Strym: A Python Package for Real-time CAN Data Logging, Analysis and Visualization to Work with USB-CAN Interface

Rahul Bhadani
Electrical & Computer Engineering
The University of Arizona
Tucson, Arizona, USA
rahulbhadani@email.arizona.edu

Matt Bunting
Institute of Software Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
matthew.r.bunting@vanderbilt.edu

Matthew Nice
Civil & Environmental Engineering
Vanderbilt University
Nashville, Tennessee, USA
matthew.nice@vanderbilt.edu

Ngoc Minh Tran
Computer Science
The University of Arizona
Tucson, Arizona, USA
leahtran193@email.arizona.edu

Safwan Elmadani
Electrical & Computer Engineering
The University of Arizona
Tucson, Arizona, USA
safwanelmadani@email.arizona.edu

Dan Work
Civil & Environmental Engineering
Vanderbilt University
Nashville, Tennessee, USA
dan.work@vanderbilt.edu

Jonathan Sprinkle
Computer Science
Vanderbilt University
Nashville, Tennessee, USA
jonathan.sprinkle@vanderbilt.edu

***Abstract*—In this report, we describe a data analysis tool developed for decoding and analyzing vehicle data obtained from a passenger vehicle's onboard controller area network (CAN) bus. The tool developed in this paper provides a timeseries framework to perform domain-specific analysis at scale when interpreting data from a vehicle or a collection of vehicles in light of how to design intelligent vehicle applications. The tool, called Strym, exploits the CAN bus mechanism of modern vehicles to capture data using commercially available CAN-to-USB hardware Comma.ai Panda devices, managed through open-source software Libpanda. Strym permits the decoding of vendor-specific CAN messages in a vehicle-agnostic manner. Through this, a researcher can characterize data throughput, assess data quality, and perform analyses. Such analyses are useful in a number of research such as studying human driving behavior in mixed-autonomy, new driver models, rare-event detection, traffic flow estimation, and custom control of vehicles.**

I. Introduction

The emergence of advanced driving assistance systems and automation beyond SAE (Society of Automotive Engineers) Level II [1] on passenger vehicles are encouraging researchers in academia and industry to solve transportation problems using these new features. One such use of automated vehicle technology is to influence bulk traffic flow for energy efficiency and traffic throughput by a sparse number of automated vehicles [2, 3, 4, 5]. Advancements in vehicle autonomy in the era of the DARPA Grand Challenge [6, 7, 8] were made possible through research testbeds that depended on high-cost sensors and customized actuators on modified vehicle platforms. As continued investments into autonomy frameworks matured, levels of autonomy such as those defined by SAE began to lay the groundwork for incremental advancements in technology. Actuation algorithms currently on passenger vehicles might not permit full autonomy, but in many cases, sensors such as radars and cameras are at least as sophisticated as the sensors used in early autonomy demonstrations.

Through the industry standard Controller Area Network (CAN) bus, sensors on modern vehicles are interconnected with one another along with advanced actuators that carry out the control of the vehicle when in operation. From a research perspective, the wide availability of vehicles with these advanced sensing capabilities provides an opportunity to explore data from the platforms. These opportunities may include research into how drivers use advanced features, the development of new intelligent driver models, gathering or identification of rare events, prediction of traffic state, and other kinds of research.

However, each vehicle manufacturer utilizes their own encoding of local network information into the CAN bus on their vehicles. In order to explore the behavior of vehicles at scale, or to develop manufacturer-independent algorithms that analyze data from an array of vehicles, new algorithms and architectures are needed. These approaches could provide quality checks, post-processing algorithms and metrics, and joint analyses with additional modalities such as GPS data from augmented sensors, in a vendor-agnostic manner.

This paper describes an open-source software package, Strym, which provides libraries for real-time data logging, file-based data analysis, asynchronous and interleaved message processing, and domain-specific visualization of vehicle data. Strym is designed to support CAN bus messages from multiple makes and models in a vendor-agnostic manner. Such a design allows us to look at a broader set of vehicle data to answer questions related to driving behavior, rare event discovery,

motion planning, control decisions, observer design, estimation algorithms, etc. Strym also facilitates posthoc analyses of controllers using CAN messages after conducting the driving experiments. At the end of the paper, we provide use-cases where we used Strym for validation of vehicle's state such relative velocity, estimation of leader car's velocity, characteristics of a driver, and exploring CAN data for additional vehicle-related information.

