Norwegian University of
Science and Technology

Using *Cooperative Intelligent Transport Systems* for Real-Time Determination of Dangerous Locations in Traffic

Lars Hellesylt

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Tor Engebret Onshus, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Abstract

A prototype solution for continuously determining the current level of danger in traffic at different locations has been developed, based on event detection on a stream of Cooperative intelligent transport systems (C-ITS) data. Detection of abrupt braking events from Cooperative awareness messages (CAMs) was chosen in order to provide a proof-of-concept of the solution, and an algorithm for the detection of such events has been constructed. Datasets were generated in real-life traffic in order to tune and evaluate the algorithm. This shows potential, but the algorithm suffers from a lack of larger volumes of data that is needed in order to train it for higher accuracy.

Abrupt braking events are combined with pre-detected Decentralized environmental notification message (DENM) events through a system of relative weighting based on a measure of event severity. The spatial and temporal characteristics of traffic events have been modeled mathematically, providing a way to express their effect dynamically on a map in real-time.

The full solution has been implemented in Python, as a series of modules that effectively divide the full problem into intuitive subtasks that can be developed and tested independently. The solution regularly outputs a list of dangerous locations and their calculated level of danger, which can be used as an input to other systems. Additionally, a way to visualize the incoming data, important intermediary results and the final output has been implemented, which, among other things, allows a human operator a clear view of the current level of danger in traffic at all times.

Sammendrag

Det har blitt utviklet en prototypeløsning som kontinuerlig beregner det gjeldende farenivået i trafikken på ulike lokasjoner, basert på hendelsesdeteksjon fra en strøm av data fra *Kooperative intelligente transportsystemer* (C-ITS). Detektering av bråbremsing fra CAM-meldinger ble valgt for å gi et konseptbevis av løsningen, og en algoritme for deteksjon av slike hendelser ble konstruert. Data ble generert i ekte trafikksituasjoner for å kunne justere og evaluere algoritmen. Dette viser potensiale, men algoritmen lider av en mangel på et større volum av data, som trengs for å trene den for høyere nøyaktighet.

Detekterte bråbremsinger kombineres med pre-detekterte DENM-hendelser gjennom et system for relativ vekting basert på et mål av alvorlighetsgrad. Trafikkhendelsers romlige og tidsmessige egenskaper ble matematisk modellert, slik at effekten deres kan uttrykkes på et kart i sanntid.

Den komplette løsningen er implementert i Python, som en serie moduler som effektivt deler problemet i mindre, intuitive del-oppgaver som kan utvikles og testes uavhengig av hverandre. Løsningen produserer jevnlig en liste over farlige lokasjoner og deres beregnede farenivå, som kan brukes som inngangsdata til andre systemer. I tillegg har det blitt utviklet en måte å visualisere den innkommende dataen, viktige mellomresultater og de endelige resultatene, som blant annet gir en menneskelig operatør full oversikt over det nåværende farenivået i trafikken til enhver tid.

Preface

This master's thesis is the culmination of five years of studies in Cybernetics and Robotics at the Norwegian University of Science and Technology (NTNU) in Trondheim. The project performed in the autumn of 2017 [1] forms the basis on which this thesis is founded, however, this is not a direct continuation of the project work. While the goal of the project work was to construct a solution for storage of large amounts of C-ITS data, the goal of the present work is to *use* such data in a specific application. Furthermore, the big data storage and analysis solution constructed in the project work is not used here. As such, the main contribution of the project work is in the experiences and lessons learned concerning the C-ITS subject.

The work has been performed in cooperation with Aventi Intelligent Communication, who formulated the original problem description found in appendix A. This has been somewhat modified by me, to better fit the present technical possibilities and incorporate my experiences from the project work. This resulted in the text of section 3, which has been approved by Aventi and my supervisor at NTNU.

Aventi has supported me with the hardware and software described in section 2.3, as well as counseling throughout the work. I am grateful to Bjørn Elnes and Terje Hundere at Aventi for providing me with this opportunity to immerse myself in an exciting and futuristic subject matter.

A big thank you to Tor Onshus, my supervisor at the Department of Engineering Cybernetics at NTNU, for a year's worth of guidance during the project and master's work.

And curse the weather gods of May for offering absolutely no help with writing this thesis.

Contents

Acronyms **6**

List of Figures **7**

List of Tables **9**

1 Introduction **10**

2 Background and theory **11**
2.1 Cooperative Intelligent Transport Systems 11
2.1.1 Purpose 11
2.1.2 Technical specification 12
2.2 Theoretical foundation 14
2.2.1 Vehicle braking, normal and abrupt 14
2.2.2 Event detection 17
2.2.3 Calculating distance on Earth 21
2.3 Equipment 22

3 Problem description **24**

4 Solution **26**
4.1 Method 26
4.2 Scope 30
4.3 Generating data 32
4.4 Utility functions 34
4.4.1 Preprocessing 34
4.4.2 DENM generator 35
4.4.3 Replaying and labeling a dataset 36
4.5 Module design and implementation 40
4.5.1 Detection 41
4.5.2 DENM handling 58
4.5.3 Weighting 60
4.5.4 Clustering 62
4.5.5 Visualization 68

5 Results and discussion **71**
5.1 Modules . . . 71
5.1.1 Detection . . . 71
5.1.2 Clustering . . . 75
5.1.3 Visualization . . . 77
5.2 Overall system . . . 78

6 Further work **80**
6.1 Detection . . . 80
6.2 Weighting . . . 81
6.3 Clustering . . . 81
6.4 Visualization . . . 82

References **83**

A Original problem description **86**

B File structure of attached digital material **90**

C Acceleration unit conversion **91**

Acronyms

C-ITS Cooperative intelligent transport systems

CAM Cooperative awareness message

CSV Comma-separated values

DENM Decentralized environmental notification message

ETSI European Telecommunications standards institute

FIFO First in, first out

ITS Intelligent transport systems

IUGG International Union of Geodesy and Geophysics

JSON JavaScript Object Notation

OBD On-board diagnostics

OBU Onboard unit

OSM OpenStreetMap

RSU Roadside unit

UML Unified modeling language

V2I Vehicle-to-infrastructure

V2V Vehicle-to-vehicle

V2X Vehicle-to-everything

List of Figures

1 Ideal profile for normal braking 16
2 Precision and recall 20
3 Kapsch EVK-3300 V2X Evaluation kit 22
4 The general structure of the ecosystem in which the solution operates 27
5 System modularization 30
6 A sample vehicle path 33
7 Path fully covered by Roadside units (RSUs) 34
8 Map view on OpenStreetMap (OSM) 37
9 Map view in empty space 38
10 Speed view 38
11 Heading view 39
12 Normal braking at 40 km/h 42
13 Raw data examples of normal and abrupt braking 43
14 Excerpt of filtered acceleration data 44
15 Determining the optimal filter window size and deceleration 46
17 The input and output of the detection module 47
16 Running the detection algorithm on the training dataset . . 48
18 The inner workings of the detection module 49
19 The `vehicle` class 52
20 The inner workings of the `Vehicle.new_data()` function . . . 53
21 The `TrafficEvent` class 57
22 System modularization with the DENM parser module . . . 59
23 The input and output of the weighting module 60
24 The input and output of the clustering module 62
25 The spatial impact of an event 63
26 The temporal impact of an event 64
27 The inner workings of the clustering module 67
28 The real-time speed and acceleration view 69
29 The real-time map view 70
30 Running the detection algorithm on the test dataset 72
31 The detection module in real-time operation 74
32 The detection module in real-time operation, using the alternative filtering method 75

33 The clustering module in real-time operation, with randomly generated DENM events 76
34 Overlapping events, increasing the level of danger 77
35 Data propagation through the system 79

List of Tables

1 Comfort at different deceleration levels 15
2 Algorithm for detecting abrupt braking events 45
3 Detection algorithm performance while tuning parameters . 47
4 Lookup table from event type to severity weighting 61
5 Optimal detection algorithm parameters for the test dataset 72

1 Introduction

Driving a vehicle is largely about continuously keeping track of the situation around you, reading the intent of others as best possible and making quick decisions if something unexpected happens. As our vehicles are now transforming into highly advanced computers, and over time learning how to communicate directly with one another and with infrastructure along the road, both the vehicles, their drivers and external traffic managers will be enriched with a vast amount of quality data, allowing them to keep a finger on the pulse of traffic like never before.

With Intelligent transport systems (ITS), the digital revolution will soon hit the roads for full. Listening to and analyzing the increasing volume of chatter between vehicles will provide us with an unprecedented insight into both the long-term evolution of traffic, and enable a continuously updated, real-time overview of the roads. Developing a system that uses this data in order to continuously determine the current level of danger at locations on a map is the subject of this thesis. This information could be valuable to both traffic controllers and emergency dispatchers, or be fed back to the vehicles themselves, in order to give them a heads-up about an upcoming critical point in their path.

Digital material, such as generated datasets and the written software, will be referenced throughout the thesis. This is included in a digital folder that should come attached, and the file structure of this material is represented in appendix B.

2 Background and theory

The entirety of section 2.1 is directly quoted from [1].

2.1 Cooperative Intelligent Transport Systems

ITS is an umbrella term for the digitization of all forms of transport, "in which information and communication technologies are applied in the field of transport, including infrastructure, vehicles and users, and in traffic management, as well as for interfaces with other modes of transport", as defined in [2]. At the time of writing, different aspects of ITS are under active development as ETSI standards in cooperation with the European Commision, car makers, network operators, electronics vendors and others. Their work can be followed at [3]. In 2013, the Norwegian government presented ITS as a focus area in the future of transport [4].

2.1.1 Purpose

Cooperative intelligent transport systems (C-ITS) is the branch of ITS that aims to connect traffic in a wireless communications network, with the goal of increasing safety and efficiency through information sharing and cooperation. Vehicle-to-vehicle (V2V) communication will allow vehicles to broadcast basic variables such as location and speed to all nearby vehicles, giving everyone a more complete overview of the traffic around them at all times. Notable events such as accidents, slippery or blocked roads or traffic congestion can be relayed from vehicle to vehicle, out to everyone in the relevant area. Emergency vehicles can warn the traffic in its path in an earlier and more controlled manner.

To begin with, information exchanged through C-ITS will be processed and presented to the driver through existing interfaces such as icons on the navigation map and other visual or auditory warnings. As vehicles become more automated, C-ITS could open the possibility of more complex communication, such as broadcasting of future intent, and vehicles coordinating their behavior through an intersection instead of relying on rigid and inefficient rules such as traffic lights.

Vehicle-to-infrastructure (V2I) communication will allow vehicles to exchange information with infrastructure. This can for instance be used

for automated tolling and parking, tunnels that can warn entering vehicles about a fire inside, speed limit notifications, weather stations that can calculate the road's coefficient of friction and inform passing vehicles, and traffic lights that broadcast the remaining duration of its red or green lights, such that vehicles can calculate the optimal speed for energy efficient and comfortable cruising through the intersection.

V2V, V2I and other variants are collectively termed V2X.

2.1.2 Technical specification

ETSI's C-ITS standards rely on a protocol termed ITS-G5 [5] which specifies 5.9 GHz as the frequency band for operation. ITS-G5 is based on the same underlying specifications as wireless communication using WiFi (namely, the IEEE 802.11 series of standards). Unlike WiFi, ITS-G5 allows communication without the creation of a central access point, and thus without any authentication/association ("pairing") between stations. Messages are broadcasted in an all-to-all/fully connected network topology, such that messages can be received by anyone being sufficiently close to the sender. The range is mandated to be at least 300 m when line of sight can be established and there is little communication channel congestion [6]. Moreover, important messages can reach even further by being relayed through a chain of vehicles.

Two different classes of messages are specified for normal traffic:

1. Cooperative awareness message (CAM) [7]: Broadcasted by every vehicle at a frequency of between 1 and 10 Hz. It contains at the very least a *high-frequency container* with the latest data about the location, speed, acceleration, and heading (direction) of the vehicle, among other things. Occasionally it will also include a *low-frequency container* with information about the vehicle which is static or not highly dynamic. Additionally, special vehicles such as emergency or road works vehicles or vehicles doing public transport operations will include a *special vehicle container* with relevant information about their operation.

   Received messages can be used to build a detailed, constantly updating map of the surrounding vehicles which can be presented to

the driver, or which the vehicle can act on directly (for instance to support the cruise control, or perform emergency braking).

2. Decentralized environmental notification message (DENM) [8]: Generation is triggered by an event, such as the detection of a road hazard or abnormal traffic conditions. The message contains information about the event, as well as its location and time of detection. The message is sent V2V or V2I (in which case it can be relayed back to other vehicles, or sent via the internet to for instance a traffic control center or emergency personnel). A received DENM may be presented to the driver as a visual or auditory warning, or trigger automatic response by the vehicle.

2.2 Theoretical foundation

2.2.1 Vehicle braking, normal and abrupt

For the purposes of detecting an *abrupt braking event* (to be relevant in sections 4.2 and 4.5.1), it is useful to formulate what distinguishes such an event from a *normal braking event.*

Braking, in general, is the act of applying the brakes of a vehicle through a foot pedal in order to slow the vehicle down, or *decelerate*[1]. We will use *normal braking* for a deceleration which is:

Safe, in that it is performed in a controlled, calm and planned manner, not in haste, and without losing control of the vehicle or the situation around it. It achieves the goal of decelerating the vehicle to the target speed in time.

Comfortable, in that the braking event is not unpleasant for the driver or any passengers.

As comfort is here a stricter criterion than safety, as well as the fact that one could argue that the driver's comfort is itself a requirement for safety, we focus on this last point. One measure of comfort level is the peak deceleration experienced during the event[2]. According to [9], a deceleration of up to at least 2.0 m/s^2 is comfortable, while [10] states 3.4 m/s^2 as a "comfortable deceleration for most drivers". In [11], measured data for the comfort at different levels of deceleration are given, which are reproduced in table 1. As these levels are for average deceleration, they are certainly lower than the maximum deceleration experienced during the braking. The literature thus mostly agrees, and seems to indicate that a change from comfort to discomfort occurs somewhere between 3 and 4 m/s^2.

[1]The term *deceleration* is used throughout this report both as "the act of slowing down", and as the mathematical negative of acceleration.

[2]In this section, and throughout the thesis, acceleration values will frequently be given in the unit of m/s^2. For humans used to traveling by car, it can be easier to think in terms of the more complex $km/h/s$, as for instance an acceleration of 10 $km/h/s$ means "increasing speed by 10 km/h over a period of 1 second". Appendix C provides a quick way to convert between the two.

Table 1: Data from [11] on the comfort at different average deceleration levels. Decelerations were performed from a speed of 112 km/h to a standstill. The tests themselves were done by [13], unfortunately only with eight test subjects.

Evaluation	**Average deceleration** (m/s^2)
Comfortable to passenger, preferred by driver.	2.6
Undesirable, but not alarming to passenger; driver would rather not use.	3.4
Severe and uncomfortable to passengers, slides objects off seats. Driver classes as an emergency stop.	4.2
Maximum stop, car stays in a 12-foot lane without skidding. Brakes in best condition.	5.9

The smoothness of the deceleration is raised by [12] as another concern for comfort, and the minimization of *jerk* (the rate of change of acceleration) is used to ensure a smooth deceleration.

Based on these observations, an ideal, theoretical braking profile can be constructed as shown in figure 1. The jerk is here constructed as piecewise linear segments, with no discontinuities or "jumps". This produces a smoothly varying deceleration profile with a maximum deceleration of approximately 2.2 m/s^2.

Abrupt braking, thus, can be said to be a deceleration that deviates sufficiently from the ideal profile, in that it is more forceful (higher maximum deceleration) and/or less smooth (higher jerk), both of which indicate a suddenness or abruptness of the maneuver.

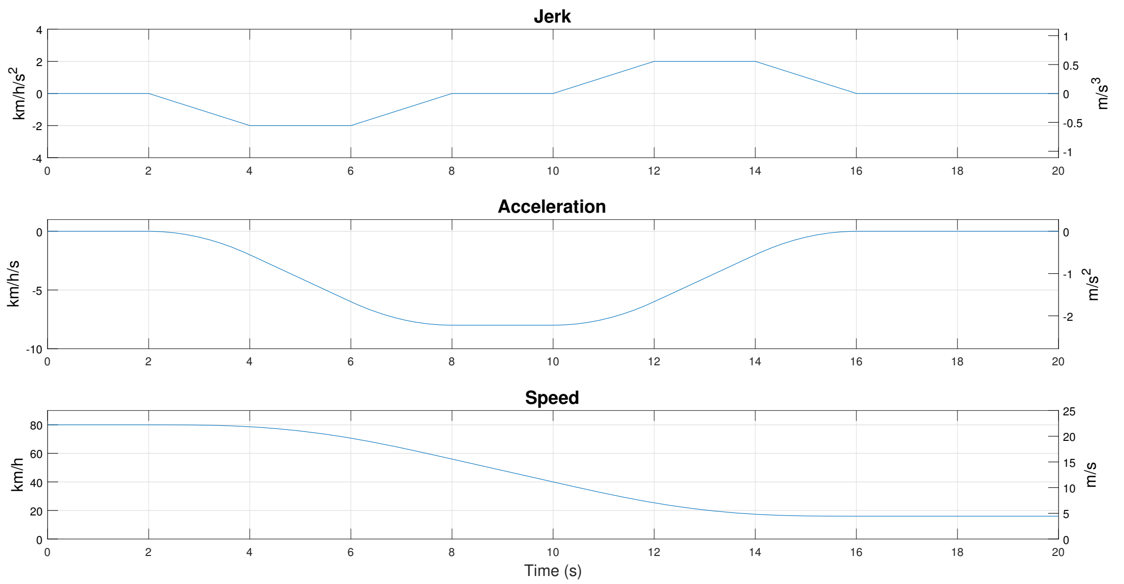


Figure 1: An ideal profile for normal braking.

