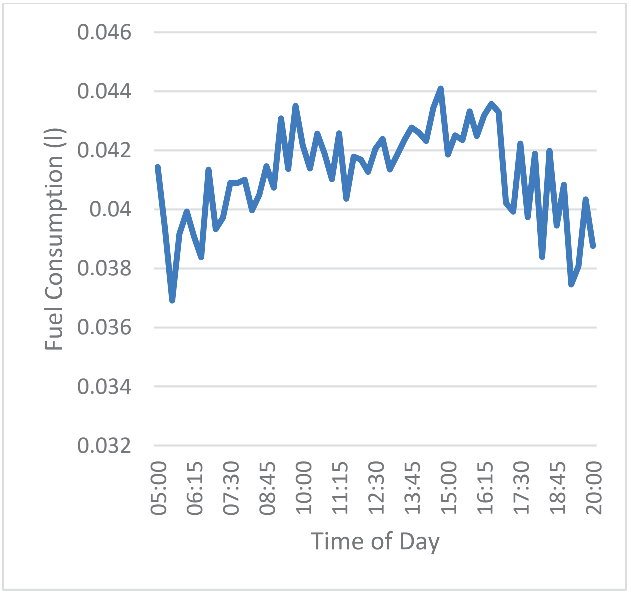


Figure 19 Fuel Consumption over Time of Day

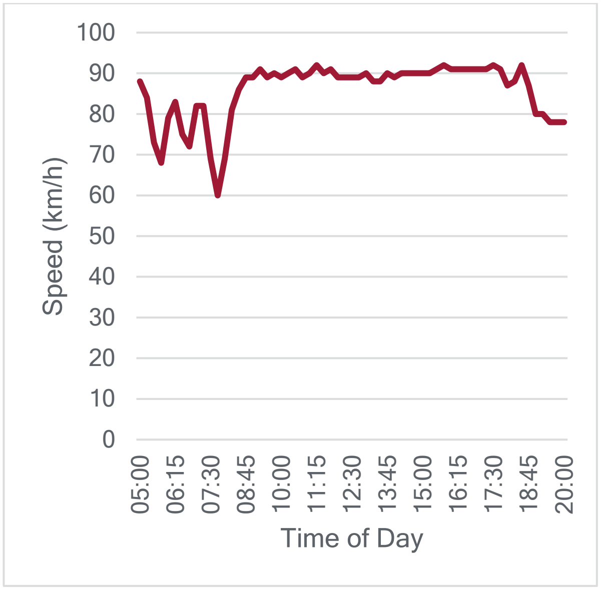


Figure 20 Speed over Time of Day

7 Master Data Management and Self-Guided Tutorials

A master data management seminar aimed at providing an introduction to open data accessibility, formats, quality, and added values in terms of economic and environmental performance evaluation was conducted during the first consortium meeting. The slides are available online.

The planned regional workshops could not be done due to travel restrictions related to the Corona Virus pandemic. The regional workshops are replaced by the following online/offline material for self-study.

- This report has been created to provide an overview of the functionality of the RESTful API. This report is targeted towards an audience that needs to have illustrated how open data can be used to support solving the challenges in the transport sector.
- A technical report providing an overview of the complete API is listed in Appendix A. This document is ready for publication in a technical outlet. The target audience is IT developers in transport organizations, particularly outside the RESPONSE project.
- A programmatic tutorial of the complete RESTful API using the Python programming language Is provided in Appendix B. The tutorial uses the very popular Jyputer Notebook framework that allows combining programming code and tables/figures. The tutorial is targeted towards a technical audience in a transports organization's IT department.
- A programmatic deep dive into computing travel times in intersections can be found in Appendix C. This tutorial builds on top of the first tutorial but is more detailed in solving a specific topic. The target audience is IT professionals in transportation organizations.
- This tutorial builds on top of the first tutorial but is more detailed in solving a specific topic. The target audience is IT professionals in transportation organizations.

8 Conclusion

This report has focused on how geodata can be utilized to evaluate the performance of the service level, the cost-effectiveness, and the sustainability of mobility. A master-data management seminar was given to all RESPONSE partners. All data is accessed via a RESTful API.

The RESTful API is built on top of a data warehouse with data from transportation. The focus has been on the two important KPIs travel time and fuel consumption. These KPIs are relevant for most, if not all transportation organizations. The solution provided is general and can be used in any region or country within the EU. The set of minimum requirements for adding data to the data warehouse has been listed.

The report has provided a large number of examples of the outputs on maps or tables. This shows that data can easily be presented in a manner that is understandable to both the public and decision-makers. The many examples of output from the RESTful API show that the two KPIs in focus (travel time/fuel consumption) can be easily presented on a map or integrated with other data in a transportation organization because the API supports a wide range of well-known output formats.

Due to the lockdown and travel restrictions related to the Coronavirus pandemic four planned seminars have been replaced by a) this report b) an online tutorial aim at developers, and c) two programming tutorials. All material is freely available for download. The tutorials are based on the widely used Jupyter Notebook technology.

9 References

[1] PostgreSQL, "PostgreSQL Homepage," [Online]. Available: https://www.postgresql.org/.

[2] Python, "Python Homepage," [Online]. Available: https://www.python.org/.

[3] "What is REST," REST API Tutorial , [Online]. Available: https://restfulapi.net/.

[4] Smartbear, "Swagger IO," [Online]. Available: https://swagger.io.

[5] T. C. Project, "curl://," [Online]. Available: https://curl.se/.

[6] Wikipedia, "Comma-separated values," [Online]. Available: https://en.wikipedia.org/wiki/Comma-separated_values.

[7] OpenStreetMap, "OpenStreetMap," [Online]. Available: https://www.openstreetmap.org/.

[8] ECMA-404, "ECMA-404 The JSON Data Interchange Standard.," [Online]. Available: https://www.json.org.

[9] QGIS, "QGIS Homepage," [Online]. Available: https://www.qgis.org.

[10] PGAdmin, "PGAmin Homepage," [Online]. Available: https://www.pgadmin.org/.

Appendix A Technical Overview of the RESTful API

A RESTFul API for GPS Trajectory Data Analysis

Magnus N. Hansen
Aalborg University
Dept. of Computer Science
Aalborg, Denmark
mnha@cs.aau.dk

Kristian Torp
Aalborg University
Dept. of Computer Science
Aalborg, Denmark
torp@cs.aau.dk

Abstract

High-frequent GPS data is being collected in very large quantities from vehicles this allows for detailed travel-time studies of a sequence of road segments and also to estimate the fuel consumption. In this paper, we present a novel RESTFul API where users can query high-frequent GPS data at three different levels: a trajectory level, a road segment level, and an individual GPS record level. Further, the API provides two different fuel estimates derived from the trajectory data. The tutorial shows how to use the RESTFul API to analyse traffic to, e.g., find problem with congestion, determine the fuel consumption, or compute turn-times in intersections. Concrete examples related to the three levels trajectory, segment, and point are presented. Results are available in multiple forms and it is shown how to visual the CSV format result in Microsoft Excel.

CCS Concepts

• **Information systems** → **Information integration**.

Keywords

trajectory, GPS, API, RESTFul, travel time, fuel estimate

ACM Reference Format:
Magnus N. Hansen and Kristian Torp. 2021. A RESTFul API for GPS Trajectory Data Analysis. In *Proceedings of* . ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

APIs from map/traffic-data vendors such as Google, HERE, and TomTom are widely used by traffic planners in both the public and private sector. With such APIs, it is possible to get information on both single road segments and routing information from Point A to Point B.

However, these APIs do not make it possible to get information about the GPS records from a single vehicle that traverses a road segment. Similarly, is travel-time information are from many different vehicles and merged into a single value. This means that it is not possible to examine the spread in travel-time on a single road segment or know the ground-truth travel-time information on a route, i.e., all vehicles have strictly followed the same route from Point A to Point B without detours or stops (non-traffic related).

UN's focus on reducing CO_2 emissions makes it important know about the fuel consumption on a route. Real fuel consumption from vehicles is very hard to get access to therefore fuel estimates are good alternatives. As an example, Figure 1 shows the fastest route (East-to-West) on a major road in Aalborg, Denmark.