Contribution

The purpose of this work is to describe how Strym enables downstream analyses of CAN bus vehicle data obtained from onboard sensors along with a few other modalities such as GPS data. Analyses of CAN messages from vehicles further contribute to the understanding of human-driving behavior in an array of driving conditions, vehicle-to-vehicle (V2V) interaction, developing novel mathematical models of driving under varying spatiotemporal conditions, and low-cost feedback vehicle control applications. We first describe an overview of the CAN bus in Section II. Next, in Section III, we provide a data quality assessment of CAN messages with Strym. In sections IV-VII, we discuss Strym python package and its various modules in detail. In Section VIII, we provide some use cases that enabled analysis of driving behavior, established relationships between various signals, and performed some operations needed for the data-driven development of autonomous vehicle control. We end the paper with a discussion and conclusion in Section IX.

II. Overview of CAN Bus messages

Most modern cars use either a CAN bus, or a time-triggered bus such as FlexRay [9] to transmit data between Electronic Control Units (ECUs). These vehicle data adopt a CAN protocol for encoding vehicle sensor reading such as vehicle speed, accelerometer, fuel rate, gas pedal, and throttle in addition to less safety-critical messages such as status of window-panes, brake lights, etc. In recent years, car-makers introduced several onboard sensors for driving assistance [10] and safety measures such as anti-lock brake systems, airbags, lane-keeping assistance, etc. that use CAN bus for communication and control. On-board vehicle sensors may be augmented with additional modalities such as dashcam, radar, wearable devices [11] for personal or research use. Vehicle data from multiple modalities allows for new efforts to develop novel models for traffic behavior at different geographical locations and time points, as well as what factors are consequential of human driving behavior [12].

CAN bus was introduced by Bosch at the SAE conference in 1986 [13]. In 2012, Bosch released CAN with Flexible Data-Rate 1.0 (CAN FD) which can achieve 5 Mbps in practice and has a 64-byte payload compared to 8 bytes in the initial CAN specification. The CAN bus was initially used for ECUs and emission tracking, but in recent years additional features such as the Advanced Driving Assistance System (ADAS) have made use of CAN to listen to radar proximity data and send control commands to regulate throttle and braking [14]. In addition to that, features such as Lane-Keeping Assist (LKA) are used to operate steering torque which leverages CAN buses [15]. CAN messages captured from vehicles vary in encoding, and per manufacturer, make, model and year. As per CAN protocol, messages are prioritized based on IDs. A standard CAN message packet structure is shown in Figure 1.

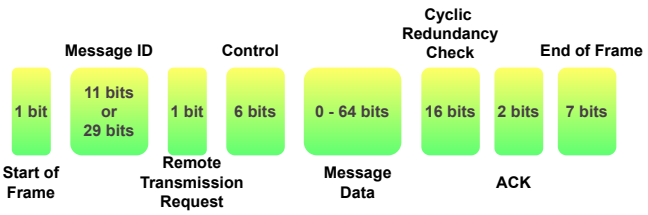

Fig. 1: Packet structure of a standard CAN message. The first part consists of 1 bit that denotes the start of the packet frame. It is a dominant 0 that tells other ECUs that a message is coming. The second part, either 11 bits for CAN 2.0A or 29 bits for CAN 2.0B, is used for message identifier, lower identifier means higher priority. For our cars, only 11-bit identifiers are used. Remote transmission request consists of 1 bit; it is meant for allowing ECUs to request messages from other ECUs. The control consisting of 6 bits contains the Identifier Extension Bit (IDE) which is a 'dominant 0' for 11-bit. It contains the 4-bit Data Length Code (DLC) and specifies the length of the data bytes to be transmitted (0 to 8 bytes). The actual message being transmitted varies from 0 to 64 bits. Cyclic Redundancy Check (CRC) consisting of 16 bits ensures data integrity. The ACK indicates that the CRC process is okay and ECU has received the message correctly. The last 7 bits mark the end of the CAN message.

For a given vehicle with the manufacturer, make, and year, the definition of each CAN message signal is defined in Vehicle's CAN Database called as DBC file which is a plain text file with file extension `.dbc`. The DBC file contains a recipe on how to decode CAN messages. Once decoded, CAN messages open a new avenue for understanding human driving behavior in a wide range of traffic conditions and developing vehicle applications. Considering connected vehicles as a cyber-physical system, the vehicle data needs to adhere to three quality standards: (i) data rate, (ii) timeliness, and (iii) synchronization. In 2016, we conducted a field experiment with 22 cars to understand the phenomenon of phantom traffic jams [2]. Each vehicle was outfitted with an OBD-II style interface based on the ELM-327, an OBD-II interpreter to measure fuel consumption and velocity. Although data analysis from OBD-II data would be trivial, data obtained from OBD-II devices was of poor quality with low data rate and coarser. Hence, additional modalities – 360 camera and LiDAR were used to obtain high-frequency velocities and observe emergent traffic waves. While it was possible to extrapolate OBD-II and synchronize them with camera data, significant efforts were needed for post-processing to make inferences. Further, signals captured through OBD-II were limited to a few measurements such as speed, fuel rates, and acceleration. All of these would have been avoided with off-the-shelf technology such as direct consumption of CAN bus messages.