2.2.2 Event detection

Consider a dataset consisting of a number of measurements of some process variable, distributed discretely in time, i.e. a number of *samples.* If an event is modeled as being instantaneous, such that it occurs at some discrete point in time corresponding to the timing of one of the samples, then the process of detecting events in the dataset entails picking a subset of the samples from the whole, and designating these as representing events. This process can be automated by designing an event detection algorithm.

Filtering the raw dataset can be included as a step in this algorithm, often used in order to reduce noise present in the data. The *moving average* is a type of filter that has a smoothing effect on the data, by exchanging each sample for an average of several samples in its neighborhood. There are two variants which are relevant here:

Simple moving average: A *window* with a certain size n is defined statically, and each sample is swapped with the unweighted mean of all the samples on its left and right side which lie within the filter, stretching $n/2$ number of samples in each direction.

Exponential moving average: Uses a weighted mean, where more recent values are given more weight than distant ones, in an exponentially decreasing fashion. This is implemented using a recursive formula:

$$F_n = \begin{cases} R_0, & n = 0 \\ \alpha R_0 + (1 - \alpha)\, F_{n-1}, & n > 0 \end{cases} \tag{1}$$

where F_n is the n'th filtered value, R_n is the n'th raw value, and α is a smoothing factor that determines how fast the weights decay, as one moves further away from the most recent sample.

This can thus be better suited than the simple moving average for filtering a stream of data in real-time, where samples "to the right" (in the future) are unavailable, and due to the fact that it only requires a single value F_n to be retained over time.

Judging the performance of a detection algorithm, which is important for optimizing it, comparing different ones or building confidence in its detections, one can use a labeled dataset. This means that there exists a so-called *ground truth* designating what samples represent actual events, which is most often manually constructed by humans. We can then define the following terms, which are illustrated in figure 2:

Element: Here, each sample is an element, because every sample can potentially be picked to represent an event.

Relevant/positive elements: These are the samples that represent actual events, i.e. the ground truth.

Irrelevant/negative elements: The samples that do not represent actual events. These are implicitly part of the ground truth, by *not* being labeled as actual events.

Selected elements: The samples that the detection algorithm picks, i.e. the algorithm's best guess for what samples represent events.

True positives: The intersection between the positive elements and the selected elements, i.e. the subset of the selected events that were picked correctly.

False positives: The intersection between the negative elements and the selected ones, i.e. the selected samples that are "wrong".

False negatives: The actual events that failed to be picked by the detection algorithm.

True negatives: The irrelevant elements that the detection algorithm correctly did not pick.

As shown in the figure, one can now calculate two metrics in order to judge the performance of a detection algorithm:

$$\text{precision} = \frac{\text{true positives}}{\text{selected elements}} \tag{2}$$

$$\text{recall} = \frac{\text{true positives}}{\text{relevant elements}} \tag{3}$$

Precision thus indicates to what degree one can trust that the events picked by the detection algorithm are actual events, while *recall* tells us how often an actual event is detected as such by the algorithm, both of them taking on values between 0 and 1. These can be combined into a single metric called the *F-score*

$$\text{F-score} = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4}$$

which is thus a single value between 0 and 1 that can be used to judge the performance of a detection algorithm.

It is advantageous to use separate, labeled datasets for tuning an algorithm, called *training*, and for testing it. Training a detection algorithm "too hard" on a dataset may lead to *overfitting*, in which the algorithm has adapted to all minor variations in the data, even taking its embedded noise into account. It will then not perform well for other datasets where the noise is not exactly equivalent. Conducting training and testing on separate datasets ensures that such a failure of the algorithm to model the actual, underlying structure of the data will be spotted.

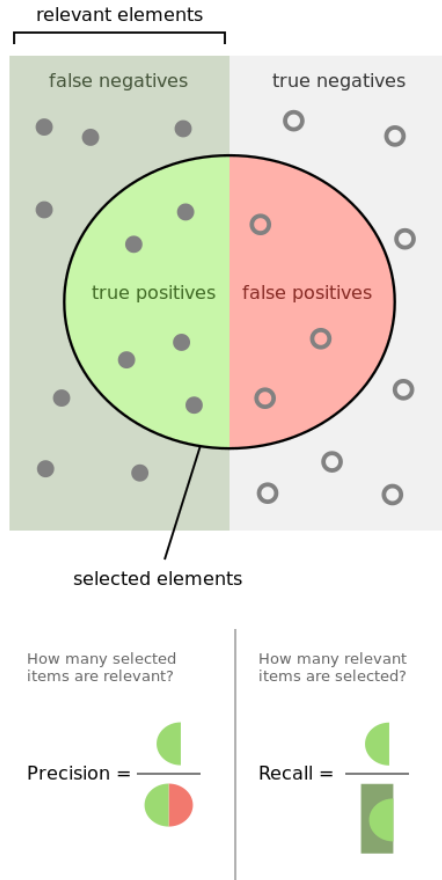


Figure 2: Precision and recall. Illustration by Walber, licenced under CC BY-SA 4.0 [14].

2.2.3 Calculating distance on Earth

Calculation of the distance between two points on Earth is not as straightforward as applying the Pythagorean theorem to calculate a straight-line Euclidean distance. This is due to the fact that the earth is a sphere, on which no straight lines exist. To obtain accurate measurements of distance, one must thus calculate the distance along the surface of the sphere, i.e. the *great-circle distance*. There exists several methods to solve for this distance which yield mathematically exact solutions, given two points (θ_1, γ_1) and (θ_2, γ_2), expressed in the spherical coordinates of latitude and longitude. One of them is the so-called *Law of cosines for spherical trigonometry*:

$$d = R\, arccos(sin(\theta_1)sin(\theta_2) + cos(\theta_1)cos(\theta_2)cos(\gamma_2 - \gamma_1)) \tag{5}$$

where R is the radius of the Earth and d is the calculated distance. However, as explained in [15], equation 5 suffers from a drastically increasing requirement for floating-point precision (the number of decimal digits maintained by the computer performing the computation) as the distance between the two points becomes small. As γ_1 approaches γ_2, the term $cos(\gamma_2 - \gamma_1)$ approaches 1 from below, and requires an ever extending series of decimal 9s to represent accurately. Even for modern computers using 64-bit floating point numbers, the law of cosines will start behaving erroneously when distances get down to a few meters [16].

Instead of relying on the law of cosines, one can use the Haversine formula, which is well-behaved even at small distances. A fitting formulation of the formula is

$$d = 2\, R\, arcsin\sqrt{sin^2\left(\frac{\theta_2 - \theta_1}{2}\right) + cos(\theta_1)cos(\theta_2)sin^2\left(\frac{\gamma_2 - \gamma_1}{2}\right)} \tag{6}$$

Note that both equation 5 and 6 model the Earth as a perfect sphere, and do not take elevation differences or terrain into account. The mean radius of Earth is defined by the International Union of Geodesy and Geophysics (IUGG) as $6,371,008.7714\, m$ [17], and is our best choice for R.

2.3 Equipment

The hardware and software described in this section have been provided by Aventi Intelligent Communication.

Figure 3: Kapsch EVK-3300 V2X Evaluation kit. Image from Kapsch TrafficCom AG [18].

In order to enable real-life data generation, to be detailed in section 4.3, two EVK-3300 V2X Evaluation kits from Kapsch [19] have been used. One such device is shown in figure 3, without the accompanying GPS antenna and power cords attached. These have a 5.9 GHz radio transceiver built-in, and include the full ETSI ITS-G5 protocol stack, thus enabling quick prototyping of ITS systems that are under development. The device can track its own position, as well as its current speed and heading. In addition, it contains an interface that can be connected to a vehicle's On-board diagnostics (OBD) port, thus enabling it to read certain parameters from the vehicle itself. However, the proprietary Kapsch software currently installed on the devices do not support this functionality, and this has thus not been used.

One of the devices was configured as an Onboard unit (OBU), to function as an ITS station mounted in a vehicle, broadcasting data for other stations to receive. Data is broadcasted at least once a second. The other was configured as a Roadside unit (RSU), which is meant to be integrated in traffic infrastructure, exchanging relevant information with the OBUs installed in vehicles.

The Kapsch devices use 12V power supplies, enabling them to be pow-

ered from the cigarette lighter socket in a vehicle. A 12V battery was used in order to power the RSU.

A piece of proprietary Kapsch software called the **ITS_server** can be run on a computer which is attached to the RSU via an ethernet cable, in order to display relevant information that the RSU receives from nearby OBUs.

3 Problem description

In [1], the original problem description given by Aventi (see appendix A) is divided into two challenges:

1. Setting up a Big Data analysis system to handle data from C-ITS and using this to extract relevant data from a generated dataset.
2. Using e.g. machine learning to classify driving patterns leading up to traffic accidents, and then warning drivers in real-time when such a driving pattern is recognized.

The first challenge was the focus of [1], and is considered as solved at the prototype/proof-of-concept level. The second challenge is the focus of this thesis. It has been slightly modified in cooperation with Aventi, to better fit their needs and the present technical possibilities. The formulation of the problem description used for the present work is:

Using RSUs integrated in road infrastructure to listen for CAM and DENM packages broadcasted by OBUs embedded in vehicles, identify dangerous traffic situations and use this to determine dangerous locations in real-time.

The objective was thus changed from a single-vehicle focus, where vehicles would be warned of a potential impending accident, to a fleet focus, where the sum of data from a fleet of vehicles would be used to determine the danger in the areas covered by them, in a way that could benefit the entire fleet.

To facilitate the development of such a solution, this general problem description was further split into three main goals corresponding with different levels of functionality:

(a) The ability to detect certain, specific types of dangerous situations in real-time from listening to vehicle data in the form of CAMs and DENMs.

(b) Using detected dangerous situations to assess the danger at particular locations in real-time.

(c) The ability to visualize these dynamics in the current level of danger in a way that would be valuable to a traffic control center operator, a driver or even the vehicles themselves.

4 Solution

4.1 Method

The solution to the problem description of section 3 will be entirely based on one simple assumption, which is also implicit in the problem description itself. It can be formulated as follows:

Where there are many accidents and/or near-accidents, generalized as "dangerous situations" (within a certain period of time), there is a high probability for an accident in the future.

This assumption makes perfect intuitive sense, as one knows that traffic accidents are not uniformly distributed across all stretches of road, but rather tend to pile up at critical points. An accident, near-accident or otherwise dangerous situation can serve as an indicator that a certain location is such a critical point, and that the danger associated with this location is higher than at other places. The assumption is also supported by research, which shows that, for instance, there are approximately 2.5 times as many accidents involving bodily harm in X-type intersections than in T-type intersections, and more than double the amount of accidents around the entrance to a tunnel as within it [20]. Furthermore, one could imagine that the level of danger at certain locations varies dynamically, for instance with the time of day and weather conditions.

In the most general sense, the solution to be made will fit into an ITS ecosystem as shown in figure 4. RSUs integrated in road infrastructure would listen to the broadcasted CAM and DENM packages from nearby vehicles, and transmit relevant extracted information as input to a data processing and analysis system running at some centralized or decentralized location, i.e. "the cloud". Designing and implementing a prototype version of this system is the goal of this work, and the result will be referred to as "the system" or "the solution", interchangeably. The system will be processing incoming data in real-time, and output some sort of assessment of the danger at particular locations at irregular time intervals.

Designing this system entails "filling in the blanks" in the data pro-

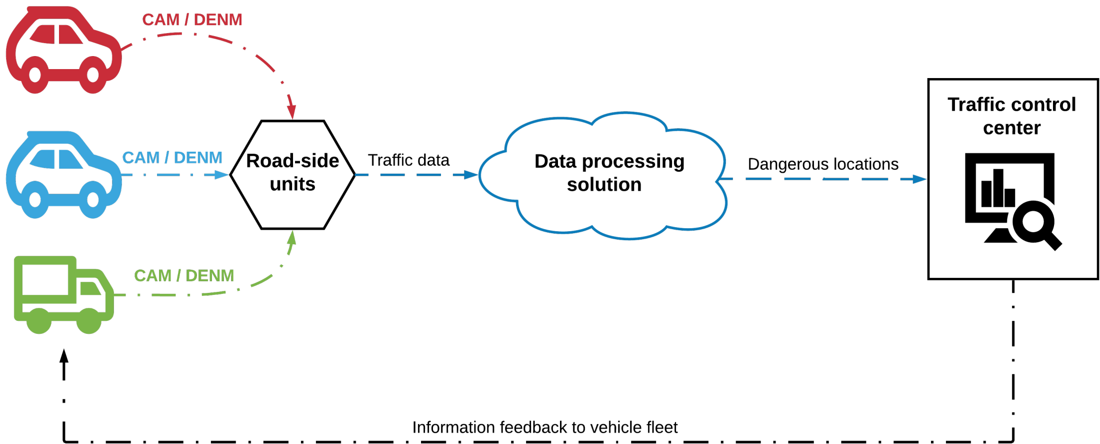


Figure 4: The general structure of the ecosystem in which a solution fitting the problem description of section 3, marked in blue, operates. The feedback from the traffic control center to the vehicle fleet illustrates one possible application of the results from the data processing solution.

cessing solution marked in blue in figure 4. Looking at the figure, there are three main questions to answer:

1. What is the input to the data processing solution?
2. What happens within it?
3. What is its output?

It was shown in [1] that the following variables can be extracted from each CAM package broadcasted by a vehicle, in the current state of the C-ITS equipment used (see section 2.3):

- Station ID, the vehicle's unique ID
- Timestamp, giving the time at which the other variables were captured
- Position, given as latitude and longitude coordinates

- Speed
- Heading

These variables contain the most important, highly dynamic information concerning a vehicle, and are what we have available to work with for further data processing. This information will therefore be passed forward from the RSUs as inputs to the data processing solution, answering part of question one above.

As explained in section 2.1.2, DENM packages contain information about events detected by the vehicles themselves, meaning that the detection part of the problem description (point **(a)** in section 3) has already been taken care of in this case. This has the potential of providing the system with a lot of valuable information about traffic situations "for free", and is thus definitely something one wants to pass forward as inputs to the data processing solution. Examples of the types of DENM events defined in [8] are:

- Accident
- Wrong way driving
- Dangerous end of queue
- Collision risk
- Slow vehicle
- Emergency vehicle approaching
- Dangerous situation
- Adverse weather conditions: Visibility
- Hazardous location: Obstacle on the road

ETSI [8] does not concern itself with how these events are detected, and neither do we. DENMs don't even necessarily have to originate from a vehicle, they can in theory be generated by the RSUs themselves. The important point is that there are laid out containers in the DENM protocol, which vehicle and equipment manufacturers can use to broadcast

these kinds of events once their own event detection systems get advanced enough. The system must therefore be set up to also be able to handle DENM data as an input.

Question two, concerning the inner workings of the data processing solution, is naturally the most demanding among the three. It can be further broken down into three distinct parts:

Detection of different events from the incoming CAM data stream. Events to be detected must be defined, their characteristics specified, and an algorithm for detection must be designed and tuned to accurately filter out an actual event from the surrounding noise. Different detection algorithms should run in parallel in order to be able to detect different types of events.

Consolidation of the detected events. The system should be designed with the ability to handle all kinds of events gracefully, whether predetected in a vehicle or RSU and transmitted as a DENM package, or detected by applying some collection of algorithms to the incoming stream of CAM data. Event types should be weighted differently according to a relative measure of severity (statically defined or dynamically calculated somehow), so that they can be compared and combined.

Calculation of danger over a map based on detected events, and triggering of an output when a location is deemed sufficiently dangerous. This involves expressing each event in space by characterizing how it affects its surroundings, in time by characterizing how its impact changes as time goes by, and combining several events in order to calculate the total danger indicated by them. This process will be termed *clustering*.

As mentioned, the system should generate an output when it determines that a location has a certain level of danger associated with it. This will be the final output of the system, and can for instance be used to generate an alarm in a traffic control center, or serve as input to other systems that can take some automated action. The fact that detection, consolidation and calculation happens continuously in real-time means that a

location which has previously been regarded as dangerous can go back to being safe, which, upon detection, should generate an output as well.

Figure 5 illustrates a fitting modularization of the described system. Partitioning the system into separate modules that each solve smaller parts of the bigger picture, and having simple data flow between them ("signals"), means that implementation can focus on and optimize each module independently. Note, however, that the figure only serves as an overall guide for further design and implementation, meaning that the keywords written on the modules and signals will only serve as hints, not as hard restrictions. Nevertheless, figure 5 serves as a nice specification of the overall layout of the solution.

The design and implementation to be laid out in the coming sections will entail fully specifying each module in terms of input and output, as well as designing and implementing all their inner functions.

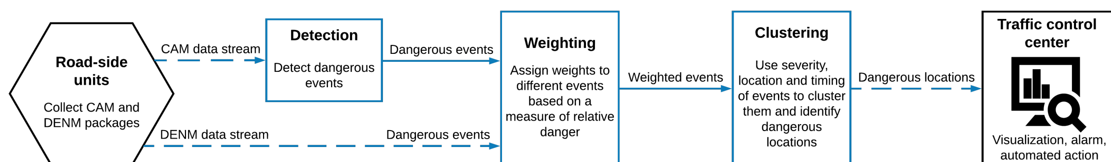


Figure 5: The modules that make up the full system, as well as the information flow between them, and the system's initial input and final output. The modules and signals that are internal parts of the solution are marked in blue, while the system's interfaces with the outside environment are shown as dashed, blue lines.