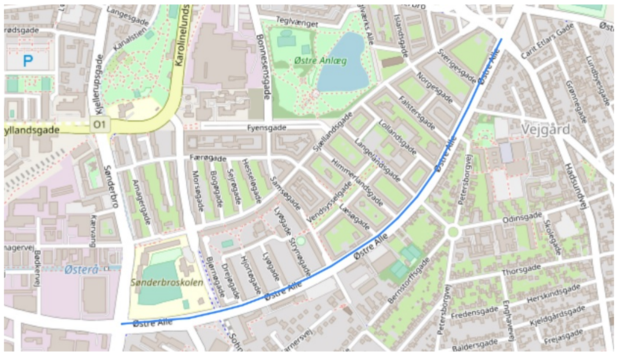
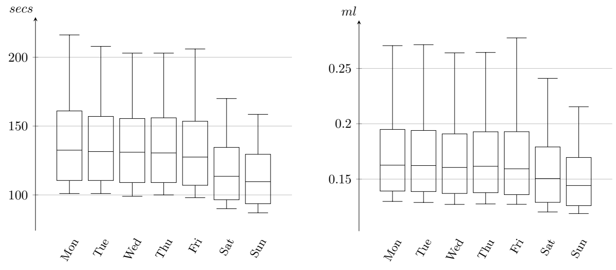


Figure 1: Blue-Line Fastest Route

The distribution of the travel-time and fuel-consumption estimates for each day in the week is shown in Figure 2. Both the travel time and fuel consumption are lower during the weekend due to lower traffic congestion.

On this 1216 meters long route, there is a total of 9436 trajectories over two years. There is an average of 10.92 GPS points on the first segment that is 135 meters long and has the OSM road category *secondary*. On the last segment on the route shown in Figure 1 the entry speed is on average 38.86 km/h. The minimum and maximum entry speeds are 0 km/h and 79.00 km/h, respectively. These numbers are all examples of data that can be extracted from the API and that is demonstrated in the tutorial. The API [2] is publicly available.



(a) Travel Time (*secs*) **(b) Fuel Estimate (*ml*)**

Figure 2: Boxplot for Data in Route in Figure 1

The API runs on top of a GPS trajectory data warehouse containing ~1.5 billion GPS records (1Hz) from ~1.35 million trajectories recorded by ~500 vehicles over two years.

In this paper, we provide an overview of the RESTFul API endpoints users can call and the results returned. In addition, we outline the content of a presentation of the API. We briefly described the data foundation and sums up the paper.

2 RESTFul Endpoints

Examples of the endpoints in the RESTFul API are shown in Table 1. The endpoints are divided into three groups: *Single GPS Record* where users can query information from single GPS records, *Single Segment* that provides information on a single segment, and *Trajectory* where information is based on trajectories (or sub-trajectories). Note that the information for endpoints is based on map-matched high-frequent GPS data [5].

Endpoint	Description
GPS Record	
entrySpeed	The entry speed on a segment
exitSpeed	The exit speed on a segment
Segment	
avgNoObservations	Avg. number of GPS records on a segment
segmentSpeed	Speed on a segment in 15/60 min. interval
Trajectory	
spqTravelTimeList	Travel time each trajectory on a route
spqFuelEstimateList	Fuel estimation each trajectory on a route

Table 1: Examples of RESTFul Endpoints

The trajectory-based endpoints use strict-path queries (SPQ) [4]. This query type ensures that all trajectories returned, e.g., for the *spqTravelTimeList* endpoint follow exactly the same route (sequence of road segments) and only do traffic-related stops, e.g., at signalized intersections. As an example, the *spqTravelTimeList* endpoint is used to produce the graph in Figure 2a (data imported into Latex for visualization).

The two fuel-estimation endpoints return two different estimates. One using the Sidra-Trip estimation [1] and one using the Vehicle Specific Power (SP) estimate [3]. The first estimate has the unit *mL/s* the second estimate is without a unit. As an example, the Sidra-Trip estimate produced by the endpoint *spqFuelEstimateList* is used to produce the graph in Figure 2b.

The spatial and temporal parameters to the endpoints in Table 1 are listed in Table 2. Note that trajectory endpoints use two GPS points parameters.

Parameter	Description
point	WGS-84 (lon, lat) point
from date	The from date, a smartkey, e.g., 20190101
from date	The to date, a smartkey, e.g., 20191231
days	Bit strings of days, e.g., 1111100 for Monday-Friday
from time	The from time, a smartkey, e.g., 0815
to-time	The to time, a smartkey, e.g., 2359

Table 2: Overview Spatial/Temporal Parameters

3 Tutorial Outline

The tutorial is split into two parts: A 15 minutes general part and a second 75 minutes hands-on part where the attendees solve various specific tasks with the proposed RESTFul API based on a Jupyter Notebook and Microsoft Excel. The audience is naturally able to use the API from their computers.

3.1 Short Overview

The first part shortly introduced the concepts needed to work with the API, e.g., when is it trajectory, segment, or point. The parts covers.

- The map-matching of GPS records to a digital road network
- The idea behind the trajectory (strict-path queries) queries
- The conversion of GPS data to fuel-consumption estimates

3.2 Hands-On

The second part of the tutorial is hands-on, where several exercises in a Jupyter Notebook demonstrate the various parts of the API to the attendees. This includes how the API allows users to download data, e.g., to visualize the result in a spreadsheet such as Microsoft Excel or integration the result with other data sources.

The second part covers the topics.

- Documenting issues with congestion on single segments and routes. Further, checking if the morning congestion level is lower than the afternoon level?
- Examining the relation between travel speed and fuel consumption
- Testing if the individual travel times on a segment/route follow a normal distribution.
- Quantifying how travel time in an intersection is different going left, right, or straight.

The Jupyter Notebook contains a number of exemplary examples to be used as an outset. This includes the following.

- Routes with significantly different fuel estimates
- Routes with a different number of trajectories
- Segments with and without a rush hours on workdays

The second part contains examples where empty results are returned due the EU's strict GDPR rules. Further, we hope the audience provides feedback on missing functionality in the API.

4 Summary

In this paper, we presented a novel set of RESTFul endpoints that provides new capabilities to query GPS trajectory data. The API allows users to access GPS data at three different levels: trajectory, road segment, and GPS record. A major concern is retaining the anonymity and privacy of the drivers. Therefore sparsely covered areas cannot be queried. Similarly using strict-path queries means that we can only provide results on routes where vehicles have driven.

Acknowledgement

This work is supported by the EU Interreg RESPONSE project response-project.eu.

References

[1] Rahmi Akçelik and Mark Besley. 2003. Operating cost, fuel consumption, and emission models in aaSIDRA and aaMOTION. In *CAITR 2003.*

[2] Magnus N. Hansen and Kristian Torp. -. Daisy MapAPI. https://mapapi.cs.aau.dk/apidocs/.

[3] JL Jimenez-Palacios. 1998. Understanding and quantifying motor vehicle emissions with vehiclespecific power and TILDAS remote sensing. PhD thesis, Massachusetts Institute of Technology.

[4] Benjamin Krogh, Nikos Pelekis, Yannis Theodoridis, and Kristian Torp. 2014. Path-based Queries on Trajectory Data. In *22nd ACM SIGSPATIAL GIS.*

[5] Paul Newson and John Krumm. 2009. Hidden Markov Map Matching Through Noise and Sparseness. In *17th ACM SIGSPATIAL GIS.*

Appendix B Jyputer Notebook General API Usage

MapAPI tutorial

Table of Contents:

Setup

In [1]:
```
# Import requirements for later use
import requests
from matplotlib import pyplot as plt
import pandas as pd
import folium
```

First we define the API url and the API key. Your key can be found at http://mapapi.cs.aau.dk/home when you're logged in. Otherwise use this public key.

In [2]:
```
api_url = 'https://mapapi.cs.aau.dk/api/v1/'
key = 'WyIxIiwiJDUkcm91bmRzPTUzNTAwMCRDd3JSSjNmWkozUG9lOURuJEViTkJ3TnR6THRSV2J5Mnk4TzNqYlI
```

We define some example lat-lon points to demonstrate the API.