III. CAN Data Quality Check with Strym

We used a commercially available hardware device from Comma.ai to capture CAN messages from a vehicle on the USB port of a computer [16]. Data capture must go beyond bulk downloads in order to permit feedback control, so it is important to understand data throughput and quality if acquired in real-time.

Strym provides tools that assess the quality of the data capture, which is important for research applications in data science and machine learning. Through bulk downloads with Python drivers, CAN message recording dropped approximately 75% of messages. Figure 2 shows the data capture quality.

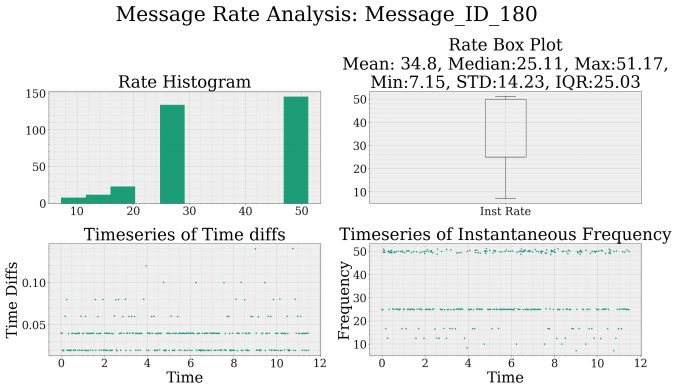

Fig. 2: Data health check done with Strym for CAN bus messages captured using Python library. From data-rate histogram (top-left) and time-diffs plot (bottom-left), we see inconsistency in arrival of messages, where 75% of data-packets were being dropped. A boxplot of data-rate shows a broader inter-quartile range of 25.03 which is non-ideal for applications such as determining dynamics of the vehicle.

As a result, we developed a high-speed data capture library called Libpanda [17] in C/C++ with low-level APIs such as libusb to boost the performance and data rate. By using Libpanda, we were able to drastically reduce CPU usage. With our efficient implementation, we captured high-quality CAN data along with GPS data at 35% CPU usage, and with twice the throughput of data collection observed with Python. These tests were performed on a Raspberry Pi 4 (4 GB RAM). A similar data-health check on CAN bus data captured via Libpanda library is shown in Figure 3. We find out that the quality of data in terms of data rate, timeliness, and synchronization was superior to that obtained using the Python library. Since data are gathered through Libpanda's real-time interface, analysis of the quality of the data are done on the static recorded files, and not in real-time. We provide further detail on Strym and how it has helped us in getting a better understanding of vehicle data to come up with strategies for vehicle application development.

IV. Programming Interface in Strym

As discussed in Section II, CAN messages have a specific structure with some messages having standard encoding and some with proprietary encoding. Libpanda records raw CAN

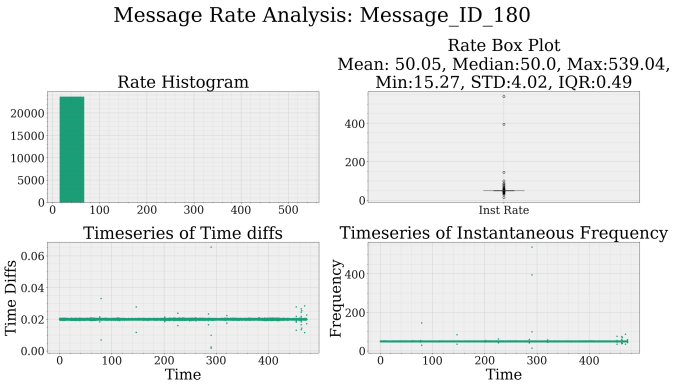

Fig. 3: Data health check done with Strym for CAN bus messages captured using Libpanda's C++ library. From data-rate histogram (top-left) and time-diffs plot (bottom-left), we see consistent data-rate. A boxplot of data-rate shows inter-quartile range of 0.49 which further supports the evidence of good-quality data.

bus data and GPS in CSV format. Raw CAN data in CSV format consists of the following columns: Time, Message-ID, Message in hex, Bus ID, and Message Length. One such example is shown in Figure 4. Each message further consists