4.2 Scope

Due to constraints in time, hardware limitations and a lack of large, high-quality data volumes, it was necessary to constrain the focus of the work, i.e. to define a proper scope. In particular when it comes to the problem of event detection, as described in the previous section, clear constraints were necessary. As C-ITS is a technology still in is infancy, there is not available any large datasets of different traffic situations, that would be usable in

designing and tuning event detection and classification algorithms. Such data therefore had to be generated manually as part of this work (see section 4.3). To be able to generate enough data for a proof-of-concept event detection system, a single, specific event type was chosen: *Abrupt braking*. This event type was chosen because of the following features:

Ease of data generation: The problem of safely, effectively and economically generating a dataset of a sample of dangerous situations is not straightforward by any means. How do you reliably simulate for instance a "near-accident" without risking a real one? How can one do this over and over to build a dataset, without vast resources? Abrupt braking, on the other hand, can be performed by a single driver, in a safe and controlled manner without involving other vehicles.

Reliable danger indicator: Without having to delve into the myriad if different scenarios that can be considered as *dangerous situations*, and having to model complex interactions between different vehicles, infrastructure, pedestrians and so on, one can simply say that "If someone performs an abrupt braking, something unexpected, potentially dangerous, has happened". In essence, we measure the symptom of a dangerous situation instead of the situation itself, reducing many different situations to one. We call abrupt braking a *danger indicator*.

Observability: As listed in section 4.1, the current state of the hardware and software only allows the measurement of a small set of variables. One of these is *speed*, which is incidentally the only variable we really need for detection of abrupt braking. This is thus one of the few types of events that *can* be reliably detected at this point.

This allowed a much-needed focus for both the manual generation of data (section 4.3) and the design and implementation of the detection module (section 4.5.1).

As support for properly generating DENM data has not been fully implemented in the equipment used, such data will only be handled at a superficial level by the system. A skeleton for handling it will be built, which can later be extended when the need comes.

4.3 Generating data

Even though we have defined a braking profile for ideal, normal braking (figure 1), and have declared *abrupt braking* as any sufficiently large deviation from this profile (in the direction of a more powerful maneuver), there was a need to manually generate real-life data, as explained in the previous section. The ideal braking profile had to be validated against genuine data, and be tested to see how it would hold up against noise and natural variations.

Ideally, one would use a vehicle carrying an OBU, and place battery-powered RSUs along a road path, such that their receptive fields[3] overlap and cover the path entirely. This will be the ideal case once C-ITS has become widespread, and such a hypothetical situation is shown in figure 7. However, time and manpower constraints demanded a simpler operation.

The solution was to use a single vehicle, an ordinary car, carrying an OBU powered by the cigarette lighter socket. In the vehicle was also an RSU powered by a 12V battery, which was thus carried along the vehicle's route, eliminating the need for multiple, stationary RSUs. This accurately approximates a situation where RSUs are spread across the route, and their CAM data aggregated to one dataset. Notes were taken along the way in the form of a continuously running voice recording. These simplification measures enabled a smaller scale, one-man operation for generating real-life CAM data. In order to capture and save the generated data, the RSU was connected by an Ethernet cable to a laptop computer. The Kapsch **ITS_server** software was modified to continuously write the relevant variables in a CAM package to a file on the computer. Continuously writing the data, instead of batch downloading it at the end of the test (as was done in [1]), was also critical for enabling real-time functionality of the system that was to be built.

The vehicle was driven around the Trondheim area. Abrupt braking was performed in a controlled manner, when all other vehicles were at a safe distance, as to not confuse or potentially endanger anyone. Care was taken to perform abrupt braking at all speed levels, i.e. from approximately 80 km/h to 60 km/h, from 50 km/h to 30 km/h, and from 30 km/h to a standstill. In addition to abrupt braking, ordinary decelerations with

[3]The region of space from which the RSU can receive messages broadcasted from OBUs.

a varying degree of force was of course also performed, as well as some that can be said to have been in the borderline area between strong/determined and abrupt. Having both samples that are clearly positive, as well as ones that are negative[4], is essential for properly tuning a detection algorithm.

After driving, the voice recorded notes were used to label the data at the places which were deemed as abrupt brakings, providing a ground truth for the detection algorithm. This labeling process was performed using the tools developed in section 4.4.3.

The result was two separate datasets, originating from two distinct legs of driving which were performed consecutively on the same day. The one containing slightly more data was designated as the *training set*, while the other was used as a *test set*. They are placed in the **datasets** folder in the attached material, and called **training_set** and **test_set** respectively. The raw data is in the form of **.log** files, while **.csv** files contain the data after preprocessing (see section 4.4.1), and the labeled data come as **.mat** files.

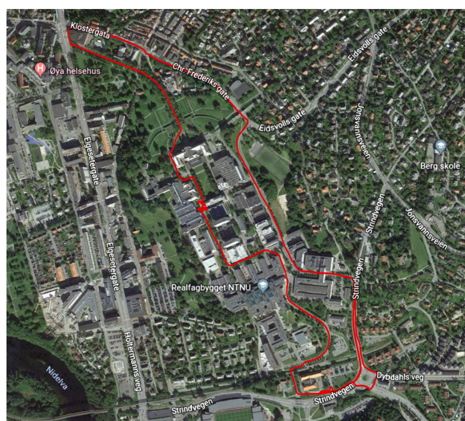


Figure 6: The path traveled by a vehicle with an OBU broadcasting data to an RSU which the vehicle carries along. The figure is from [1].

[4]See section 2.2.2 for a definition of these terms.

Figure 7: A hypothetical situation where the path from figure 6 is entirely covered by RSUs, shown as antennas. Their receptive fields are shown as blue circles. The figure is from [1].

4.4 Utility functions

A number of software functions and scripts were written to serve as aids in processing and visualizing the datasets that were generated. These have been termed *utility functions*, and are placed in the **utilities** folder in the attached material. Some of them are listed below, in sections 4.4.1-4.4.3, while some will be presented as part of later sections.

4.4.1 Preprocessing

Such a simple script is called **log_to_csv.py**, and is a Python [21] script that takes the raw dataset (**.log**) file as input, and converts the data to a nicely formatted CSV file. It also performs some light preprocessing on the data, converting all units from proprietary versions to proper ones, i.e. decimal degrees for latitude and longitude, km/h for speed, and degrees for heading. Timestamps, which are originally 16-bit numbers wrapped to 65536 (2^{16}), meaning that they start over at 0 after 65.535 seconds [7], are "unwrapped" by making them monotonically increasing[5]. These measures

[5] "Not decreasing"

make later processing simpler and less error-prone.

The script produces a file with the following columns:

- station_id
- timestamp
- unwrapped_timestamp
- lat
- long
- speed_kmph
- heading

Generated CSV files are placed in the same location as the original log file, i.e. in the **datasets** folder in the attached material.

4.4.2 DENM generator

A script called **denm_generator.py** was constructed in order to generate different types of DENM events as input to the system, as this could not easily be done using the Kapsch equipment. The script is given a probability p between 0 and 1 as input, designating the probability per second that an event will be generated. The event's type is chosen at random among the available ones (listed in figure 21), the severity is set between 1 and 5, and the event is placed at a random location on a given map. A station ID of the vehicle reporting the event is also set randomly.

This effectively replaces the RSU as the origin of the DENM data stream in figure 5 for testing purposes, allowing us to test the system with a variety of DENM data. It introduces some degree of randomness, which will make sure that the implementation is tested for a multitude of different cases. Generated events are outputted to the file **its_server_denm.log** in the **log** folder, with the following format:

```
Received DENM: { 'ts': 1525447097, 'reporter_id': 93289, 'type':
    'driving_wrong_way', 'lat': 634220000, 'long': 104140000,
    'severity': 1.40 }
```

4.4.3 Replaying and labeling a dataset

Replay for labeling

In order to visualize a generated CAM dataset, as well as assisting in properly synchronizing the voice recorded notes with the data (which is needed to label the data with abrupt braking events in the right places), a script called **replay_cam.m** (in the **matlab** folder) was written in Matlab [22]. This script takes a dataset (in the form of a CSV file, see the previous section) as input, and plays it back in real-time through three visualization windows:

Map view
: The map view (figure 8) visualizes the vehicle's movement as a path in 2D space. It can either be plotted on a road map acquired from the OpenStreetMap (OSM) project [23][6], or just as a path in empty space (as seen in figure 9). Note that plotting on the OSM map is very computing power intensive, so it may break the real-time playback, making it lag behind after a short while. The full path of the vehicle is plotted as an orange or green line, and the vehicle's current position, as the dataset is being played back, is marked as a blue dot. The view provides interactive control, such as zooming and panning around.

Speed view
: The speed view (figure 10) is a graph of the speed of the vehicle over time. Again, the full history (past and future) is plotted in green, and a blue dot marks the vehicle's current speed.

Heading view
: The heading view (figure 11) indicates the current heading of the vehicle on a familiar circular compass. North (0°) is at the top, south (180°) at the bottom. This view provides extra information that is of assistance when visualizing the vehicle's driving pattern.

Note that there is a button in the lower left corner of the speed view in figure 10. This button can be clicked to place an *Abrupt braking event*

[6] A fitting map for the training dataset is included as a **.osm** file in the **datasets** folder in the attached material.

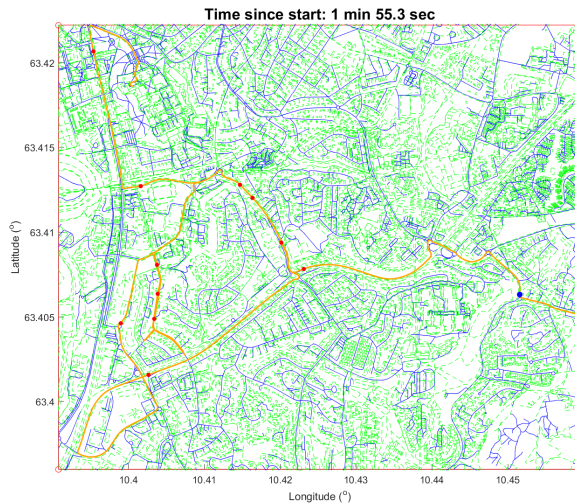


Figure 8: The map view with an OSM road background. The orange line shows the full path of the vehicle, while the blue dot marks the its current position. Abrupt braking events are shown as red dots.

label at the current position in time. Such an event will then appear as red dots in the speed and map views, as shown. Labeled events are automatically saved in a Matlab data file (**.mat**) together with the dataset, for later use.

This script made it possible to easily listen to the voice-recorded notes taken while generating a dataset while playing it back in real-time and labeling the data in the right places.

Replay for testing

A Python script called **replay_cam.py** was constructed in order to easily provide real-time-like CAM data as input to the detection module. It takes a generated dataset as input, in the form of lines written to a **.log** file by the **ITS_server**, and uses the timestamps of the dataset's CAM packages to output them to a new log file (**its_server_cam.log**) with proper timing,

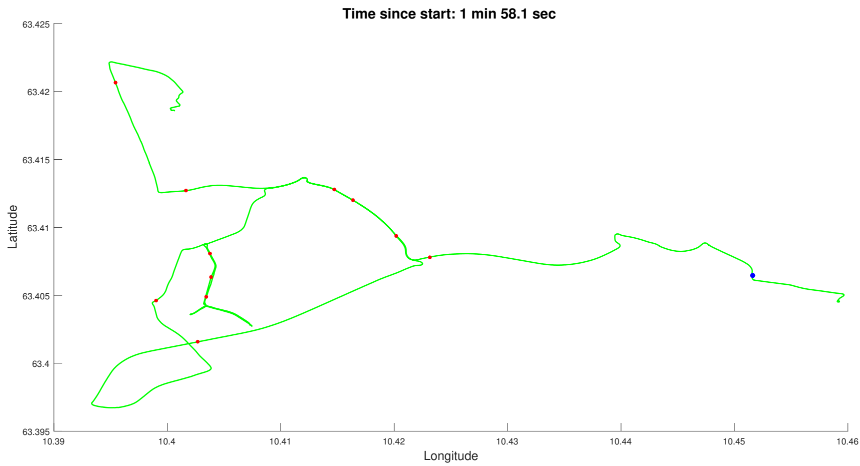


Figure 9: The map view without the OSM background. The green line shows the full path of the vehicle, while the blue dot marks the its current position. Abrupt braking events are shown as red dots.

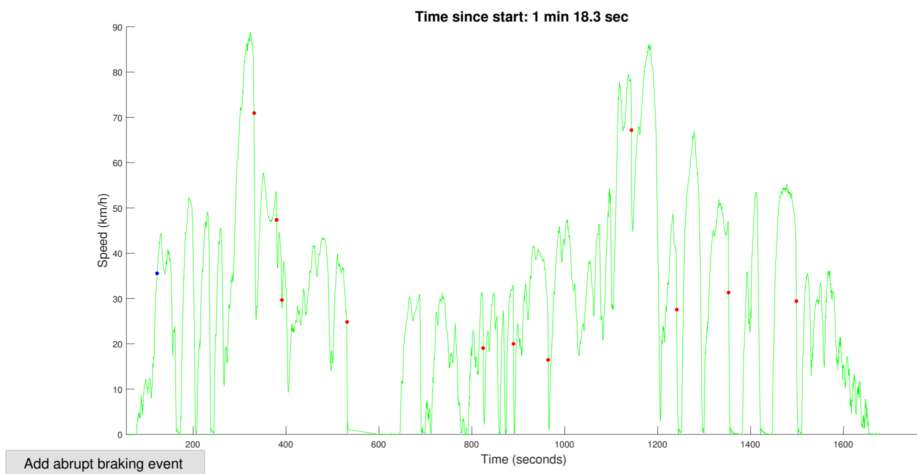


Figure 10: The speed view, with a blue dot marking the current speed. Abrupt braking events are shown as red dots.

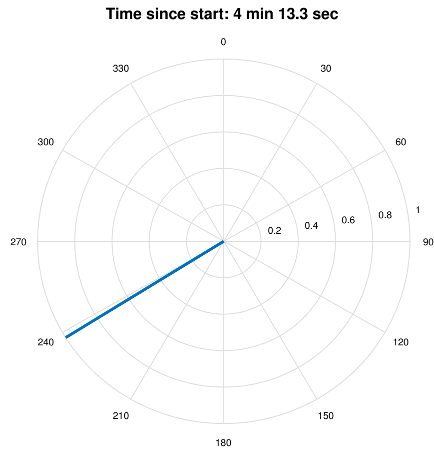


Figure 11: The heading view shows the current heading of the vehicle on a compass.

giving the appearance that the data is collected from a vehicle in real-time. This replaces the RSU as the origin of the CAM data stream in figure 5 for testing purposes.

4.5 Module design and implementation

Python [21] was chosen as the programming language for implementation of the solution, due to its high level of abstraction enabling quick development and testing, its wide support across operating systems and its rich ecosystem of libraries and helpful online communities. The system implementation can be found in the **solution** folder in the attached material, and all the files and folders referenced in this section resides there, unless otherwise specified. The implementation requires a number of external libraries to be installed for correct operation. These can be found in the **requirements.txt** file, and can be automatically installed using *pip* [24]:

```
pip install -r requirements.txt
```

The program is written such that each of the modules discussed in section 4.1, and shown in figure 5 are implemented as a separate thread[7]. This makes it easier to create an implementation that adequately approximates the design, where each module is independent, with black-boxed[8] internal workings, and simple signals between them. These signals are implemented in terms of Python *queues*, which are First in, first out (FIFO) pipes that allow threads to pass data between each other in a safe way.

Each module, implemented as a thread, resides in its own **.py** file. The **main.py** file is responsible for starting all the threads, and is the entry point of the program. Thus, the program can be started by running

```
python3 main.py
```

The **main.py** file also takes care of some real-time visualization functionality, which will be further explained in section 4.5.5.

Each module logs its operation to a separate file in the **log** folder. A simple script called **monitor_logs.sh** in the **monitoring** folder has been

[7]Having multiple threads allow programs to be split into concurrent processes, that can run in parallel (either by being executed on a multiprocessor or multi-core system, or by being run on a single processor that switches between the threads often and fast enough to emulate parallel execution).

[8]In dealing with a black box module, one cannot see, or does not care, about its internal workings. A black box is expressed only in terms of its inputs and outputs.

written to open each log file in a separate terminal window, displaying new logs as they appear. This provides useful insight into each module's operation, as well as their interaction.

In addition to the **main.py** file and the files corresponding to each of the modules from figure 5, there are also a number of classes and helper modules in separate files, which will be further explained in the coming sections. The constructed software will sometimes be described using flowcharts that illustrate operation, and sometimes by presenting and describing the code itself. In such cases, the code may be slightly modified and irrelevant parts omitted, in order to maximize readability. The full code is always included in the attached material.

4.5.1 Detection

In a future, further developed system, the detection module will run several algorithms in parallel, each trying to detect a separate type of event. However, as explained in section 4.2, the purpose of the detection module at this stage will be to detect *abrupt braking* events only.

As explained in section 2.2.1, the theoretically ideal way to brake a vehicle is by performing a smooth deceleration with a magnitude of approximately 3-4 m/s^2 at maximum. However, a robust detection algorithm cannot be made on the basis of theoretical considerations alone, but also (and perhaps mainly) by investigating real-world data, and using such data for tuning the algorithm.