In [3]:
```
# points used for querying later
aalborg_university = '57.01755,9.97231'
shopping_center = '57.00137,9.87148'
horse_tracks = '57.0545,9.88105'
airport = '57.08639,9.87259'
harbor = '57.06002,9.95453'
```

```
aalborg = '57.04633,9.91902'
viborg = '56.44781,9.39786'
aarhus = '56.15058,10.20421'

otterup = '55.5142,10.3988'
odense = '55.41797,10.41913'
svendborg = '55.07025,10.61826'

e45_aalborg_tunnel = '57.0620805,9.9495412'
e45_th_sauers_vej = '57.0197823,9.9602263'

# areas for querying later
copenhagen_area = {'point1': '55.6214,12.4084', 'point2': '55.7837,12.6185'}
svendborg_area = {'point1': '55.0289,10.5216', 'point2': '55.0923,10.6415'}
aalborg_area = {'point1': '57.0099,9.8695', 'point2': '57.0963,9.9912'}
```

In [4]:

```
# Some endpoints accept a sequence of lat-lons separated by the pipe-symbol
# The lat-longs can be concatenated like this
aalborg_locations = '|'.join([aalborg_university, shopping_center, horse_tracks, airport,
jutland_route = '|'.join([aalborg, viborg, aarhus])
funen_route = '|'.join([otterup, odense, svendborg])
funen_route
```

Out[4]:

```
'55.5142,10.3988|55.41797,10.41913|55.07025,10.61826'
```

Travel Time Data Endpoints

Endpoints in this section contain data used for training some deep neural travel-time models of the following section.

We use the `requests` module to query the API. The endpoint parameters are defined in a python dictionary. You can check all the available parametern for each endpoint at https://mapapi.cs.aau.dk/apidocs/

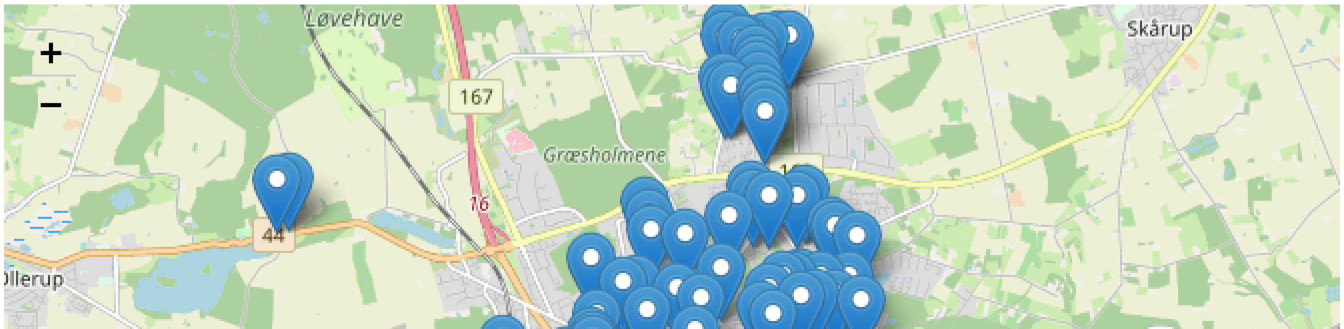
Crossings

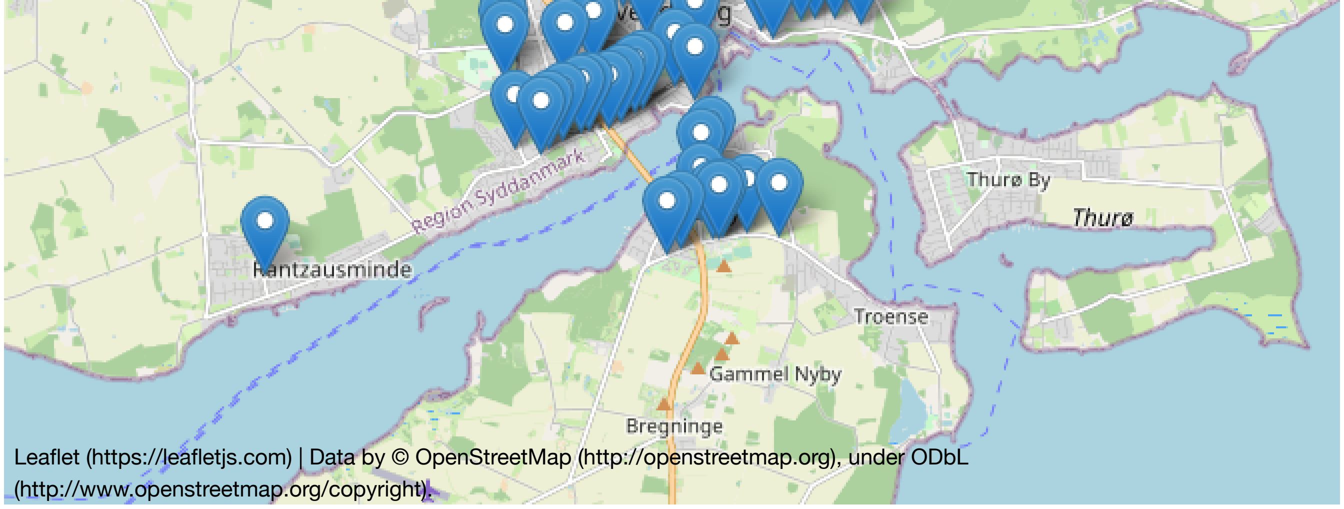
Returns all OSM crossings within a bounding box

In [5]:

```
cr_params = {
    'key': key,
    'country_code': 'dk'
}
cr_params = {**cr_params, **svendborg_area} # join crossings parameters with point1 and po
cr_response = requests.get(api_url + 'crossings', cr_params) # make the request

# We use module folium to display points and other geometry on a map
m = folium.Map(location=[55.057119, 10.606324], zoom_start=12) # Create the map and set th
for crossing in cr_response.json()['results']:
    folium.GeoJson(crossing['location']).add_to(m)
m
```

Out[5]:




Thurø By
Thurø
Region Syddanmark
Rantzausminde
Troense
Gammel Nyby
Bregninge
Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

All endpoints that return a geometry have a `geometry` parameter, which either accepts `'geojson'` or `'wkt'` as parameters. The default is `'geojson'` .

In [6]:
```
cr_params = {**cr_params, **{'geometry': 'wkt'}}  # Same parameters but with the geometrie
cr_response = requests.get(api_url + 'crossings', cr_params) # make the request
cr_response.json()['results'][:10]
```

Out[6]:
```
[{'location': 'POINT(10.6170641 55.0874809)'},
 {'location': 'POINT(10.606893 55.0387745)'},
 {'location': 'POINT(10.6126892 55.042249)'},
 {'location': 'POINT(10.6293288 55.0717479)'},
 {'location': 'POINT(10.5941398 55.0579636)'},
 {'location': 'POINT(10.5890377 55.0499668)'},
 {'location': 'POINT(10.5896126 55.056361)'},
 {'location': 'POINT(10.6212044 55.0726581)'},
 {'location': 'POINT(10.6307935 55.0722924)'},
 {'location': 'POINT(10.6161897 55.0646431)'}]
```

Segment Details

OpenStreetMap metadata for the segment closest to a point

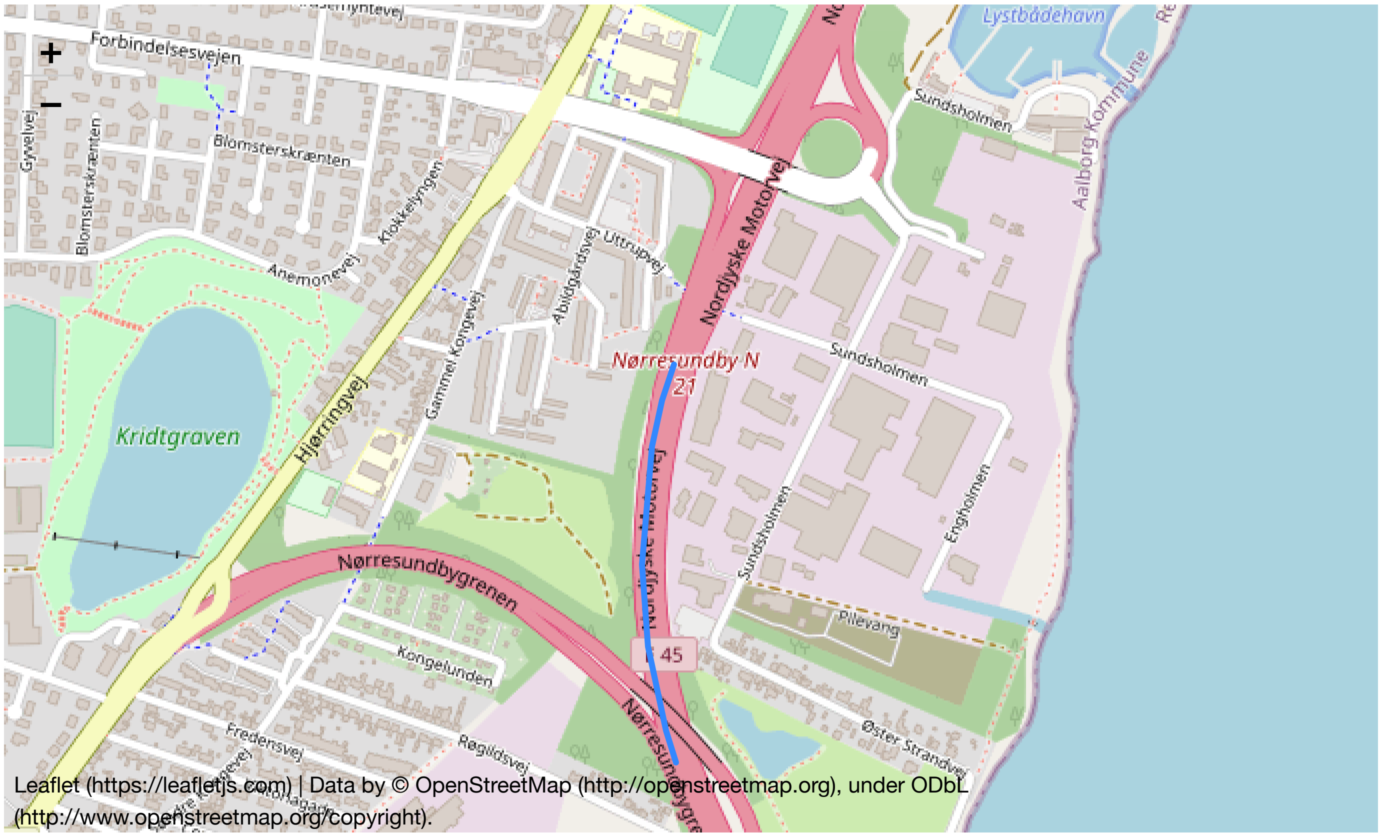
In [7]:
```
sd_params = {
    'key': key,
    'point': e45_aalborg_tunnel
}

sd_result = requests.get(api_url + 'segmentDetails', sd_params).json()['results'][0]
sd_result
```

Out[7]:
```
{'category': 'motorway',
 'direction': 'FORWARD',
 'geometry': {'coordinates': [[9.9494746, 57.0667182],
   [9.9492564, 57.0662983],
   [9.9490285, 57.0657448],
   [9.9489453, 57.0654302],
   [9.948853, 57.0648569],
   [9.9488165, 57.0643828],
   [9.9488522, 57.0638974],
   [9.9489549, 57.0634161],
   [9.9490952, 57.0629771],
   [9.9493396, 57.0624284],
   [9.9495482, 57.062063]],
  'type': 'LineString'},
 'key': 532483,
 'meters': 527,
 'name': 'Nordjyske Motorvej',
 'speedLimit': 110}
```

In [8]:

```
m = folium.Map(location=[57.0662983, 9.9492564], zoom_start=15)
folium.GeoJson(sd_result['geometry']).add_to(m)
m
```

Out[8]:



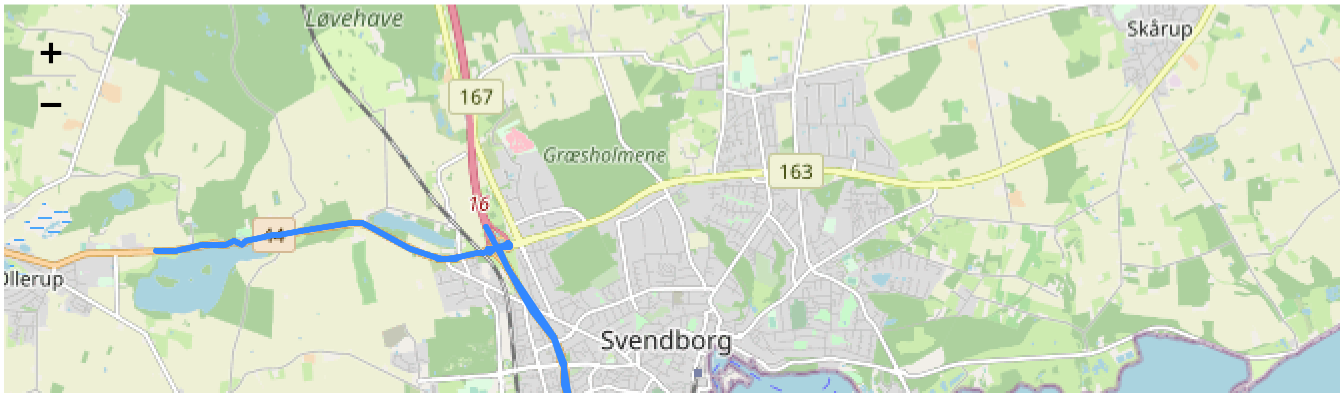
Segments

Get a list of road segments of a specific category within a bounding box. Available categories: https://mapapi.cs.aau.dk/apidocs/#/Travel%20Time/get_api_v1_segments

In [9]:

```
seg_params = {
    'key': key,
    'country_code': 'dk',
    'category': 'primary'
}
seg_params = {**seg_params, **svendborg_area}
seg_results = requests.get(api_url + 'segments', seg_params).json()['results']
m = folium.Map(location=[55.057119, 10.606324], zoom_start=12)
for seg in seg_results:
    folium.GeoJson(seg['geometry']).add_to(m)
m
```

Out[9]:



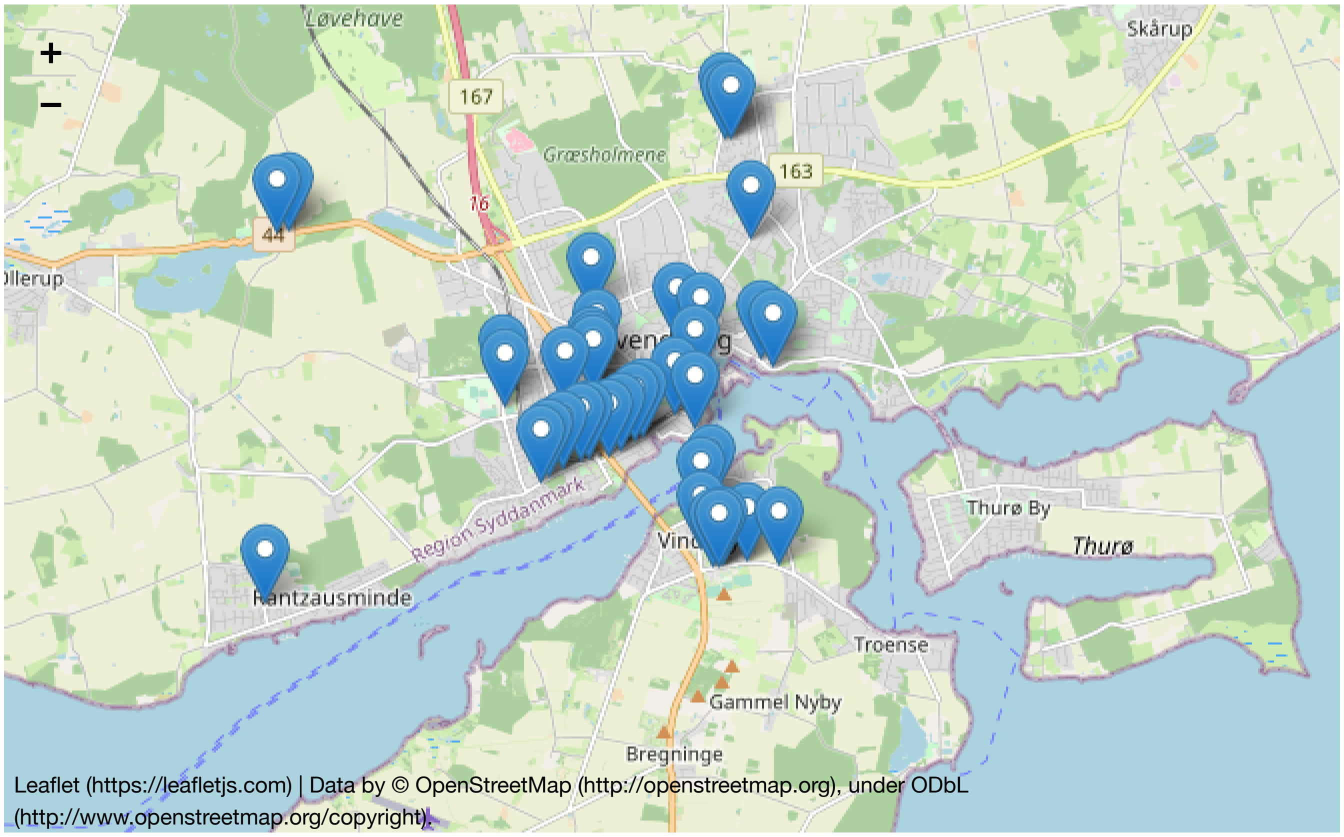
Region Syddanmark
Vindeby
Thurø By
Thurø
Rantzausminde
Troense
Gammel Nyby
Bregninge
Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