Investigating the data

The training dataset contains 11 abrupt braking events, as well as approximately 50 normal braking events[9]. Using Matlab to first investigate the normal braking events, we get data such as that shown in figure 12. We can clearly see five such events in the speed graph, two of which are decelerations from approximately 50 km/h to a standstill, one from 40 km/h to 10 km/h, and one from 40 km/h to a standstill by way of two separate braking events, with a short period of cruising between them. Looking at

[9]Exact labeling of all the normal braking events is not important, as we are not designing a detector for normal braking events. Such events will in the end just be considered as "not abrupt braking", alongside all the other, non-eventful data.

the acceleration data, which is the time derivative of the raw speed data, we see that the deceleration at three of these events peak at between 2 and 3 m/s^2, and at up to 4 m/s^2 for the two others. Analyzing the rest of the dataset, it is clear that these values are very typical for normal braking events, even at other speed levels. For instance, both the braking from 80 km/h to 70 km/h (at approximately 1120 seconds), and from 80 km/h to an almost standstill (at 1200 seconds) shown in figure 13, have deceleration peaks of around 2 m/s^2. Furthermore, it is evident that some of the deceleration spikes, particularly the most severe ones, are artifacts of the act of differentiating a somewhat noisy speed signal, as these do not correspond with large, persisting drops in the speed graph.

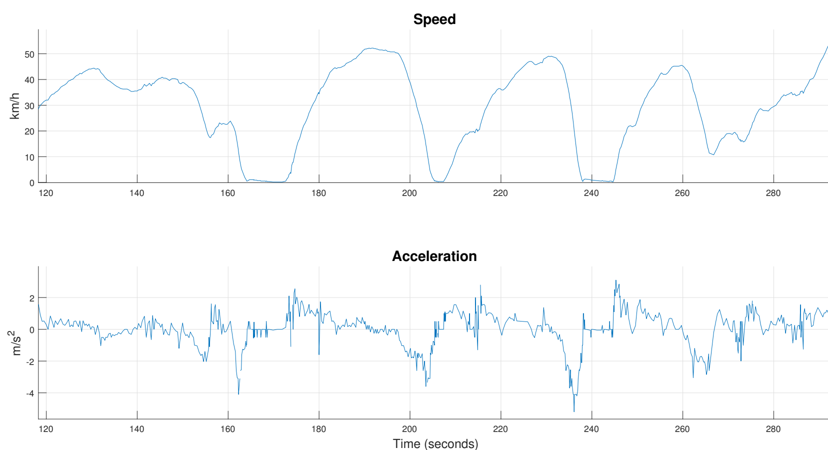


Figure 12: Five normal braking events at speed levels of approximately 40 to 50 km/h. Two in close succession at around 160 seconds, and then at 200, 235 and 265 seconds.

Filtering

One can use different filtering methods in order to remove or reduce this noise, and obtain a cleaner, smoother acceleration signal. It is important to remember that the large speed drops themselves must not be smoothed out, as these represent the braking events that we want to detect and

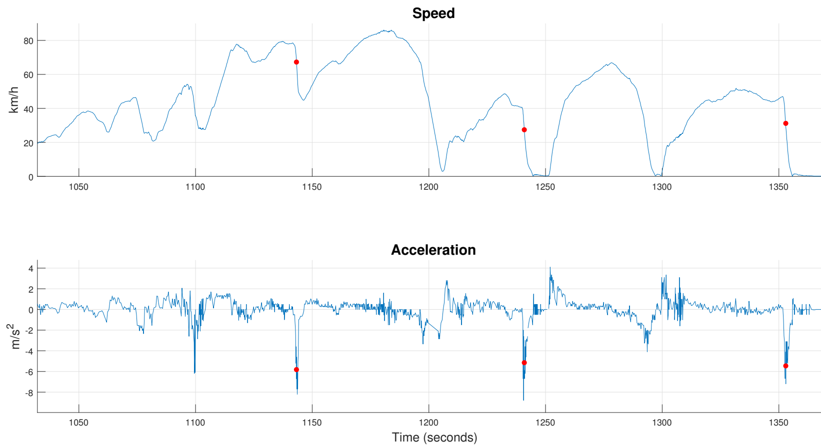


Figure 13: Raw data examples of normal (unmarked) and abrupt (marked with red dots) braking. Note that as the abrupt braking events are marked manually using the tools developed in section 4.4.3, the markers are not necessarily right at the acceleration minima.

classify as normal or abrupt, and their slopes (accelerations) are exactly the characteristic we need to preserve in order to distinguish between the two. Furthermore, we can take advantage of the fact that braking events will differ from noise in that they persist over some time, while noise often materializes as short-duration spikes. Thus, instead of filtering the speed data before differentiating, we can rather differentiate the raw speed data, and then apply a moving average smoothing filter in order to remove the noise. Using a simple, symmetrical filter ensures that the filtered data will be aligned with the raw data, instead of being shifted in time, though the filtering action will lag behind the incoming data stream by a few data samples (half the filter window size). Even for a real-time solution, this is okay as long as the filter window size is reasonably small. The filter window size is chosen as part of optimizing the detection algorithm, as detailed below. An excerpt of the resulting acceleration data is shown in figure 14, which can be compared with the raw acceleration data in figure 13.

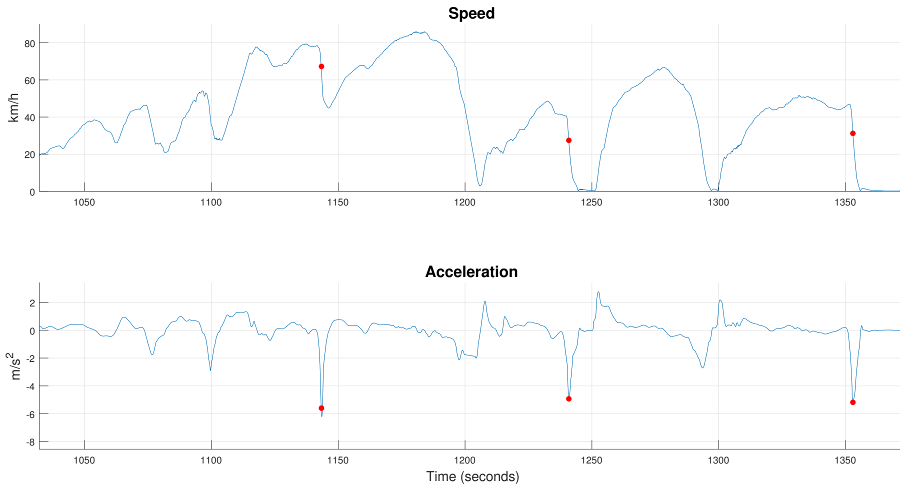


Figure 14: An excerpt of the filtered acceleration data, using a filter window size of 15 data points. The large spikes present in figure 13 are gone.

Detection algorithm

One can see from figure 14 that the normal braking events shown have a peak (filtered) deceleration of no more than approximately 3 m/s^2, while abrupt braking events have peaks at more than 4 m/s^2. This suggests that a simple deceleration threshold should mostly suffice to detect abrupt braking events, an approach that has the added benefit of being highly intuitive and in agreement with the theoretical reflections of section 2.2.1.

In order to detect each event only once, and to use the most severe moment of the deceleration as its "center", we first extract all the local minima of the smoothed acceleration data (the local maxima of the deceleration). The minima are then filtered by discarding the ones that do not pass a threshold value. Finally, as there may be more than one local minima associated with the same braking event, those that are close to each other in time (less than 2 seconds between them) are regarded as one, by discarding all but the most severe one. The full algorithm is summarized in table 2.

Table 2: The designed algorithm for detecting abrupt braking events

1. Differentiate the raw speed data, to obtain raw acceleration data.

2. Apply a moving average smoothing filter to the raw acceleration data.

3. Identify the local minima of the smoothed acceleration.

4. Use a static threshold to discard the minima which do not have sufficiently large magnitudes.

5. Reduce minima that are close together in time to one, by keeping only the single most severe one.

6. Each remaining minima is a detected event. Use the absolute value of the acceleration minima as the event's *severity*, and the location and time at which the minima is situated as its *center* in space and time.

Tuning the algorithm

This algorithm has two tunable parameters: The window size of the moving average smoothing filter, and the deceleration threshold[10]. A Matlab script called **detection_algorithm_tuning.m**, placed in the **utilities/-matlab** folder, was made in order to use the training dataset to tune these parameters through trial and error, the objective being to maximize the F-score of the detection algorithm, which is calculated according to equations 2-4. The script iterates over a given range of filter window sizes and deceleration thresholds, calculating the smoothed acceleration and applying the detection algorithm for every combination of the given parameters. Figure 15 shows the output of the script while it is running, illustrating the iterative trial and error approach to finding the optimal parameter values.

The script then uses the resulting F-scores to declare the best combi-

[10]Again, deceleration is used as the negative of acceleration, meaning that a deceleration of e.g. 2 m/s^2 is equivalent to an acceleration of -2 m/s^2.

```
Running detection algorithm for 26 filter window sizes and 41 thresholds (1066 combinations).
1.  Processed filter_window_size = 1, best F-score = 0.54054 (at threshold = -5.6)
2.  Processed filter_window_size = 3, best F-score = 0.4878 (at threshold = -5.6)
3.  Processed filter_window_size = 5, best F-score = 0.7619 (at threshold = -5.4)
4.  Processed filter_window_size = 7, best F-score = 0.8 (at threshold = -4.8)
5.  Processed filter_window_size = 9, best F-score = 0.84211 (at threshold = -4.8)
6.  Processed filter_window_size = 11, best F-score = 0.81818 (at threshold = -3.8)
7.  Processed filter_window_size = 13, best F-score = 0.85714 (at threshold = -3.6)
8.  Processed filter_window_size = 15, best F-score = 0.85714 (at threshold = -3.5)
9.  Processed filter_window_size = 17, best F-score = 0.85714 (at threshold = -3.3)
10.  Processed filter_window_size = 19, best F-score = 0.81818 (at threshold = -3)
11.  Processed filter_window_size = 21, best F-score = 0.81818 (at threshold = -2.7)
12.  Processed filter_window_size = 23, best F-score = 0.8 (at threshold = -3)
13.  Processed filter_window_size = 25, best F-score = 0.8 (at threshold = -2.9)
14.  Processed filter_window_size = 27, best F-score = 0.8 (at threshold = -2.8)
15.  Processed filter_window_size = 29, best F-score = 0.8 (at threshold = -2.7)
16.  Processed filter_window_size = 31, best F-score = 0.8 (at threshold = -2.6)
17.  Processed filter_window_size = 33, best F-score = 0.7619 (at threshold = -2.5)
18.  Processed filter_window_size = 35, best F-score = 0.7619 (at threshold = -2.4)
19.  Processed filter_window_size = 37, best F-score = 0.7619 (at threshold = -2.3)
20.  Processed filter_window_size = 39, best F-score = 0.7619 (at threshold = -2.2)
```

Figure 15: Determining the optimal filter window size and deceleration through scripted trial and error.

nation(s) of the parameters. The five best-performing parameter combinations and their scores are listed in table 3. As can be seen, a window size of 15 data samples and a detection threshold at -3.5 m/s^2 of acceleration has the best score. Note that this deceleration threshold corresponds nicely with the theoretical values that were determined in section 2.2.1.

The full performance of the algorithm, tuned and tested on the training dataset is as follows:

True positives: 9

False positives: 1

False negatives: 2

Precision: 0.900

Recall: 0.818

F-score: 0.857

Table 3: Detection algorithm performance for different parameter combinations

Filter window size (number of data points)	**Deceleration threshold (m/s^2)**	**F-score**
15	3.5	0.857
9	4.8	0.842
21	2.7	0.818
13	3	0.800
19	2.8	0.783

Figure 16 neatly presents the details of this performance. The actual events (ground truth) are marked as dots in the speed graph, and color coded based on whether each event was detected by the algorithm (true positive) or not (false negative). In the acceleration graph, the detection threshold at -3.5 m/s^2 is displayed as a horizontal, red line, and the detected events are marked with dots, color coded based on whether they correspond to a real event (true positive) or were wrongly detected (false positive). A detected event is deemed to be a true positive if there exists an actual event within two seconds of it, and is otherwise regarded as a false positive.

Implementation

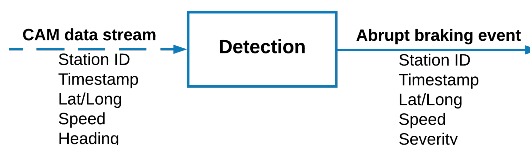


Figure 17: The input and output of the detection module.

Once designed, the detection algorithm was implemented as a Python module, as part of the larger system of modules shown in figure 5. The

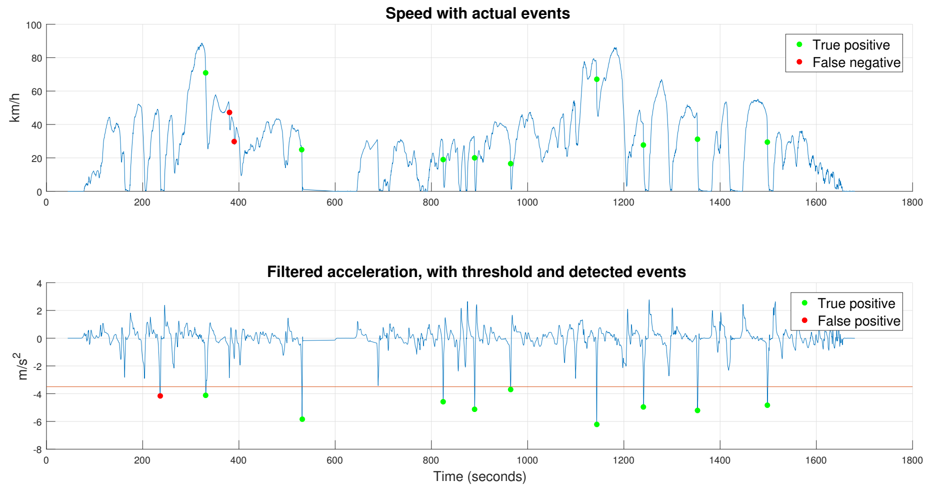


Figure 16: Running the detection algorithm on the training dataset, with color coded events marked as true positives, false negatives or false positives. The upper speed graph shows the actual events, while the lower acceleration graph shows the detected events.

input and output of the detection module is further specified in figure 17.

The entry point of the implementation of the detection module is the **detection.py** file in the **solution** folder. It implements a sequence of operations, described in general terms in figure 18, that is triggered by the reception of a new CAM package from the **ITS_server** connected to the RSU.

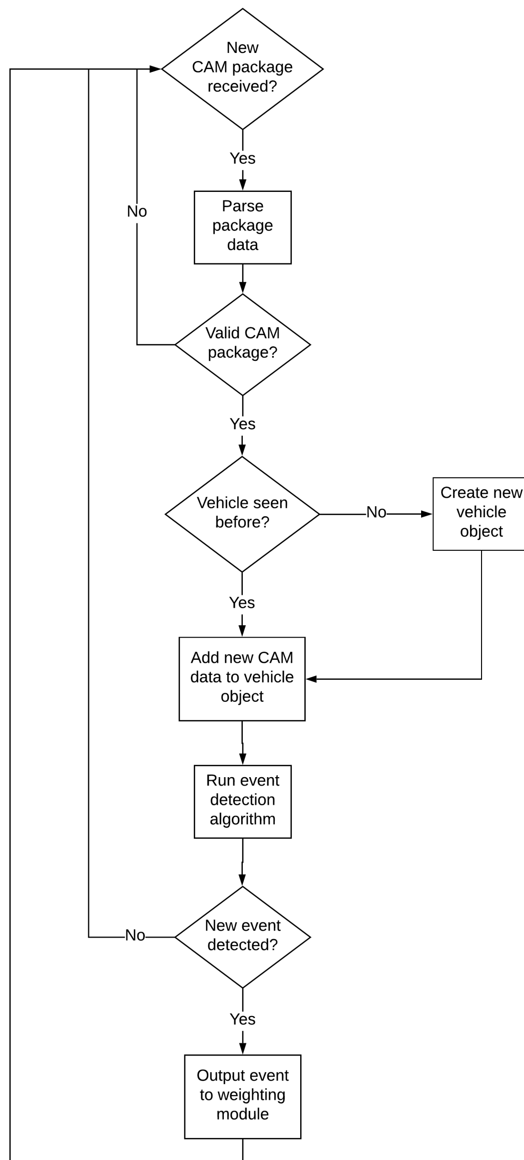


Figure 18: A general description of the sequence of operations that constitute the inner workings of the detection module.

For deeper detail, the code implementing these operations is given below, and then further explained.

```
42   with open(its_server_log_file, 'rb') as f:
43       # Set read head at the current end of the file, so only new,
             incoming lines are read
44       f.seek(0,2)
45
46       while 1:
47           line = f.readline().decode('utf-8')
48
49           try:
50               cam_data = parse_line(line)
51
52           except ValueError:
53               continue
54
55           except EOFError:
56               f.seek(-len(line), 1) # Offset back to start of line
                     from current file position
57               logger.debug("Line is not a full package, seek
                     position reset.")
58               time.sleep(0.01)
59               continue
60
61           # Line has been successfully parsed into a package
62
63           logger.debug(cam_data)
64
65           try:
66               v = get_or_create_vehicle(vehicles, cam_data,
                     detection_threshold, event_duration,
                     filtering_method, filter_window_size)
67               v.new_data(cam_data)
68               event = v.detect_event()
69
70               if event:
71                   logger.info(event)
72                   weighting_input_queue.put(event)
```

When executed, this code starts listening to incoming CAM packages in the form of new lines at the end of the **its_server_cam.log** file written by the **ITS_server**. The CAM packages are received in the following format:

```
Received CAM: {'station_id': 2115950905, 'ts': 24456, 'lat':
    634045166, 'long': 104591116, 'speed': 164, 'heading': 2983}
```

At line 50 above, a function `parse_line()` uses this known format to parse the line into a Python *dictionary*, a data structure that is well suited for storing the data for further processing. The function also checks that the package is properly formatted and complete, and raises errors that are handled at lines 52 to 59 if not. Once parsed, on line 66 the `station_id` in the CAM packet is used to create a new `vehicle` object, or return an existing one if the vehicle with this `station_id` has already been seen. The `vehicle` class which implements such objects is defined in the **vehicle.py** file, and contains a set of data structures and methods that are meant to represent a single observed vehicle in the digital domain. This enables concurrent tracking of data from multiple vehicles. The most important contents of the vehicle class are summarized in the UML diagram in figure 19.