Traffic Calming

Returns all OSM traffic calming within a bounding box

In [10]:
```
tc_params = {
    'key': key,
    'country_code': 'dk',
}
tc_params = {**tc_params, **svendborg_area}
tc_results = requests.get(api_url + 'trafficCalming', tc_params).json()['results']
m = folium.Map(location=[55.057119, 10.606324], zoom_start=12)
for tc in tc_results:
    folium.GeoJson(tc['location']).add_to(m)
m
```

Out[10]:


+
−
Løvehave
Skårup
167
Græsholmene
163
16
44
Ollerup
Region Syddanmark
Thurø By
Thurø
Rantzausminde
Troense
Gammel Nyby
Bregninge
Leaflet (https://leafletjs.com) | Data by © OpenStreetMap (http://openstreetmap.org), under ODbL (http://www.openstreetmap.org/copyright).

Travel Time Model Endpoints

Endpoints in this section use estimated travel-times produced by deep neural models.

The endpoints in this section all have a travel time `'model_id'` parameter, which can either be `'A'`, `'B'`, `'C'` or `'D'`. Model A only use length, speed and category. Model B adds tortuosity and one-way. Model C adds traffic calmings and crossings. Model D adds temporal information. Model C is the default and recommended. Model D also needs a `timestamp` parameter to work to add temporal information, which should be an iso-format datetime.

Costmatrix

Create a cost matrix from a list of points

In [11]:

```
parameters = {
    'key': key,
    'country_code': 'dk',
    'points': aalborg_locations, # lat-lons seperated by '|'
    'model_id': 'C'
}

response = requests.get(api_url + 'costMatrix', parameters)
costmatrix = response.json()['results']
# On sucessful json responses the API always return a list called 'results'
costmatrix
```

Out[11]:

```
[{'cost': 540.85, 'source': 1, 'target': 2},
 {'cost': 645.83, 'source': 1, 'target': 3},
 {'cost': 771.04, 'source': 1, 'target': 4},
 {'cost': 446.93, 'source': 1, 'target': 5},
 {'cost': 515.84, 'source': 2, 'target': 1},
 {'cost': 611.7, 'source': 2, 'target': 3},
 {'cost': 885.2, 'source': 2, 'target': 4},
 {'cost': 779.94, 'source': 2, 'target': 5},
 {'cost': 666.88, 'source': 3, 'target': 1},
 {'cost': 612.4, 'source': 3, 'target': 2},
 {'cost': 579.94, 'source': 3, 'target': 4},
 {'cost': 568.56, 'source': 3, 'target': 5},
 {'cost': 808.53, 'source': 4, 'target': 1},
 {'cost': 876.42, 'source': 4, 'target': 2},
 {'cost': 576.67, 'source': 4, 'target': 3},
 {'cost': 543.8, 'source': 4, 'target': 5},
 {'cost': 539.5, 'source': 5, 'target': 1},
 {'cost': 757.0, 'source': 5, 'target': 2},
 {'cost': 594.89, 'source': 5, 'target': 3},
 {'cost': 581.93, 'source': 5, 'target': 4}]
```

In [12]:

```
# The costmatrix result returns the source and target indices
# This can be mapped to names representing the locations, and displayed with a pandas Data
location_names = ['university', 'shopping_center', 'horse_tracks', 'airport', 'harbor']
cm = [[0 for i in range(len(location_names))] for j in range(len(location_names))]  # Crea

# Fill in the 2D array
for entry in costmatrix:
    cm[entry['source'] - 1][entry['target'] - 1] = entry['cost']

df = pd.DataFrame(cm, columns=location_names, index=location_names)
df
```

Out[12]:

	university	shopping_center	horse_tracks	airport	harbor
university	0.00	540.85	645.83	771.04	446.93
shopping_center	515.84	0.00	611.70	885.20	779.94

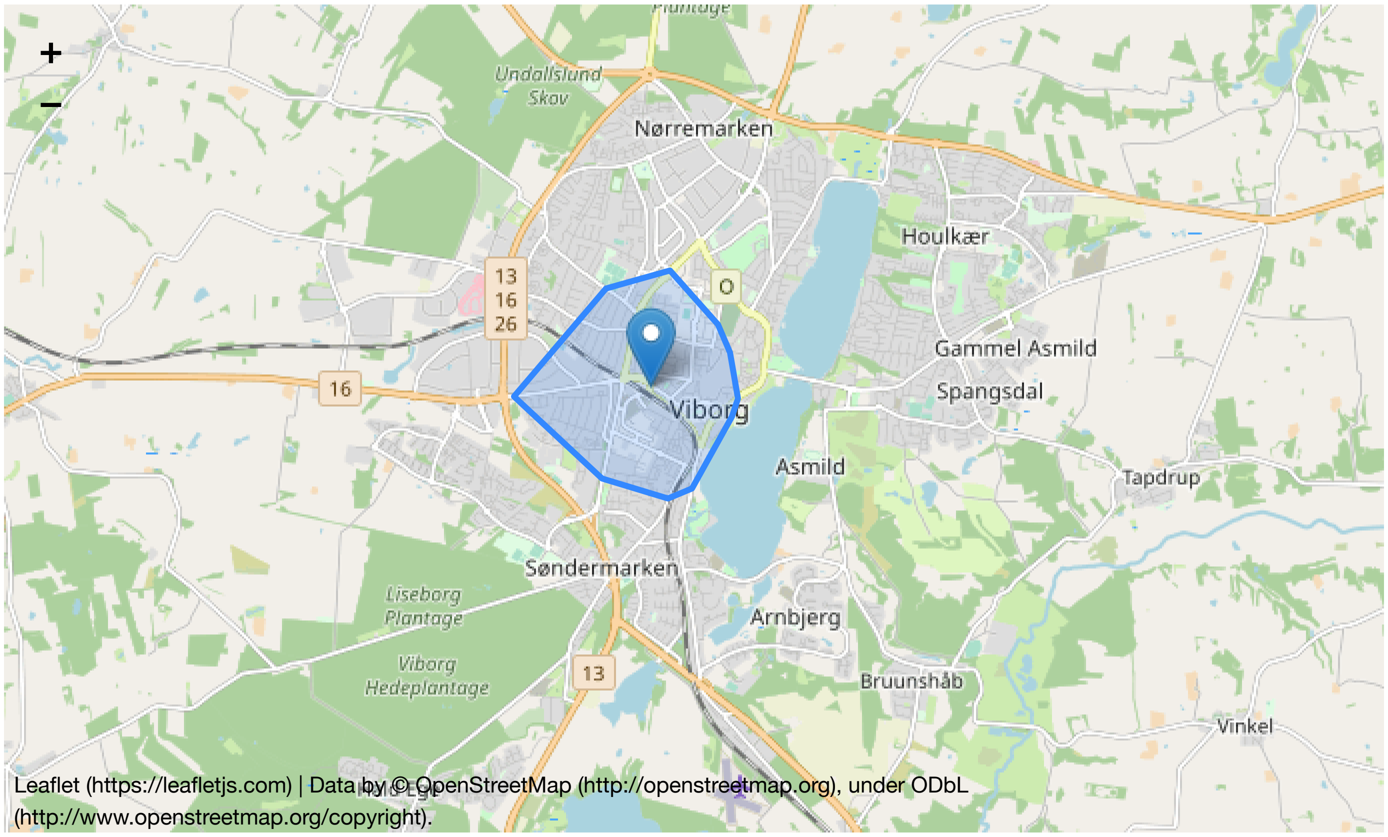
	university	shopping_center	horse_tracks	airport	harbor
horse_tracks	666.88	612.40	0.00	579.94	568.56
airport	808.53	876.42	576.67	0.00	543.80
harbor	539.50	757.00	594.89	581.93	0.00

Isochrone

Returns a polygon with the travel-time isochrone

In [13]:
```
iso_params = {
    'key': key,
    'country_code': 'dk',
    'point': viborg,
    'seconds': 120  # Number of seconds driven from the point
}

iso_results = requests.get(api_url + 'isochrone', iso_params).json()['results'][0]
m = folium.Map(location=[56.44781,9.39786], zoom_start=12)
folium.GeoJson(iso_results['polygon']).add_to(m)
folium.GeoJson(iso_results['center']).add_to(m)  # The start point snapped to a road
m
```

Out[13]:



Routing Travel Time

Finds the nearest road-segments using Euclidean distance. Uses the Dijkstra algorithm (the default). Finds the shortest route based on travel-time.

In [14]:
```
rtt_params = {
    'key': key,
    'country_code': 'dk',
    'points': jutland_route
}
rtt_result = requests.get(api_url + 'routingTravelTime', rtt_params).json()['results'][0]

print(round(rtt_result['time'], 2), 'seconds')
```

```
6532.8 seconds
```

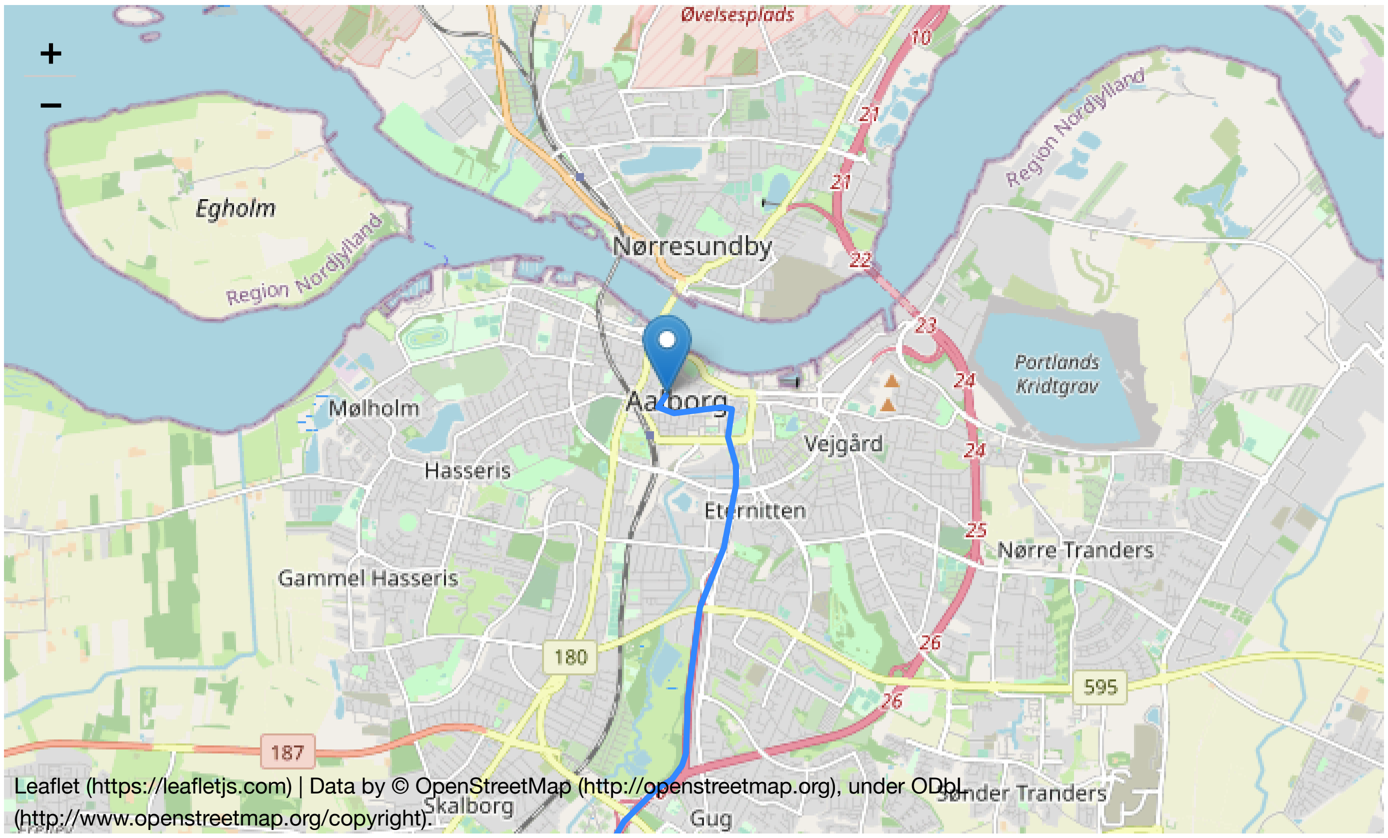
Rural Route

Returns the linestring of the shortest route and a list of snap-points

In [15]:

```
rr_params = {
    'key': key,
    'country_code': 'dk',
    'points': jutland_route
}
rr_result = requests.get(api_url + 'ruralRoute', rr_params).json()['results'][0]
m = folium.Map(location=[57.04633, 9.91902], zoom_start=12)
folium.GeoJson(rr_result['path']).add_to(m)
folium.GeoJson(rr_result['points']).add_to(m)
m
```

Out[15]:



Trajectory Based Travel Time Endpoints

Average Number of Observations

Gets the average number of GPS observations traversing a single segment.

In [16]:

```
ano_params = {
    'key': key,
    'point': aalborg
}
ano_result = requests.get(api_url + 'avgNoObservations', ano_params).json()['results'][0]
ano_result
```

Out[16]:
```
{'avgObservations': 18.85}
```

The following endpoints have a similar parameter signature that let's the user make queries in specific time frames.
`fromdate` and `todate` are date smartkeys that specifies the from and to date, defaults are `20120101` and `20130101`.
`fromtime` and `totime` are time smartkeys that specifies the from and to time, defaults are `0` and `2359`.
`days` is a day-of-week bit pattern, that specifies the days Monday to Sunday to query for, the default is `1111100` i.e. Monday to Friday.

Entry Speed

Gets minimum, average, and maximum speed from the first GPS point map-matched to the segment.

In [17]:
```
entry_params = {
    'key': key,
    'point': aalborg,
    'fromdate': 20120101,
    'todate': 20130101,
    'fromtime': 800,
    'totime': 1600,
    'days': '1111100'
}
es_result = requests.get(api_url + 'entrySpeed', entry_params).json()['results'][0]
es_result
```

Out[17]:
```
{'avg': 19.35, 'max': 55.0, 'min': 0.0}
```

Exit Speed

Gets minimum, average, and maximum speed from the last GPS point map-matched to the segment.

In [18]:
```
exit_params = {
    'key': key,
    'point': aalborg,
    'fromdate': 20120101,
    'todate': 20130101,
    'fromtime': 800,
    'totime': 1600,
    'days': '1111100'
}
exit_result = requests.get(api_url + 'exitSpeed', exit_params).json()['results'][0]
exit_result
```

Out[18]:
```
{'avg': 15.75, 'max': 40.0, 'min': 2.0}
```

Most Used Route

Returns a linestring representation of the most used route.