Data related to a vehicle is stored persistently as a Pandas [25] *dataframe*, which represents data as a time series, allowing operations such as indexing by timestamp and time differentiation to be performed on it. Configuration variables related to event detection is also stored here, potentially opening for applying different configurations to different vehicles.

At line 67, the `vehicle` object is updated with the incoming data through `v.new_data(cam_data)`, which works as illustrated by figure 20. As described here, the endpoint problem that arises when the first data point from a new vehicle is received is solved by simply setting the acceleration to zero at this point. Furthermore, one has to wait until there exists a number of data points equal to the filter window size before the filter can be applied, and filtering will then lag by

$$n_{lag} = ceiling(filter_window_size/2) \tag{7}$$

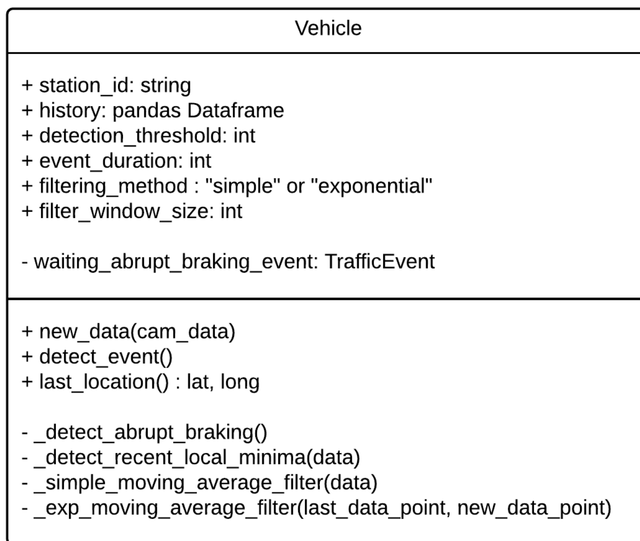


Figure 19: UML diagram summarizing the most important contents of the `vehicle` class. The top section shows the variables that hold the data related to the vehicle, while the bottom section shows the most important methods that can be called on a `vehicle` object.

data points behind the incoming raw data, as explained earlier.[11] The number of data points that have been filtered at any point then follows

$$n_{filtered} = n_{raw} - filter_window_size + 1 \tag{8}$$

i.e., for $filter_window_size = 15$, when 15 raw data points have been received, one has been filtered (the 8th data point). The filter is implemented as a simple average over the last $filter_window_size$ number of variables:

```
1 def _moving_average_filter(self, data):
2     data_inside_window = data[-self.filter_window_size:]
3     return sum(data_inside_window)/float(self.filter_window_size)
```

[11] The *ceiling()* function takes a real number as input and outputs the smallest integer larger than or equal to this number, i.e. it "rounds up" to the nearest integer.

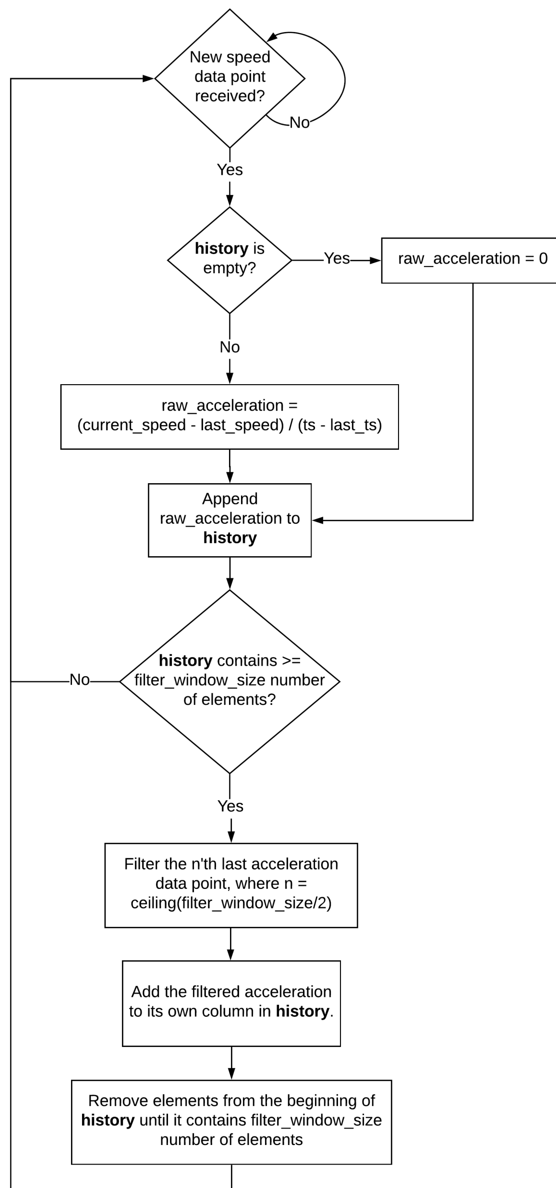


Figure 20: The inner workings of the `Vehicle.new_data()` function, which updates a vehicle object with newly received CAM data.

After being updated with the latest (filtered) data, the event detection algorithm is run at line 68, by calling `v.detect_event()`. This in turn calls the internal function `_detect_abrupt_braking()`, which is where the detection algorithm of table 2 is implemented:

```
1  def _detect_abrupt_braking(self):
2     if self.history["filtered_acceleration"].count() < 3:
3        return None
4
5     # There are no filtered values for the last _lag_ data points
6
7     lag = -ceil(self.filter_window_size/2)
8     accelerations =
          self.history["filtered_acceleration"].values[:lag+1]
9
10    minima_ix, acceleration =
          self._detect_recent_local_minima(accelerations)
```

Here, we make sure that enough filtered acceleration values have been produced, which follows equation 8, as it is not desirable to run the detection algorithm on raw acceleration values. At least three such values are needed in order to run local minima detection:

```
1  def _detect_recent_local_minima(self, data):
2     # Detects whether the next to last data point was a local minima
3     # Returns the index (always -2) and the value at the detected
          minima, or 'None, 0'
4
5     candidate_point = data[-2]
6     left_point = data[-3]
7     right_point = data[-1]
8
9     if (candidate_point < left_point) and (candidate_point <
          right_point):
10       # The candidate point is indeed a local minima
11       return -2, candidate_point
12    else:
13       return None, 0
```

Returning to `_detect_abrupt_braking()`, we check whether a found local minimum is below the threshold value:

```
11 if acceleration < self.detection_threshold:
12     # Abrupt braking detected
13     severity = abs(acceleration)
```

In that case, the event's *severity* is computed, simply as the absolute value of the acceleration minimum, as specified in point **6** of the designed detection algorithm from table 2. The severity is then compared to the severity of any previously found abrupt braking event which has not yet been transmitted as an output. This behavior is in line with point **5** of the algorithm specification, which is implemented by waiting two seconds before outputting an event, to see if there comes an acceleration minima that is even more severe. This is implemented as follows:

```
14 # If there are multiple abrupt decelerations closely spaced in
       time, compress them to one event by only picking the worst one
15 if self.waiting_abrupt_braking_event:
16
17     time_since_last = (ts_at_event -
           self.waiting_abrupt_braking_event.time).total_seconds()
18
19     if time_since_last < self.event_duration and severity >
           self.waiting_abrupt_braking_event.severity:
20         # There was recently an abrupt braking event, but this one
               is worse, so replace the last one with this
21
22         self.waiting_abrupt_braking_event =
               traffic_event.TrafficEvent(self.station_id,
               "abrupt_braking", ts_at_event, lat, lon, speed,
               severity=severity)
23 else:
24     # No recent abrupt braking events, so save this one
25
26     self.waiting_abrupt_braking_event =
           traffic_event.TrafficEvent(self.station_id,
           "abrupt_braking", ts_at_event, lat, lon, speed,
           severity=severity)
```

As seen here, events are saved as instances of the `TrafficEvent` class, which is implemented in the file **traffic_event.py**. This class is described in figure 21. Lastly, if there exists a waiting event, whether detected now or previously, we check if enough time has passed since its occurrence in order for it to be considered a separate event and outputted as such:

```
27 if self.waiting_abrupt_braking_event:
28     # Use timestamp at the last filtered data point as the current
           timestamp
29     current_ts = self.history.index[lag-1].to_pydatetime()
30     time_since_last = (current_ts -
           self.waiting_abrupt_braking_event.time).total_seconds()
31 
32     if time_since_last > self.event_duration:
33         # Enough time has passed since the waiting event to consider
               it a separate event
34 
35         # Send to plotting function in main.py for real-time plotting
36         plot_queue.put([None,
               [self.waiting_abrupt_braking_event.lat,
               self.waiting_abrupt_braking_event.long,
               self.waiting_abrupt_braking_event.time,
               self.waiting_abrupt_braking_event.severity,
               self.waiting_abrupt_braking_event.speed], None])
37 
38         event_tmp = self.waiting_abrupt_braking_event
39         self.waiting_abrupt_braking_event = None
40 
41         return event_tmp
```

Note that in addition to returning the event, to be received by **detection.py**, information about the event is also sent in a queue to **main.py** for real-time visualization[12], which will be presented in section 4.5.5. This concludes the `_detect_abrupt_braking()` function, and thus the implementation of the designed detection algorithm of table 2. It consequently achieves the most important part of point **(a)** of the problem description in section 3. Lastly, any detected events are then transmitted in a queue

[12]The library used for data plotting, Matplotlib [26], does not support plotting from threads other than the main one, so all data to be plotted must be sent to **main.py**.

from the detection module (in **detection.py**) to the weighting module:

```
70 if event:
71     logger.info(event)
72     weighting_input_queue.put(event)
```

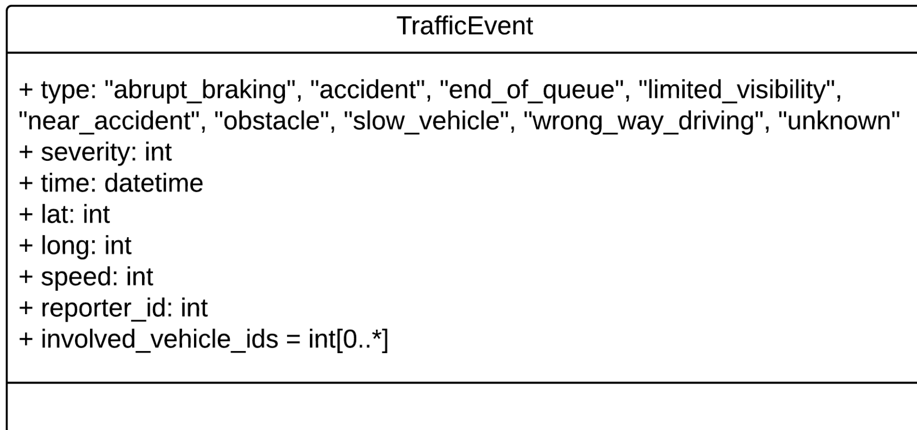


Figure 21: UML diagram showing the contents of the `TrafficEvent` class. Having no methods, it is solely a data container. Furthermore, it is built to provide a skeleton for future development, supporting several other event types in addition to abrupt braking (corresponding to some of the DENM events listed in section 4.1), as well as a container for `involved_vehicle_ids`, as a general event can conceivably involve several vehicles in addition to the one which generates the data (the *reporter*).

An alternative filtering method

In addition to the filtering method previously described (the *simple moving average*), an alternative filtering method, the *exponential moving average*, has also been implemented. Their differences are explained in section 2.2.2. This was done in order to investigate how well this latter method, which requires less computer resources in its implementation (particularly as the data volume increases significantly), performs as an approximation to the former one. The main implementation difference is in the data filtering itself, swapping the `_moving_average_filter()` function for an alternative one:

```
1  def _exp_moving_average_filter(self, last_data_point,
       new_data_point):
2      decay_factor = 0.2
3
4      if pd.isnull(last_data_point):
5          filtered_data_point = new_data_point
6      else:
7          filtered_data_point = decay_factor * new_data_point + (1.0 -
               decay_factor) * last_data_point
8      return filtered_data_point
```

The decay factor has been set to 0.2 as a reasonable approximation to the filter window size of 15 data samples, as this results in a weight of $(1 - 0.2)^{15} \approx 0.035$ placed on the 15th to last data sample.

There are some additional differences, such as not having to wait for a specific number of data points before filtering and abrupt event detection can begin. The exact implementation of this is not further detailed here, but can be found in the **vehicle.py** file. The filtering method is chosen by setting `filtering_method = "simple"` or `filtering_method = "exponential"` in **main.py**.

4.5.2 DENM handling

As shown in figure 5, there is a second data stream from the RSU to the weighting module, in parallel with the data passing through the detection module, namely a stream of DENM data. As detailed in section 4.2,

such events are already detected by something outside the scope of this system, and can thus in theory be passed directly to the weighting module for consolidation with the detected abrupt braking events. However, a simple, intermediate module had to be implemented in order to parse the DENM data output of the RSU, which is written by the **ITS_server** as log lines to a file called **its_server_denm.log**, into `TrafficEvent` objects that can be passed on to the weighting module. The input log lines are assumed to be on the following form:

```
Received DENM: { 'ts': 1525447101, 'reporter_id': 84568,
    'type': 'accident', 'lat': 634020000, 'long': 104050000,
    'severity': 0.40 }
```

Similarly to the operation of the detection module, this DENM parser module listens to incoming data in the form above, checks for proper formatting and completeness, and translates the given attributes into properties of a new `TrafficEvent` object. The object is then transmitted to the weighting module along the same queue as the detected abrupt braking events. Being able to handle DENM events on the same basis as events detected from CAM data marks the full achievement of point **(a)** of the problem description in section 3. Figure 22 shows the updated overview of the full system. The DENM parser module is implemented in the **denm_parser.py** file.

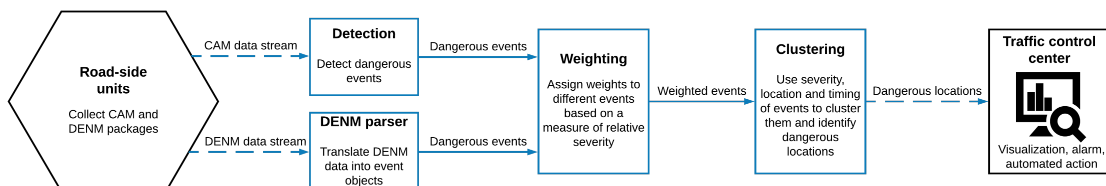


Figure 22: An updated overview of the full system, which includes the DENM parser module.

4.5.3 Weighting

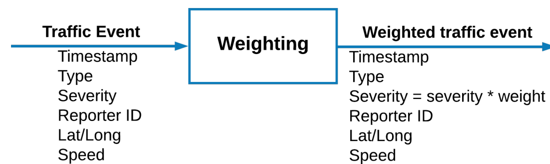


Figure 23: The input and output of the weighting module.

The purpose of the weighting module, as specified in section 4.1, is to perform a differentiation between different types of events, based on the notion that some types are more dangerous than others. The operation is illustrated by the input/output specification of the module shown in figure 23. This is implemented as a simple lookup table, which specifies a weight to be multiplied with an event's severity based on its type. The currently utilized weights are based solely on a subjective assessment of the relative danger between the different kinds of events that are currently supported by the system. These are listed in table 4. The event's severity is multiplied by the found weight, and the modified event is then transmitted to the clustering module:

```
1 try:
2     weight = weight_table[event.type]
3 except KeyError:
4     # Unknown event type
5     weight = weight_table["default"]
6
7 event.severity *= weight
8 clustering_input_queue.put(event)
```

The reason for implementing this rather simple functionality in its own module is both to keep the system's modularization simple and intuitive, and in order to lay the groundwork for what can be further developed by adding more complex functionality, for instance as proposed in section 6.2.

Table 4: The lookup table used for applying a weight to an event's severity based on its type.

Event type	**Severity weighting**
Abrupt braking	2
Accident	5
End of queue	1
Limited visibility	2
Near-accident	4
Obstacle on the road	3
Slow vehicle	1
Wrong way driving	4
Default/unknown	1

4.5.4 Clustering

Figure 24: The input and output of the clustering module.

This module is meant to implement a solution to point **(b)** of the problem description in section 3. It will take a stream of weighted events as input, perform some kind of calculation on them in order to express their combined effect in space and time (this is termed *clustering*), and produce outputs in real-time which express the calculated danger at different relevant locations. This input/output operation is illustrated in figure 24. As described in section 4.1, designing this module entails "expressing each event in space by characterizing how it affects its surroundings, and in time by characterizing how its impact changes as time goes by".

Expressing an event in space

Calculating how a traffic event affects its surroundings is certainly a complex task, probably involving about as many variables as one is able to think of; From the distance to the event, road types and configurations, to the time of day, number and types of vehicles or pedestrians present and the current weather conditions. We here opt for a simpler solution: Exponential decay along the distance vector from the event's center point, symmetrical in two dimensions. This expresses an event's *spatial impact* as a number between one and zero, with the highest impact at the center, and lower impact, converging towards zero, as one gets further away from it. This is illustrated in figure 25.

An event's spatial impact i_d at a certain distance d from its center can then be calculated as

$$i_d = e^{-\lambda_s d} \tag{9}$$

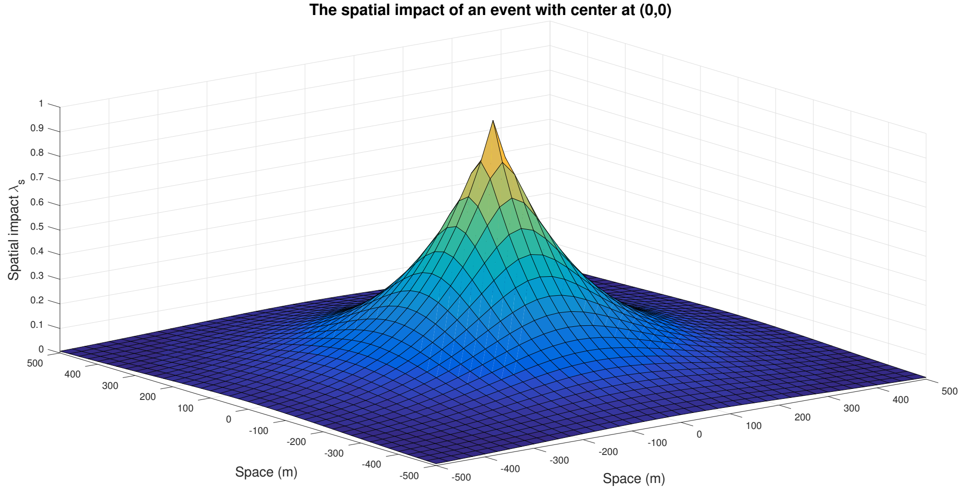


Figure 25: An event's spatial impact decays over the distance to the event's center. Here the impact is 1, while it converges towards 0 as one gets sufficiently far away.

The spatial decay rate λ_s is a design parameter that can more easily be determined by rearranging the equation:

$$\lambda_s = \frac{\ln(i_d)}{d} \tag{10}$$

so that, for instance, choosing a 50% decay, or equivalently, an impact $i_d = 0.5$, at a distance to the event center of $d = 100\ m$ results in a spatial decay rate of $\lambda_s = 0.0069$. This is the value used for the implementation.

Expressing an event in time

Similarly to how an event is modeled as having a decreasing effect on its surroundings as one moves away from it, its impact is also modeled as decreasing over time. To keep the implementation simple and intuitive, this too will be calculated using an exponential decay, entirely analogous

to the spatial impact:

$$i_t = e^{-\lambda_t t} \tag{11}$$

where i_t is the temporal impact at time t after the event's occurrence, and the temporal decay rate λ_t can be calculated similarly to the spatial one:

$$\lambda_t = \frac{\ln(i_t)}{t} \tag{12}$$

Using equation 11, an event's temporal impact will be 1 at the moment it occurs, and over time converge towards zero. Figure 26 shows this decay using $i_t = 0.5$ after $t = 10$ minutes.

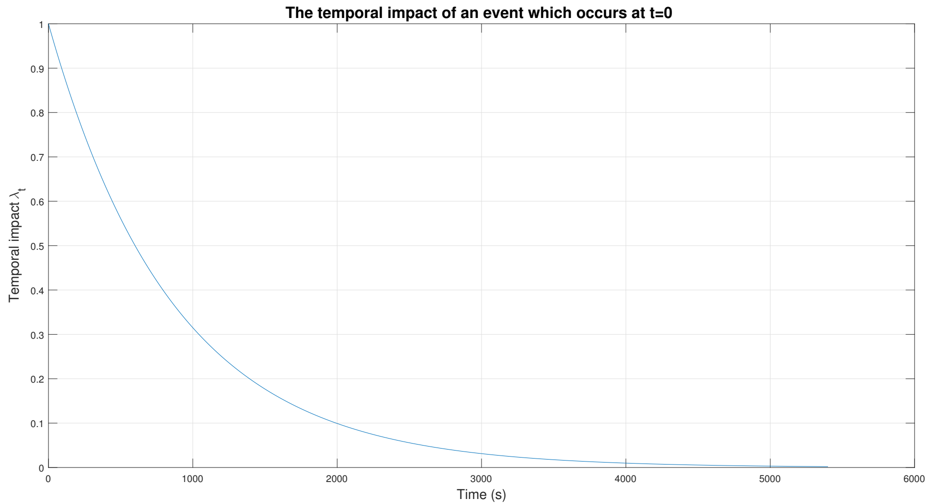


Figure 26: An event's temporal impact decays as time passes, from 1 at $t = 0$ seconds after its occurrence, towards 0. Using a decay rate $\lambda_t = 0.0012$, the event's impact is almost entirely gone after an hour.

Calculating danger

Having established the concept of an event's spatial and temporal *impact*, the *danger* D_{e,d_e,t_e} an event e inflicts on locations at a distance d_e from

its center, and a time t_e after its occurrence can easily be calculated as

$$D_{e,d_e,t_e} = s_e \; i_{d_e} \; i_{t_e} \tag{13}$$

where s_e is the (weighted) severity of the event. The total level of danger present at a certain location l at a given point in time t is then calculated as the sum of the danger inflicted on it by all events:

$$D_{l,t} = \sum_{e \,\in\, events} D_{e,d_e,t_e} \tag{14}$$

A particular location will thus experience a higher level of danger if it lies between two (or more) events that overlap sufficiently in both space and time.

Implementation

The implementation of the clustering module resides in the **clustering.py** file, and is illustrated by figure 27. As described, the module listens for events transmitted from the weighting module, and upon reception, adds the event to a list of events that the module is currently tracking. It then recalculates the level of danger over all locations in a given map using equation 14. The distance d_e from a particular location to a particular event's center is calculated using the Haversine formula, as described in section 2.2.3 and equation 6. The function `exp_decay(...)`, which is used to calculate both the spatial and the temporal impact (equations 9 and 11), incorporates a mechanism to set the calculated impact to zero when it is less than a certain threshold (set to 0.05), in order to limit the amount of negligible data points that have to be retained. The map is given as a set of boundary points for latitude and longitude, as well as a chosen granularity, resulting in a list of discrete locations uniformly distributed within the boundary.

If, during this calculation, it is discovered that the temporal impact of an event has reached zero (in practice, that it is less than 0.05, as explained above), the event is removed from the list of tracked events and effectively deemed to be "over". This makes sure that only truly relevant events continue being tracked. The duration threshold for recalculation seen in figure 27, t_r, is determined based on the temporal decay rate λ_t,

so that the level of danger on the map will be recalculated more often when the dynamics are quicker, while computing resources can be spared if events are slower to evolve. Specifically, it is set to be ten times smaller than the time it takes for 50% temporal decay.

A location is deemed to be *dangerous* if its total level of danger is larger than a threshold value `danger_threshold`, which at this point is set quite arbitrarily to 1. When there are dangerous locations present on the map, an output is generated from the module, which is also the final output from the system itself, as seen in figure 22. This is on the form of a JSON object listing the locations and their level of danger, as well as some additional, relevant data:

```
{
  "timestamp": "1527597009.254",
  "highest_danger": "3.26",
  "average_danger": "0.02",
  "dangerous_locations": {
    "63.40700,10.44100": "1.00",
    "63.40700,10.44200": "1.18",
    "63.40700,10.44300": "1.28",
    "63.40700,10.44400": "1.26",
    ...
  }
}
```

The `average_danger` is the mean level of danger across all locations in the map, and can be interpreted as a danger level for the map as a whole.

Such an output is generated every time the level of danger on the map is recalculated. A similar output, containing only the timestamp and an empty list of `dangerous_locations`, is also generated when the map becomes entirely danger free, if danger has been present previously. The output is currently only written to a log file called **output.log** (in the **log** folder), but the widely supported and language-agnostic[13] structure of the JSON object means that it could easily be implemented as the input to another system, for further processing. This fulfills point **(b)** of the problem description in section 3.

[13]Meaning that it is independent of any particular programming language, even though it has *JavaScript* in its name.

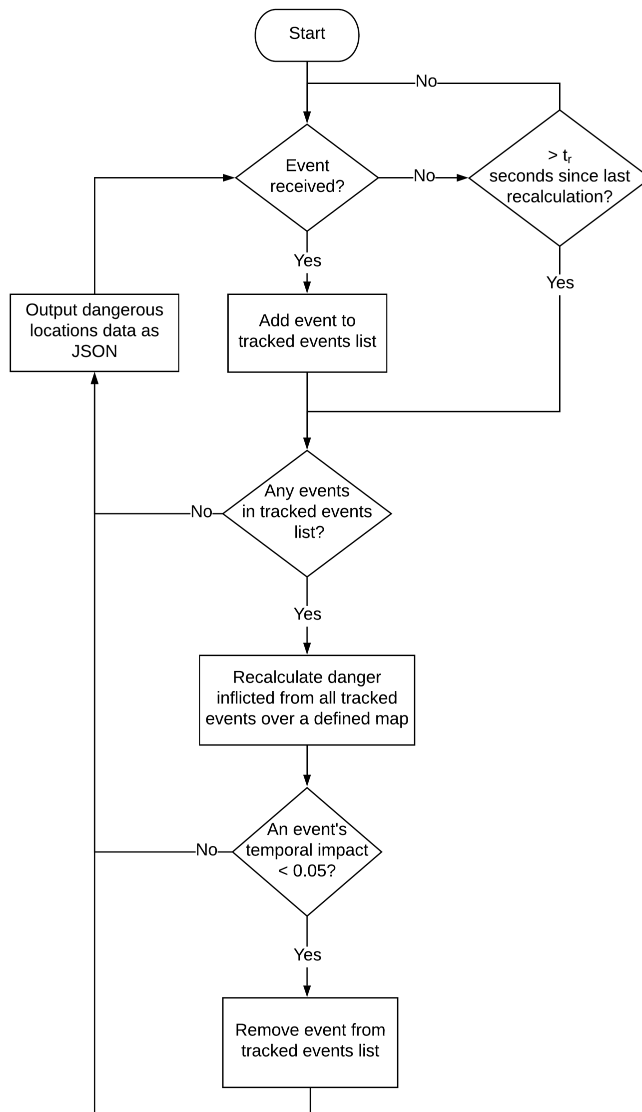


Figure 27: The inner workings of the clustering module.

4.5.5 Visualization

The developed visualization functionality is not previously shown as a separate module in figures 5 and 22, but for our purposes it largely replaces the *Traffic control center* shown in the figures. A description of its functionality is thus included in this section.

During their operation, the detection and clustering modules transmit data in a queue back to the **main.py** file for real-time visualization. Here, the plotting library Matplotlib [26] is used to display the incoming data and events, in two different views:

Speed and acceleration view
: This view, shown in figure 28, displays a vehicle's raw speed data at the top, and both raw and filtered acceleration data at the bottom. As expected, one can see in the left part of the figure that the first filtered data point appears half a filter window size after the first raw one. To the right, it is evident that the incoming filtered data stream lags half a filter window size behind the raw data, also by design (see section 4.5.1). The current filter window is shown to lie symmetrically around the last filtered value. The detection threshold is also shown in the acceleration view, and events are marked both on the speed and acceleration graphs as they are being detected, in real-time.

Map view
: The map view, as shown in figure 29, displays relevant elements on the given map. A vehicle's path is shown in blue, while event centers are marked as red dots as they are being detected. In addition, the current level of danger at all locations on the map is displayed using a heat map, with the scale shown on the bar to the right. In practice, this means that event centers are surrounded by circular, colored areas showing the event's spatial extent, which slowly fade away over time as the event's temporal impact decreases. The highest current level of danger on the map is marked on the bar in magenta, while the map's average level of danger is marked in blue.

These views are analogous to the onces created in section 4.4.3, but

are now generated in real-time as part of the overall solution. This makes it possible to easily monitor a continuously updating level of danger across all locations on a map, calculated from event detection based on C-ITS data (both CAM and DENM) that is collected in real-time by an RSU. This achieves point **(c)** of the problem description in section 3.

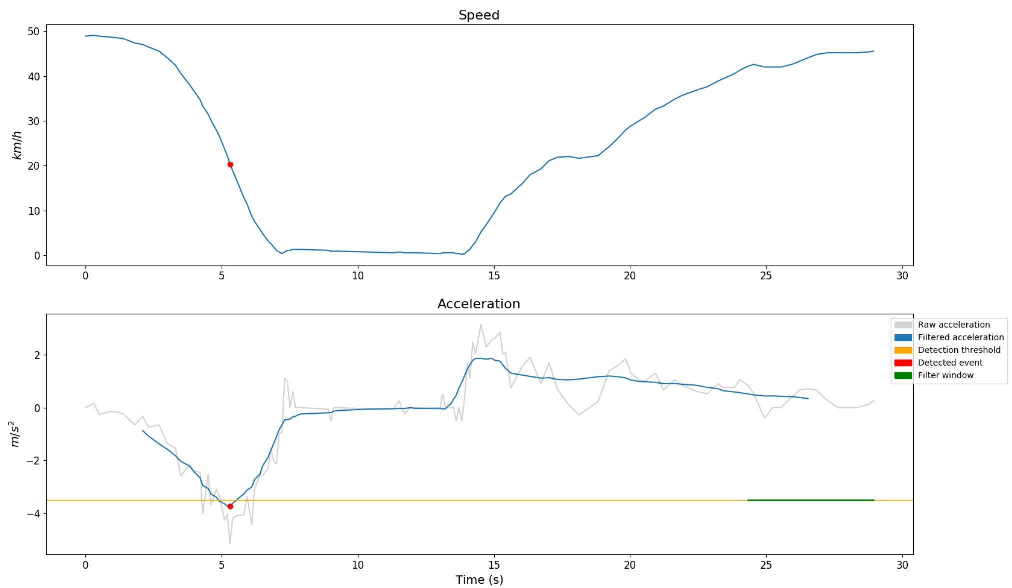


Figure 28: The real-time speed and acceleration view, showing raw and filtered data, as well as the threshold for abrupt braking detection, the current filter window, and events as they are being detected.

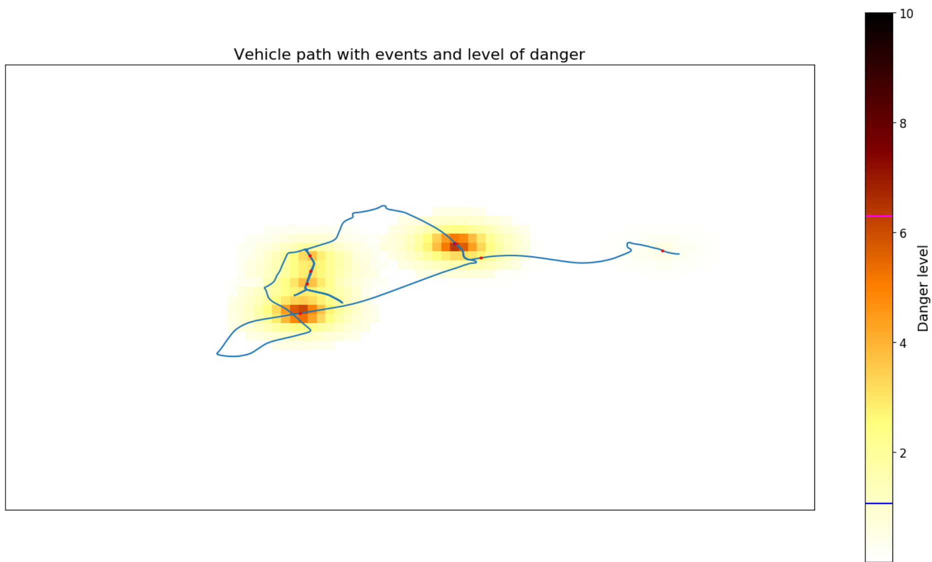


Figure 29: The real-time map view, showing a continuously updating overview of all detected events, as well as the current levels of danger across the map, and the path of the vehicle from which the abrupt braking events are detected.

5 Results and discussion

5.1 Modules

5.1.1 Detection

Testing of the detection algorithm (table 2) was performed using the test dataset, which contains a total of eight abrupt braking events. Using the algorithm parameters which were tuned to fit the training set yields the results shown in figure 30, with the total performance summed up as follows:

True positives: 3

False positives: 0

False negatives: 5

Precision: 1

Recall: 0.375

F-score: 0.54545

The perfect precision combined with the mediocre recall indicates that the algorithm has been too strict in its discrimination. In essence, it has filtered out everything but the few events it is "absolutely certain about", leading to a large number of false negatives and small number of true and false positives. The **detection_algorithm_tuning.m** script, described in section 4.5.1, can be used to determine what the optimal algorithm parameters for the test dataset would be. Those are listed in table 5.

Comparing these values to the ones determined on the basis of the training set, namely a filter window size of 15 data points and a deceleration threshold of 3.5 m/s^2, there must evidently have been a trend where the abrupt braking examples in the test set were slightly *less* abrupt than their counterparts in the training set, leading to a lower optimal threshold for the same filter window size.