In [19]:
```
mur_params = {
    'key': key,
    # Øster Alle, Kridtsvinget towards Aalborg, to Karolinelundsvej
    'frompoint': '57.04786,9.95181',
    'topoint': '57.04635, 9.93434',
    'fromdate': 20120101,
    'todate': 20130101,
    'fromtime': 800,
    'totime': 1600,
    'days': '1111100'
}
mur_result = requests.get(api_url + 'mostUsedRoute', mur_params).json()['results'][0]
mur_result
```

Out[19]:
```
{'route': {'coordinates': [[9.955336, 57.0502011],
   [9.9547055, 57.0497992],
   [9.9511408, 57.0474138],
   [9.9506017, 57.04712],
   [9.9501586, 57.0468455],
   [9.9499789, 57.0467888],
   [9.9496231, 57.0467102],
   [9.9494008, 57.0466954],
   [9.9490953, 57.0467025],
   [9.9459841, 57.0468169],
   [9.943462, 57.046787],
   [9.9349541, 57.0470342],
   [9.9345851, 57.0470145],
   [9.9343945, 57.0469963],
   [9.9343934, 57.0468984],
   [9.9344201, 57.0460941]],
  'type': 'LineString'}}
```

Segment Speed

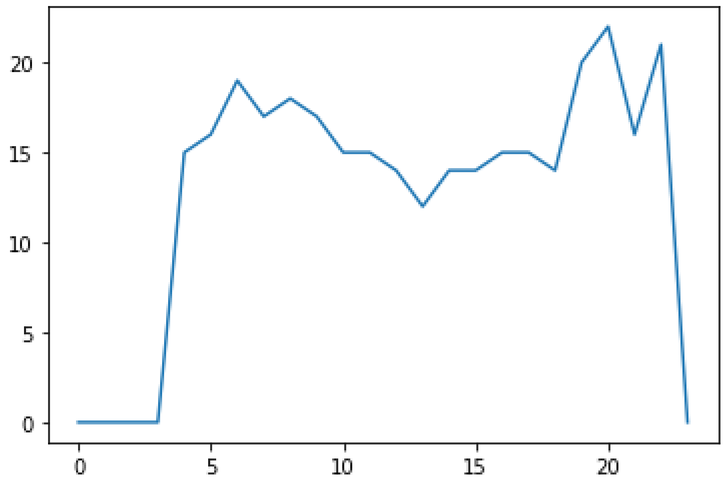
Gets the average travel-time on a segment over the day.

In [20]:
```
ss_params = {
    'key': key,
    'point': aalborg,
    'granularity': 60 # Can be 15 or 60. Default 15
}
ss_results = requests.get(api_url + 'segmentSpeed', ss_params).json()['results']
ss_results # The result is a list of dicts with a speed and time key
```

Out[20]:
```
[{'speed': 0, 'time': '0'},
 {'speed': 0, 'time': '1'},
 {'speed': 0, 'time': '2'},
 {'speed': 0, 'time': '3'},
 {'speed': 15, 'time': '4'},
 {'speed': 16, 'time': '5'},
 {'speed': 19, 'time': '6'},
 {'speed': 17, 'time': '7'},
 {'speed': 18, 'time': '8'},
 {'speed': 17, 'time': '9'},
 {'speed': 15, 'time': '10'},
 {'speed': 15, 'time': '11'},
 {'speed': 14, 'time': '12'},
 {'speed': 12, 'time': '13'},
 {'speed': 14, 'time': '14'},
 {'speed': 14, 'time': '15'},
 {'speed': 15, 'time': '16'},
 {'speed': 15, 'time': '17'},
 {'speed': 14, 'time': '18'},
 {'speed': 20, 'time': '19'},
 {'speed': 22, 'time': '20'},
 {'speed': 16, 'time': '21'},
 {'speed': 21, 'time': '22'},
 {'speed': 0, 'time': '23'}]
```

In [21]:

```
# The speed time series can be plotted with matplotlib
time_intervals = [int(s['time'].replace(':', '')) for s in ss_results]
speed_intervals = [s['speed'] for s in ss_results]
plt.plot(time_intervals, speed_intervals)
plt.show()
```



spqFuelEstimate

Gets the average total fuel estimates on a route for trips/trajectory.

In [22]:

```
spq_fe_params = {
    'key': key,
    'frompoint': e45_aalborg_tunnel,
    'topoint': e45_th_sauers_vej,
    'fromdate': '20120101',
    'todate': '20120601',
    'fromtime': 800,
    'totime': 1600
}
spq_fe_result = requests.get(api_url + 'spqFuelEstimate', spq_fe_params).json()['results']
spq_fe_result
```

Out[22]:

```
{'length': 5609, 'sidraEstimate': 0.72, 'spEstimate': 2338.53, 'trips': 32}
```

spqFuelEstimateList

Gets the total fuel estimates on a per trip bases for a route.

In [23]:

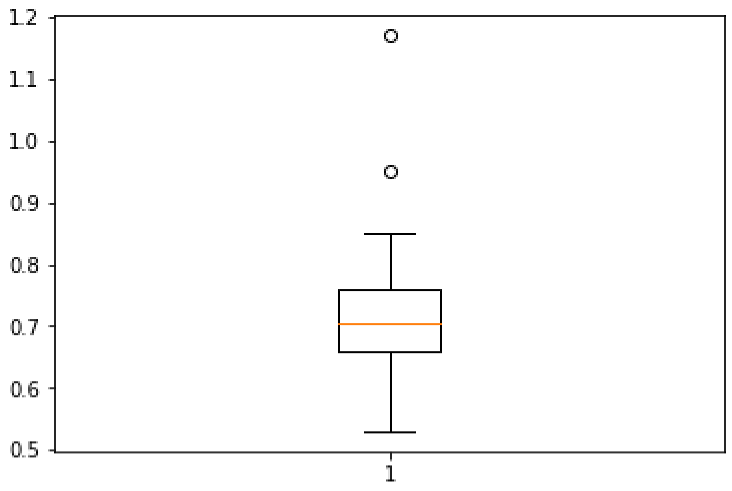
```
spq_fel_result = requests.get(api_url + 'spqFuelEstimateList', spq_fe_params).json()['resu
spq_fel_result[:10]
```

Out[23]:

```
[{'datetime': 'Mon, 30 Apr 2012 10:00:00 GMT',
  'duration': 238,
  'sidraEstimate': 0.6,
  'spEstimate': 1871.09},
 {'datetime': 'Mon, 30 Apr 2012 14:00:00 GMT',
  'duration': 214,
  'sidraEstimate': 0.69,
  'spEstimate': 2337.2},
 {'datetime': 'Tue, 01 May 2012 08:00:00 GMT',
  'duration': 248,
  'sidraEstimate': 0.53,
  'spEstimate': 1539.98},
 {'datetime': 'Tue, 01 May 2012 13:00:00 GMT',
  'duration': 185,
  'sidraEstimate': 0.79,
  'spEstimate': 2427.86},
 {'datetime': 'Wed, 02 May 2012 15:00:00 GMT',
  'duration': 167,
  'sidraEstimate': 0.95,
  'spEstimate': 3102.27},
 {'datetime': 'Fri, 04 May 2012 08:00:00 GMT',
  'duration': 226,
  'sidraEstimate': 0.57,
  'spEstimate': 1705.5},
 {'datetime': 'Fri, 04 May 2012 10:00:00 GMT',
  'duration': 188,
  'sidraEstimate': 0.82,
  'spEstimate': 2818.53},
 {'datetime': 'Fri, 04 May 2012 16:00:00 GMT',
  'duration': 181,
  'sidraEstimate': 0.79,
  'spEstimate': 2582.26},
 {'datetime': 'Mon, 21 May 2012 15:00:00 GMT',
  'duration': 202,
  'sidraEstimate': 0.68,
  'spEstimate': 2147.37},
 {'datetime': 'Tue, 22 May 2012 10:00:00 GMT',
  'duration': 199,
  'sidraEstimate': 0.75,
  'spEstimate': 2662.79}]
```

Create a boxplot of sidra estimates

In [24]:
```
sidraEstimate = list(map(lambda x: x['sidraEstimate'], spq_fel_result))
plt.boxplot(sidraEstimate)
plt.show()
```



spqTravelTime

Gets the fastest travel-time on a route specified by the two points.