Investigating the table, one can also see that a higher deceleration threshold can be compensated by a shorter filer window size. This makes perfect sense, as a shorter filter window size allows more of the highly

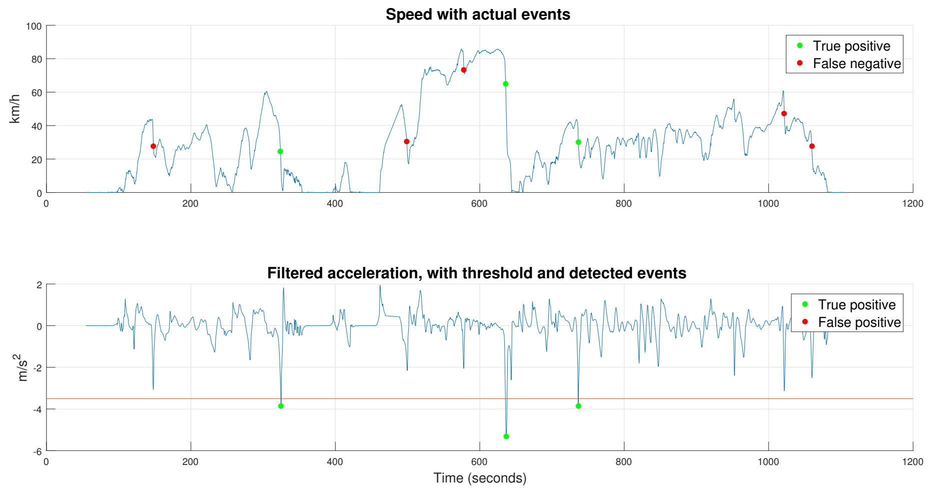


Figure 30: Running the detection algorithm on the test dataset, with color coded events marked as true positives, false negatives or false positives. The upper speed graph shows the actual events, while the lower acceleration graph shows the detected events.

Table 5: The optimal combinations of detection algorithm parameters for the test dataset, all resulting in an F-score of 0.889

Filter window size (number of data points)	**Deceleration threshold (m/s^2)**
15	2
13	2.1
13	2.2
11	2.4
9	2.6

dynamic, abrupt behavior in the raw data to get through the filter, such that the deceleration peaks become higher. This would allow for a higher deceleration threshold.

It is interesting to see the results if the training and test datasets were swapped with one another. Keeping the filter window size at 15 data points and lowering the deceleration threshold to 2 m/s^2, as per table 5, produces the following result for the training data:

True positives: 10

False positives: 12

False negatives: 1

Precision: 0.455

Recall: 0.909

F-score: 0.606

As would be expected, there is now an opposite behavior, with a low precision but high recall. Almost all the actual events are now detected, while the number of false positives increases drastically, in comparison to the algorithm's performance in section 4.5.1.

The algorithm is well-suited for implementation in a real-time system, as the amount of data that needs to be retained is static, and equal to the filter window size. As long as this is kept reasonably short, it is negligible in comparison to the total amount of received data, as illustrated in figure 31, where the current filter window is marked in green. However, the amount of data required to be retained can be reduced even further by using the alternative filtering method described at the end of section 4.5.1. This could be desirable if one is tracking a large number of vehicles simultaneously, in which case the algorithm only needs to retain a single data point per vehicle. Figure 32, which can be compared with figure 31, show that this method of filtering produces approximately the same results as the original one.

It is evident from comparing the test and training dataset results that truly accurate abrupt braking detection is not as clear-cut as applying a simple threshold on the filtered deceleration data, as even the same driver

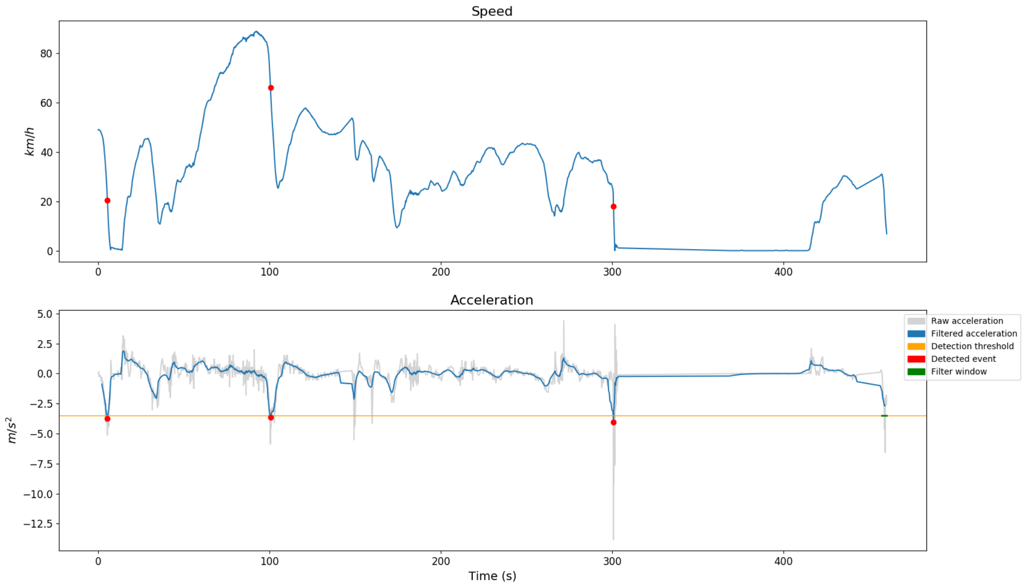


Figure 31: A visualization of the detection module in real-time operation. The filter window, representing the data that needs to be retained in order for the algorithm to function, is evidently negligible in comparison to the total data volume.

in the same vehicle on the same day will evidently perform the maneuver with enough variance to throw the algorithm off. Even so, the designed algorithm was probably about the best one could do with the small amount of data that was available.

This lack of a large quality dataset on which to base the development and tuning of an event detection algorithm is arguably the greatest weakness of this work. The fact that such data had to be generated manually, and in order for that part of the work to not fully consume all available hours, made it necessary to settle for the detection of a single, simple type of event, as explained in section 4.2, using simple methods.

In addition, the implementation serves well as a first proof-of-concept

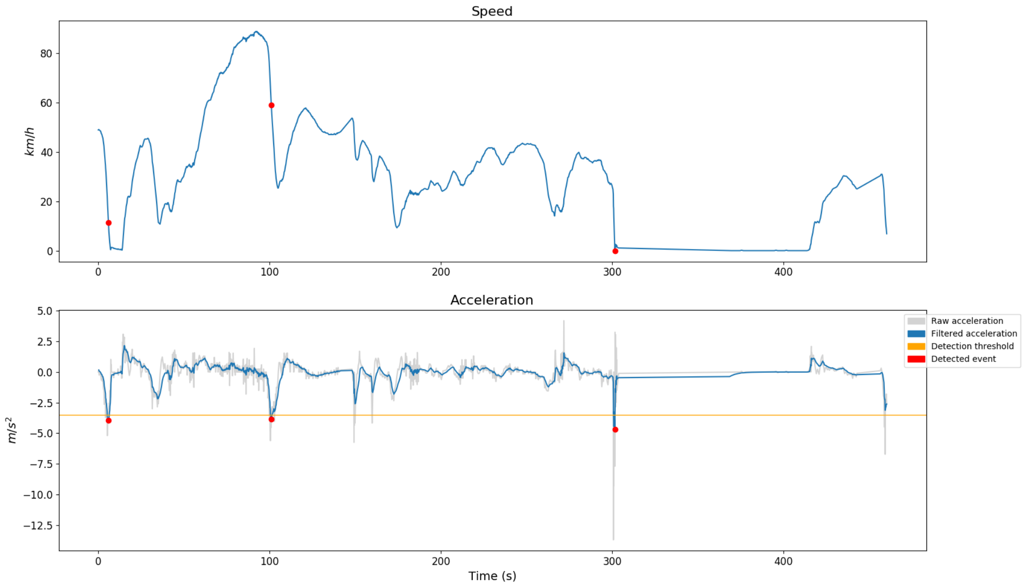


Figure 32: The detection module operating with the alternative filtering method. The operation is almost indistinguishable from figure 31, where the original filtering method is used.

for the detection module, showing how event detection with basis in CAM data can be used to supply a larger data processing system with useful traffic information in real-time.

5.1.2 Clustering

The clustering module was tested for its ability to handle a large amount of different types of events, using the CAM replay script described in section 4.4.3 in combination with the DENM generator described in section 4.4.2, which was set to output an event every second. As shown in figure 33, it handles all types of events equally, with no differentiation on whether they originated from CAM or DENM data. The exponential de-

cay function (equations 9 and 11) serves well as a simple way to express events in space and time. Figure 34 shows that the danger level increases at locations where several events overlap. This confirms that the clustering module functions well as a way to combine the severity of different types of detected events, and finally produce an assessment of the level of danger at different locations in a map. This is in line with the simple assumption stated in section 4.1.

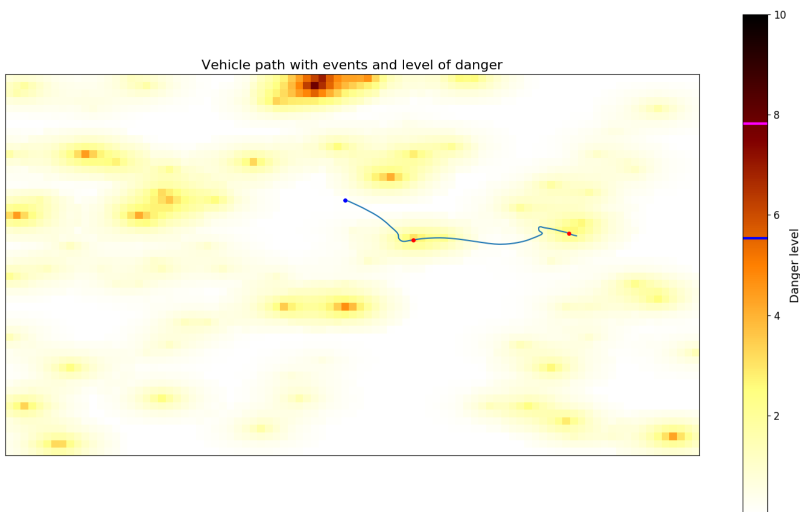


Figure 33: The clustering module in real-time operation, with randomly generated DENM events sprinkled on top of the detected abrupt braking events from a vehicle's CAM data. The highest current level of danger on the map is marked on the bar in magenta, while the map's average level of danger is marked in blue.

The output of dangerous locations functions as intended, printing JSON objects to a log file as long as there is any danger present on the map, and finally printing an empty list once all danger is gone:

```
{
    "timestamp": "1527773122.941",
    "highest_danger": "1.05",
    "average_danger": "0.01",
    "dangerous_locations": {
    "63.40900,10.44300": "1.05",
    "63.40900,10.44400": "1.02"
    }
}

...

{
    "timestamp": "1527773123.505",
    "dangerous_locations": {}
}
```

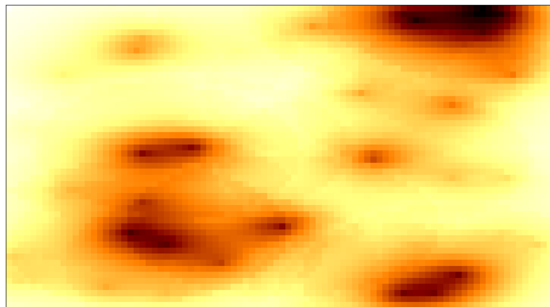


Figure 34: A zoomed view of the map. Several events overlapping increases the level of danger.

5.1.3 Visualization

The real-time visualization of the system's operation, as already shown in figures 31-34 is rudimentary but functional at this stage. The visualization of the data coming from a specific vehicle is created mostly for testing purposes; as such it only supports data coming from a single vehicle, though modification to support several should not be too difficult.

The map view (figure 33) serves as a visualization of the main goal of this work as stated in the problem description in section 3, namely to determine dangerous locations in traffic in real-time. It supports event data originating from any kind of source, be it different vehicles or RSUs. It was attempted to incorporate a road map on which the data could be overlaid, similarly to the OSM map used in figure 8. Unfortunately, this proved too large of an obstacle for too little gain, and had to be abandoned.

Large amounts of event data inputted over a given timespan causes the visualization to lag somewhat behind the data processing itself. This is both due to the inefficient way that the level of danger is calculated over the map, and due to heavy calculations performed by the plotting library. It is thus unlikely that it would be able to handle truly massive amounts of data in real-time without some modification.

5.2 Overall system

Figure 35 shows how data propagates through all modules of the system, along the paths previously illustrated in figure 22. The data is successfully handled seamlessly through the system, from input in the form of CAM and DENM data streams from an RSU, to an output in the form of a continuously updating stream of current dangerous locations and their level of danger.

Unfortunately, there was not enough time to perform a test of the system in true real-time operation. Such a test would involve the same setup as described in section 4.3, this time with the system being fed CAM and DENM data in real-time for event detection and subsequent output of dangerous locations. However, replaying datasets using the functionality developed in section 4.4.3 proves that this would work as expected, as there are no differences between this and true real-time operation.

Figure 35: Logs for each of the modules, showing the data propagation through the system.

6 Further work

6.1 Detection

As previously explained, the design of event detection algorithms must be data-driven, meaning that it is based on empirical data acquired from real-life observations of the phenomena that are to be detected. As stated in section 5.1.1, the lack of large amounts of such data represents the greatest weakness of this work. Consequently, this also represents the greatest potential for future improvement. As calculated in [1], the accumulated volumes of CAM and DENM data generated in real-life traffic will quickly rise to the levels of terabytes, petabytes and exabytes once the C-ITS technology matures and starts being embedded in actual vehicles on the road. Such datasets would enable vastly more advanced detection methods to be used, resulting in more accurate detection of both abrupt braking and more complex types of events.

Improving the detection of abrupt braking events

The detection of the simple abrupt braking event type can surely be improved by using a larger amount of data to tune and test the developed algorithm, finding values for the tunable parameters that fit even better. However, as alluded to in section 5.1.1, it does not seem like this alone will allow the accuracy of the detection to reach satisfactory levels. Once in possession of greater volumes of data, one should thus re-do a thorough analysis of the data, to investigate whether there are other variables and methods that could be suited for achieving a higher accuracy. For instance, perhaps one would benefit from making the deceleration threshold dynamic, taking into account e.g. the road type and number of vehicles present nearby, and the speed of the vehicle performing the braking maneuver.

Detecting other types of events

Abrupt braking events were chosen, among other things, for their ability to be detected using only the *speed* variable (see section 4.2). More complex events, such as near-collisions between vehicles, illegal maneuvers or

reckless driving, depend on a much higher number of variables, with intricate relations between them. Even if one had all the relevant variables available for measurement, it is unlikely that it is even possible for humans to analyze and manually configure a detection algorithm for such complex types of events.

Fortunately, the emergence in later years of increasingly more sophisticated machine learning methods pose a perfect fit for this problem, once the volume and quality of the available datasets reach sufficient levels. Where humans quickly lose control when faced with large volumes of data, computers thrive in such conditions, performing better the more data they are fed. Machine learning detection methods, such as neural networks, would be able to handle a large number of different variables, and identify perhaps unintuitive correlations between them.

6.2 Weighting

As stated in section 4.5.3, the weights currently used to differentiate between the severity of different types of events are purely based on a subjective assessment. The first improvement of the weighting module would be to base these values on empirical observations. Going further, the weighting could be changed from using statically defined values to dynamically calculated ones. Events could be weighted differently according to the time of day, road conditions and so on. One could also implement a feedback from the clustering module, such that, for instance, subsequent events happing at the same location are given increasingly large weights. All this, of course, places even higher demands on the availability of empirical data supporting such adjustments.

6.3 Clustering

There are a number of optimizations that can be performed on the current implementation of the clustering module in order to make it more efficient in its calculations. At present, on recalculation of the level of danger across a map, an iteration is performed once per event over the entire map, as such:

```python
for lat in np.arange(lat_min, lat_max, lat_step_size):
    for lon in np.arange(lon_min, lon_max, lon_step_size):
        for event in events:
            # Calculate danger inflicted by event to location (lat,
                lon) and add to total danger at location (lat, lon)
            ...
            ...
```

This places limitations, mostly on the granularity of the map, in order for the calculation to not become too slow. An improvement could be made in order to reduce unnecessary calculations, for instance by identifying the boundaries of events, and not calculate the danger inflicted by an event outside of these boundaries (where it will be zero regardless).

For further development, there is a large amount of potential in using empirical data to express events more precisely in space and time. Especially in the way events are modeled with symmetrical spatial decay is likely to be quite inaccurate in the real world. An improvement would be to use knowledge of the road infrastructure to extend an event's impact along the nearby roads, perhaps only in one particular direction of travel. For instance, a severe event on a highway can have ramifications along its entirety, with its impact stretching tens or hundreds of kilometers along it. At the same time, it might have negligible impact on traffic traveling in opposite lanes along the same highway.

6.4 Visualization

As previously explained, parts of the visualization solution is built mostly to be of help during development, debugging and tuning of the entire system. Future development would demand even better methods of visualization. An obvious extension is to build support for visualizing data coming from multiple vehicles in parallel. Another is to incorporate the display of a road map in the map view, on which the data can be overlaid.

References

[1] L. Hellesylt, "A distributed storage and analysis solution for vehicle data from Cooperative Intelligent Transport Systems," 2017-12.