In [25]:
```
spq_tt_params = {
    'key': key,
    'frompoint': e45_aalborg_tunnel,
    'topoint': e45_th_sauers_vej,
    'fromdate': '20120101',
    'todate': '20120601'
}
spq_tt_result = requests.get(api_url + 'spqTravelTime', spq_tt_params).json()['results'][(
spq_fe_result
```

```
Out[25]: {'length': 5609, 'sidraEstimate': 0.72, 'spEstimate': 2338.53, 'trips': 32}
```

spqTravelTimeList

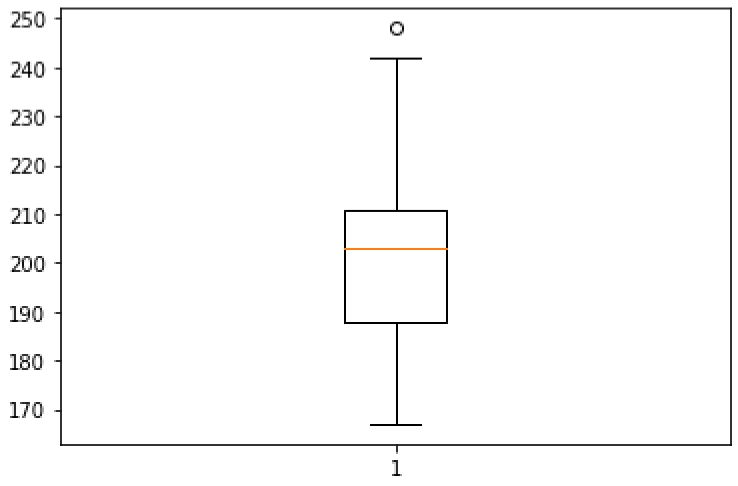
Gets the travel-time for each trip/trajectory on a route specified by the two points.

```
In [26]:
spq_ttl_results = requests.get(api_url + 'spqTravelTimeList', spq_tt_params).json()['resul
spq_ttl_results[:10]
```

```
Out[26]:
[{'datetime': 'Mon, 30 Apr 2012 10:00:00 GMT', 'duration': 238},
 {'datetime': 'Mon, 30 Apr 2012 14:00:00 GMT', 'duration': 214},
 {'datetime': 'Tue, 01 May 2012 08:00:00 GMT', 'duration': 248},
 {'datetime': 'Tue, 01 May 2012 13:00:00 GMT', 'duration': 185},
 {'datetime': 'Wed, 02 May 2012 15:00:00 GMT', 'duration': 167},
 {'datetime': 'Fri, 04 May 2012 08:00:00 GMT', 'duration': 226},
 {'datetime': 'Fri, 04 May 2012 10:00:00 GMT', 'duration': 188},
 {'datetime': 'Fri, 04 May 2012 16:00:00 GMT', 'duration': 181},
 {'datetime': 'Mon, 21 May 2012 15:00:00 GMT', 'duration': 202},
 {'datetime': 'Tue, 22 May 2012 10:00:00 GMT', 'duration': 199}]
```

Create boxplot of travel times

```
In [27]:
durations = list(map(lambda x: x['duration'], spq_ttl_results))
plt.boxplot(durations)
plt.show()
```



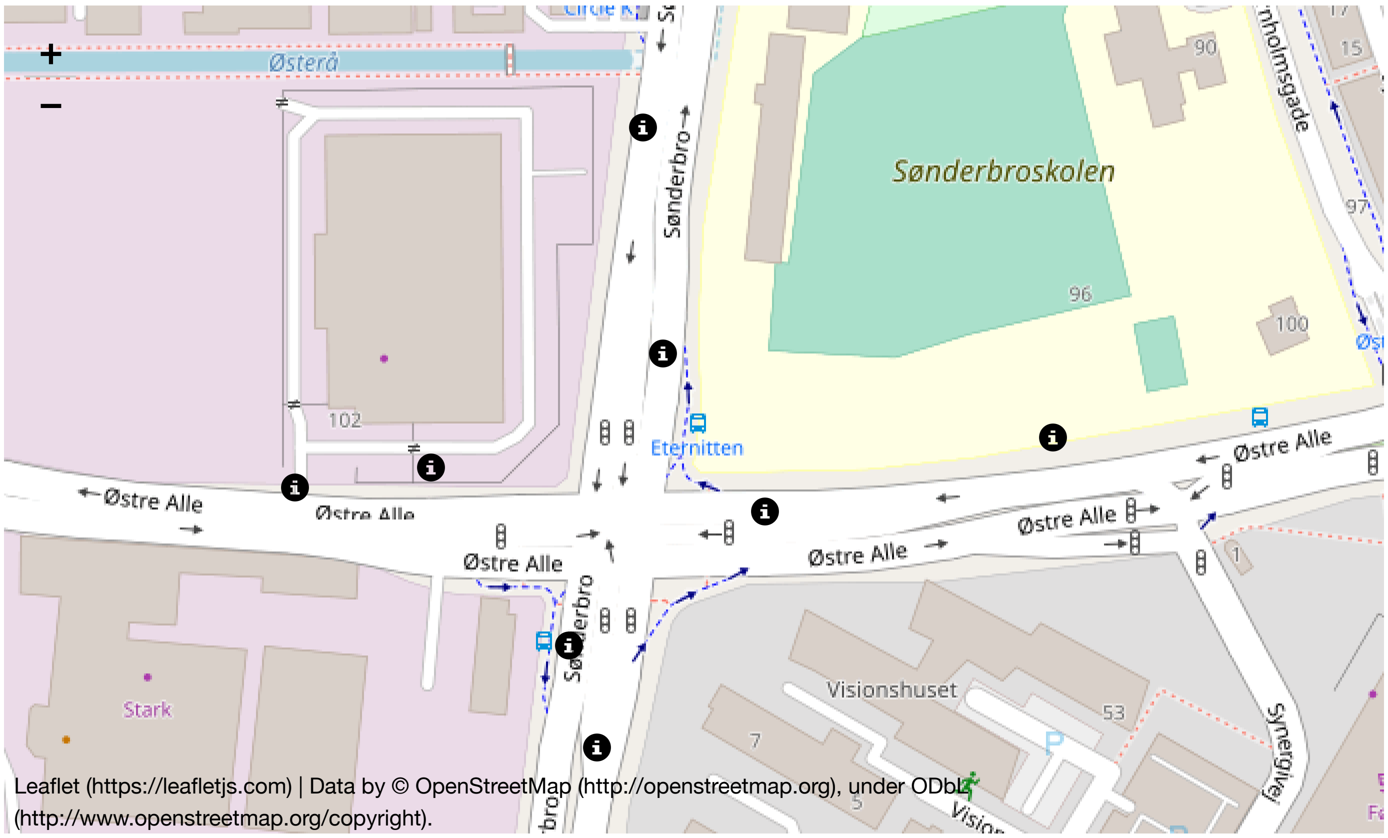
Appendix C Jyputer Travel Time in Intersections

Travel Times in an intersection

In this example we use the spqTravelTime endpoint, to analyse the how the travel time in an intersection is different going left, right or straight.

In [1]:
```python
import itertools
import requests
import folium
import pandas as pd

api_url = 'https://mapapi.cs.aau.dk/api/v1/spqTravelTime'
key = 'WyIxIiwiJDUkcm91bmRzPTUzNTAwMCRDd3JSSjNmWkozUG9lOURuJEViTkJ3TnR6THRSV2J5Mnk4TzNqYl

# Define the lon-lat pairs of each directions from and to road coordinates
north = {'direction': 'north', 'from': (57.0387472,9.9316021), 'to': (57.0380883,9.931700
south = {'direction': 'south', 'from': (57.0369342,9.9313483), 'to': (57.0372342,9.931205
west = {'direction': 'west', 'from': (57.037692,9.9297273), 'to': (57.0377534,9.9304561)}
east = {'direction': 'east', 'from': (57.0378398,9.9337937), 'to': (57.037626,9.9322534)}

# Visualize the from and to coordinates
m = folium.Map(location=[57.0380883,9.9317002], zoom_start=17)
all_points = [north, south, west, east]
for point in all_points:
    folium.Marker(location=point['from'], icon=folium.Icon(color='blue'), popup=f'From {p
    folium.Marker(location=point['to'], icon=folium.Icon(color='red'), popup=f'To {point['
m
```

Out[1]:



In [2]:
```python
directions = ['north', 'south', 'east', 'west']
cm = [[0 for i in range(len(directions))] for j in range(len(directions))]  # Create 2D a


# Loop over all pair-wise permutations of the cardinal directions
for p_from, p_to in itertools.permutations(all_points, 2):
    # Create the spqTravelTime parameters
    params = {
        'key': key,
        'frompoint': str(p_from['from']).replace('(','').replace(')',''),
        'topoint': str(p_to['to']).replace('(','').replace(')',''),
        'fromdate': 20110101,
        'todate': 20160101
    }

    # Make the requests
    res = requests.get(api_url, params)

    # Fill in the result matrix
    cm[directions.index(p_from['direction'])][directions.index(p_to['direction'])] = res.

# Display the result matrix as a dataframe
df = pd.DataFrame(cm, columns=directions, index=directions)
df
# The column is the from direction, the row is the to direction
```

Out[2]:

	north	south	east	west
north	0	24	36	28
south	31	0	38	23
east	32	30	0	26
west	35	25	35	0