[2] "Directive 2010/40/EU of the European Parliament and of the Council on the framework for the deployment of Intelligent Transport Systems in the field of road transport and for interfaces with other modes of transport," *Official Journal of the European Union*, vol. L 207/1, 2010. [Online]. Available: http://eur-lex.europa.eu/LexUriServ/LexUriServ.do?uri=OJ:L:2010:207:0001:0013:EN:PDF

[3] "Automotive Intelligent Transport Systems," Accessed: 2017-12-03. [Online]. Available: http://www.etsi.org/technologies-clusters/technologies/automotive-intelligent-transport

[4] Stortingsforhandlinger, "Nasjonal transportplan 2014-2023," *Meld. St. 26 (2012-2013)*, p. 191. [Online]. Available: https://www.regjeringen.no/no/dokumenter/meld-st-26-20122013/id722102/

[5] *Intelligent Transport Systems (ITS); Access layer specification for Intelligent Transport Systems operating in the 5 GHz frequency band*, Std. (Draft) ETSI EN 302 663, 2012-11, v1.2.0. [Online]. Available: http://www.etsi.org/deliver/etsi_en/302600_302699/302663/01.02.00_20/en_302663v010200a.pdf

[6] *Intelligent Transport Systems (ITS); V2X Applications; Part 3: Longitudal Collision Risk Warning (LCRW) application requirements specification*, Std. ETSI TS 101 539-3, 2013-11, v1.1.1. [Online]. Available: http://www.etsi.org/deliver/etsi_ts/101500_101599/10153903/01.01.01_60/ts_10153903v010101p.pdf

[7] *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 2: Specification of Cooperative Awareness Basic Service*, Std. ETSI EN 302 637-2, 2014-11, v1.3.2. [Online]. Available: http://www.etsi.org/deliver/etsi_en/302600_302699/30263702/01.03.02_60/en_30263702v010302p.pdf

[8] *Intelligent Transport Systems (ITS); Vehicular Communications; Basic Set of Applications; Part 3: Specification of Decentralized Environmental Notification Basic Service*, Std. ETSI EN 302 637-3, 2014-11, v1.2.2. [Online]. Available: http://www.etsi.org/deliver/etsi_en/302600_302699/30263703/01.02.02_60/en_30263703v010202p.pdf

[9] P. Greibe, "Braking distance, friction and behaviour: Findings, analyses and recommendations based on braking trials," *Trafitec*, 2007.

[10] American Association of State Highway and Transportation Officials, *A policy on geometric design of highways and streets*, 4th ed., 2001.

[11] L. Hoberock and U. S. D. of Transportation. Office of University Research, *A Survey of Longitudinal Acceleration Comfort Studies in Ground Transportation Vehicles*, ser. Research report. Council for advanced transportation Studies, University of Texas at Austin, 1976.

[12] T. Hiraoka, T. Kunimatsu, O. Nishihara, and H. Kumamoto, "Modeling of driver following behavior based on minimum-jerk theory," 2005.

[13] E. E. Wilson and R. A. Moyer, "Deceleration distances for high speed vehicles," *Proceedings of the Highway Research Record*, vol. 20, pp. 393–398, 1941.

[14] Walber, "Precisionrecall.svg," Accessed: 2018-05-10. [Online]. Available: https://commons.wikimedia.org/wiki/File:Precisionrecall.svg

[15] R. W. Sinnott, "Virtues of the Haversine," *Sky and Telescope*, vol. 68, no. 2, 1984. [Online]. Available: http://daimi.au.dk/~dam/thesis/Sky_and_Telescope_1984.pdf

[16] Chamberlain, Robert G., "What is the best way to calculate the great circle distance (which deliberately ignores elevation differences) between 2 points?" Accessed: 2018-05-08. [Online]. Available: http://www.faqs.org/faqs/geography/infosystems-faq/

[17] H. Moritz, "Geodetic reference system 1980," *Journal of Geodesy*, vol. 74, no. 1, pp. 128–133, Mar 2000. [Online]. Available: https://doi.org/10.1007/s001900050278

[18] Kapsch TrafficCom AG. EVK-3300 V2X Evaluation Kit. Accessed: 2017-12-12. [Online]. Available: https://www.kapsch.net/KapschInternet/media/Press/ktc/KTC_EVK-3300.jpg

[19] Kapsch TrafficCom, "EVK-3300 V2X Evaluation Kit," https://www.kapsch.net/ktc/downloads/datasheets/in-vehicle/5-9/KTC_DB_EVK-3300_17_web.pdf, Accessed: 2017-12-09.

[20] Transportøkonomisk institutt, "Ulykker og risiko i vegtrafikken," Accessed: 2018-05-13. [Online]. Available: https://tsh.toi.no/?21291

[21] Python Software Foundation, "Python 3.6." [Online]. Available: http://www.python.org

[22] The MathWorks Inc., "MATLAB R2016a." [Online]. Available: https://www.mathworks.com/products/matlab.html

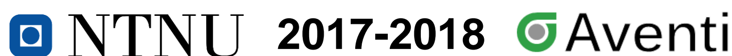
[23] OpenStreetMap contributors, "Planet dump retrieved from https://planet.osm.org ," https://www.openstreetmap.org, 2017.

[24] PyPA, "pip 9.0.3." [Online]. Available: https://pip.pypa.io

[25] W. McKinney, "Data Structures for Statistical Computing in Python," in *Proceedings of the 9th Python in Science Conference*, S. van der Walt and J. Millman, Eds., 2010, pp. 51 – 56. [Online]. Available: https://pandas.pydata.org/

[26] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing In Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

A Original problem description

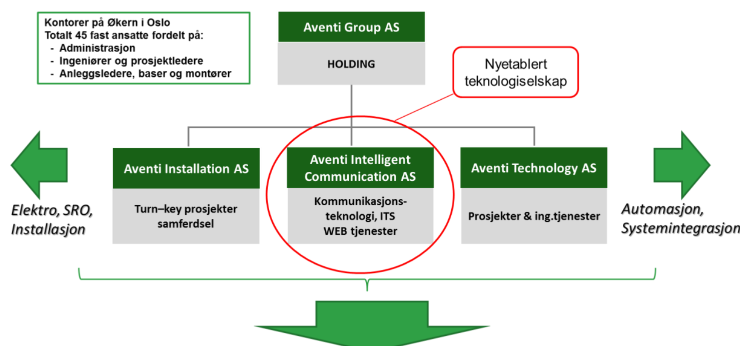


Forslag til Prosjekt- og Masteroppgave

Studentoppgave nr.2 – Forutse trafikkulykker før de skjer

Om Aventi Intelligent Communication AS

Aventi Group AS holder til i Oslo, og består av de tre selskapene Aventi Technology AS (AT), Aventi Installation AS (AI) og Aventi Intelligent Communication AS (AIC). Sistnevnte står for denne studentoppgaven.



AI jobber med elektroinstallasjoner for veier, tunneler og bruer, mens AT foretar engineering og programmering av dette utstyret, og knytter det opp mot Veitrafikksentralene. For eksempel så har alt utstyr i tak, vegger og tekniske rom i Strindheimstunnelen i Trondheim blitt installert av Aventi.

Disse systemene er alt veldig smarte og har blant annet videosystem som gir automatisk alarm dersom en bil kjører i feil retning, dersom en person vandrer i tunnelen eller dersom en kasse har falt av en lastebil. Men dette er ingenting i forhold til de trafikkstyresystem vi nå ser i horisonten. Om få år vil vi se de første autonome bilene i nyttetrafikk på norske veier, og da skal Aventi være klar med løsninger for autonome veier. Derfor har Aventi Group opprettet Aventi Intelligent Communication som de neste fem årene skal delta sammen med Statens Vegvesen, Sintef, NTNU og mange andre i forsknings- og utviklingsprosjekt for Connected Vehicles (CV), Cooperative Intelligent Transport Systems (C-ITS), Internet of Things (IoT), Location Based Services (LBS), Big Data, Machine Learning, Cloud Computing og Edge Computing. AIC sine forskningsprosjekt vil bli knyttet opp mot de reelle veiprosjektene til AT og AI, slik at vi får testet ting under virkelige forhold

Aventi har gode erfaringer fra tidligere master-oppgaver, og har på den måten funnet både gode tekniske løsninger og noen av sine beste medarbeidere. Nå prøver vi igjen, for nå skal vi bygge opp et helt nytt selskap.

Om oppgaven

I et kommuniké fra EU kommisjonen 30.november 2016 bes det om at alle land i Europa, inkludert Norge, begynner utrulling av C-ITS for *Connected Vehicles* og *Autonomous Vehicles* i 2019:
http://europa.eu/rapid/press-release_MEMO-16-3933_en.htm

C-ITS er en standardisert måte for biler av forskjellige typer (Volvo, Audi, Opel) å kommunisere på, der de utveksler såkalte CAM pakker hvert sekund. En slik pakke er ca. 350 bytes og inneholder posisjonskoordinater, hastighet, retning, data hentet ut fra kjøretøyets OBD-II port og mye mer. I tillegg kan bilene og infrastruktur (veikantstasjoner, tunneler, bruer, veiarbeidsområder) utveksle såkalte DENM pakker som inneholder trafikkmeldinger. Disse er også ca. 350 bytes, men sendes kun ut ved behov. Standardene som beskriver alt dette finner man her:
https://goo.gl/Mouv8r

For å kommunisere brukes ETSI-ITS-G5 radioer. Tanken er at disse vil være innbygd i nye biler fra ca. 2020. For eldre kjøretøy kan de bli ettermontert. Teknologien er basert på IEEE 802.11, tilsvarende vanlige Wi-Fi, men med noen justeringer som illustrert nedenfor.

Wifi – Router		**G5-radios**	
IEEE 802.11 a/b/g/n		IEEE 802.11 p/ocb	
SSID:	My-Wifi	SSID:	N/A
WPA2:	password	WPA2:	N/A
Freq.:	2.4GHz and 5GHz	Freq.:	5.9GHz
Max:	20 dBm	Max:	33 dBm (1km range)
Comm:	many to one	Comm:	many to many

Aventi har alt jobbet med denne teknologien i flere år både med hands-on programmering og deltakelse i forskjellige fora. Her er en YouTube video fra en av våre C-ITS tester:
https://www.youtube.com/watch?v=Nv7z_Xjkj_k

Når ETSI-ITS-G5 radioene sender ut og tar imot CAM og DENM pakker, så lagres alle disse i PCAP filer som kan åpnes og analyseres i Wireshark. Og det er nettopp det vi ønsker med denne oppgaven. Men i stedet for å lete igjennom titusenvis av CAM og DENM pakker manuelt, så ønsker vi å benytte en Big Data løsning som for eksempel Hadoop.

© Aventi Group AS

Første utfordring:

1. Studenten og noen hjelpere kjører rundt omkring på NTNU campus med biler (eller lekebiler) utstyrt med ETSI-ITS-G5 radioer (I tillegg kan det settes opp en radio som veikantstasjon), der det genereres CAM og DENM pakker. Se YouTube video ovenfor som viser denne kjøringen.
2. Etter en god stund med kjøring, så dumpes alle PCAP-filer inn i Hadoop (eventuelt Azure Machine Learning e.l.).
3. Der lages det queries som forsøker å hente ut nyttig data som f.eks.:
    a. Minimum, maksimum og middels hastigheter.
    b. $C0_2$ produksjon
    c. Radiosignalstyrke
    d. Antall ulykker og trafikkmeldinger av forskjellige typer (DENM)
4. Resultatene sammenlignes med håndnotater gjort underveis.

Andre utfordring:

1. Studenten og noen hjelpere kjører rundt og rundt i samme mønster om og om igjen. Dersom for eksempel den røde bilen kjører vestover på Kolbjørn Hejes vei, mens den blå og den gule kjører sørover på O. S. Bragstads Plass, ja så gjentas altså dette kjøremønsteret hundre ganger. Bilene vil hele tiden generere CAM pakker som lagres i PCAP-filene. Når den røde og den blå bilen møtes i veikrysset, så kolliderer de. Dette indikeres ved at man trykker på DENM for Accident på Android nettbrettet (se YouTube videoen). DENM pakkene vil også bli lagret i alle kjøretøyenes (og veikantstasjonens) PCAP-filer.
2. Studenten kan også velge å simulere all denne kjøringen ved å overstyre GPS koordinatene i C-koden til radioene (veldig enkelt). Da mistes imidlertid variasjoner i radiostyrke, GPS-signal og andre ting som oppstår når man kjører innimellom bygningene, og det kan jo tenkes at dette er ting maskinlæringsalgoritmene kunne tatt tak i.
3. Vel, atter en gang så dumpes alle PCAP filene inn i Hadoop, men denne gangen skal man kjøre maskinlæringsalgoritmer som gjenkjenner kjøremønsteret for de tre bilene, og som ledet opp til en ulykke i form av en DENM for Accident.
4. Når studenten føler seg sikker på at algoritmen kan gjenkjenne kjøremønsteret som ledet opp til ulykken, så skal det utvikles en live løsning som samler inn CAM og DENM pakker fortløpende (og ikke PCAP filer fra de siste par timer). Dette kan gjøres via et JSON interface på veikantstasjonen. Når så studenten og hjelperne denne gangen kjører den røde bilen vestover på Kolbjørn Hejes vei, og den blå og den gule sørover på O. S. Bragstads Plass, så skal systemet varsle om at kollisjon er nært forestående.

Prosjektstyring

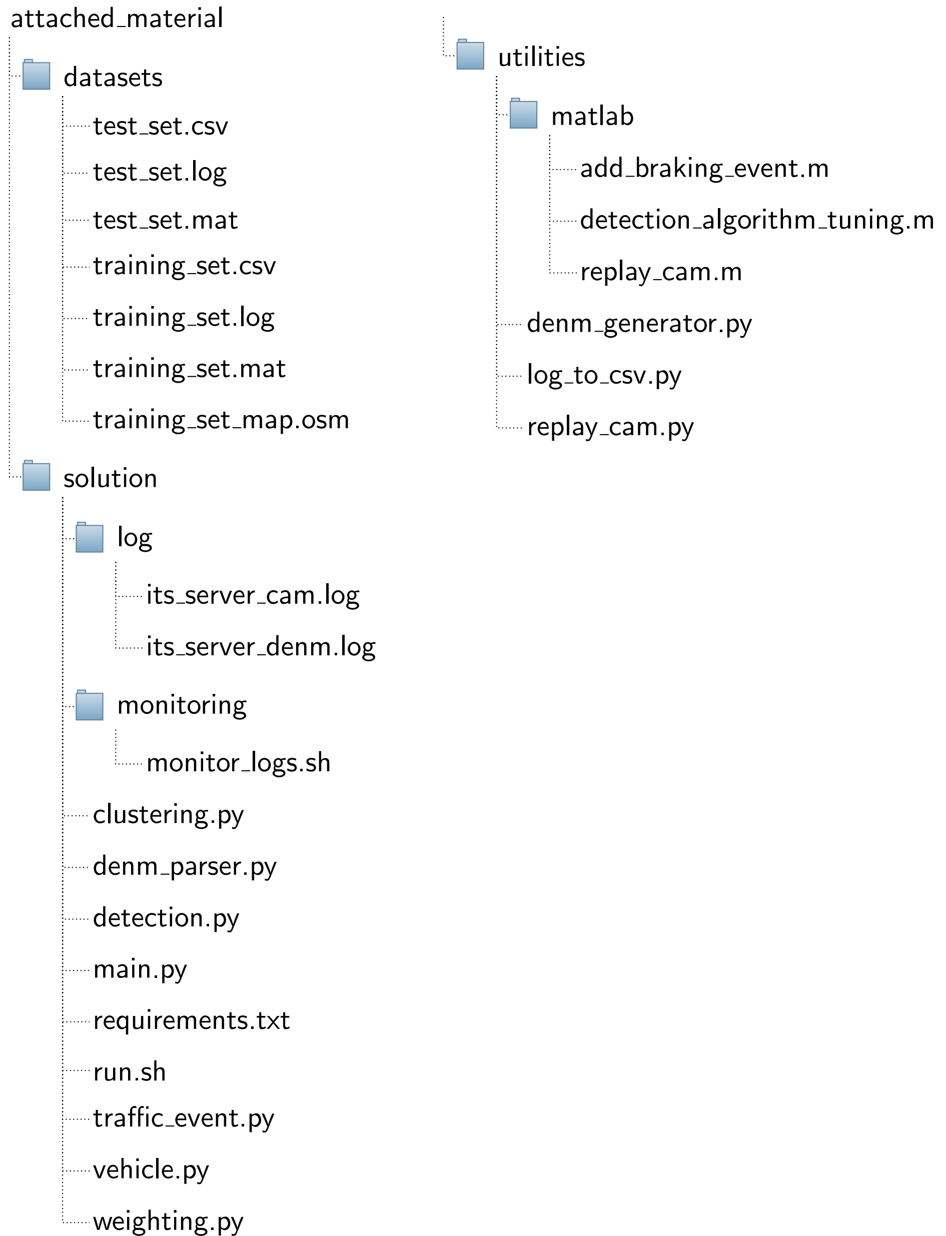
I forbindelse med denne oppgaven, så ønsker Aventi samtidig å høste erfaring med Scrum/Agile prosjektstyring. Det betyr at vi ikke ønsker at studenten skal bruke tiden fram til jul for «engineere» hele løsningen på papir, og deretter å implementere løsningen utover våren. I stedet så ønsker vi at oppgaven deles opp i såkalte «sprints», der studenten gjør ferdig verdifulle delløsninger underveis. Vi foreslår sprints på ca. en måned, og stand-up møter to ganger i uken. Vi ønsker at prosjekt-kommunikasjonen fortrinnsvis foregår i Ryver og at prosjektstyringen skjer i Axosoft eller lignende.

Kontaktpersoner

Rolle	Navn	Telefon	Epost
Student			
Faglærer			
Faglig veileder			
Aventi kontakt	Terje Hundere	+47 90 59 99 60	Terje.Hundere@aventi.no

© Aventi Group AS

B File structure of attached digital material



C Acceleration unit conversion

The simple figure below is included in order to provide a quick way of converting between the two acceleration units m/s^2 and $km/h/s$. The former is used throughout the thesis, while the latter is easier to relate to for humans used to traveling by car. For instance, a deceleration of 10 $km/h/s$ means slowing down by 10 km/h over a period of 1 second.