Clock	Time	Bus	MessageID	Message	MessageLength
2020-07-08 22:15:55.778914048	1.594247e+09	0	170	1a6f1a6f1a6f1a6f	8
2020-07-08 22:15:55.779704064	1.594247e+09	0	166	0000934000000081	8
2020-07-08 22:15:55.780620032	1.594247e+09	0	562	362e302c2c26	6
2020-07-08 22:15:55.780620032	1.594247e+09	0	743	7d04000000000072	8
2020-07-08 22:15:55.780710912	1.594247e+09	0	296	0000100000000041	8
...	...	...	...	...	...
2020-07-08 22:15:56.214337024	1.594247e+09	1	407	6400000000000004	8
2020-07-08 22:15:56.215118848	1.594247e+09	1	408	6400640260fff0ba	8
2020-07-08 22:15:56.215118848	1.594247e+09	0	295	0010000800174aa9	8
2020-07-08 22:15:56.215118848	1.594247e+09	0	550	008d108b20000000	8
2020-07-08 22:15:56.215956992	1.594247e+09	1	409	64de00046002f03a	8

Fig. 4: A set of encoded CAN messages acquired through Libpanda.

of several signal components. For example, message ID 384-399 is for radar traces in Toyota RAV4 with Toyota safety package. Radar traces further consists of signal components such as longitudinal distance, lateral distance, relative velocity, and checksum for objects being tracked by onboard radar. A specific DBC rule to decode a message for Toyota RAV4 is shown in Figure 5. More information can be added in the DBC

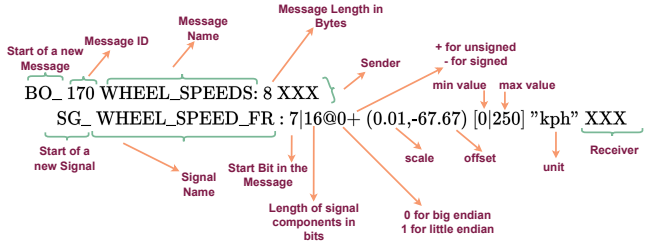

Fig. 5: A recipe in DBC file to decode a message received from CAN bus.

file such as description, enumeration types, and additional comments. Strym uses the DBC file and CSV-formatted CAN

message file as inputs to decode messages. Decoded messages are produced as timeseries data representing specific signals such as speed, acceleration, brake, lead distance, etc. A complete workflow from setting up hardware and logging CAN messages using Libpanda to decoding them as timeseries, data analysis, and visualization is shown in Figure 6.

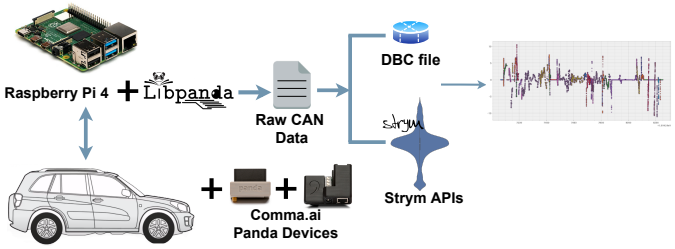

Fig. 6: A workflow of CAN data capture and analysis using Strym. CAN messages are captured using Libpanda on a Raspberry Pi connected to the Comma.ai Panda device. The captured messages are decoded after the completed drive, using Strym, for further downstream analyses.

Once messages are decoded using the DBC file, further processing and analyses are based on interpretation of the signals as timeseries data, with a *Time* column and a *Message* column. Even though the CAN bus promises a fixed data rate for messages, due to hardware and software latency, captured messages have variable data rates around the promised mean data rate. Besides, messages received from the CAN bus have neither uniform sampling time nor equal latency due to the asynchronous nature of some signals, and the runtime priorities of ECUs on the bus. As a result, any statistical analysis involving more than one signal component requires further processing of data before it can be suitably used for assessment, visualization, and further vehicle control. In the next few paragraphs, we discuss various utilities provided by Strym to empower transportation data scientists and engineers to leverage CAN technology for a number of research projects.

A. *Modules in Strym*

Strym adopts an object-oriented paradigm with abstract concepts split into multiple classes. Strym version 0.4.13 consists of following modules: (i) strymread (ii) strymmap (iii) phasespace (iv) meta (v) dashboard (vi) tools. Strymread module consists of `strymread` class that aggregates features and methods to allow reading, post-processing, and visualizing CAN data. Strymmap module provides `strymmap` class that allows reading and visualizing GPS data. Phasespace module provides tools for two-dimensional phasespace analysis of timeseries signals using `phasespace` class. In addition, Strym also provides metadata generation through `meta` and `dashboard` class. Tools modules provide procedures to house some commonly used algorithms for extending data-engineering tasks. A user can install Strym with the command `pip install strym`.

B. *Reading CAN Bus Data Using* `strymread`

Strym provides a high-level class called `strymread` to read raw CAN data. The encoded data from the data file can be read through the attribute `dataframe`. Further, if a reading of the CSV file fails for reasons such as an empty file, corrupted file, or wrong filename, etc. then the object attribute `success` can be used to check for failure/success of reading the CSV file.

`strymread` allows reading certain vehicle information in a vehicle-agnostic manner. To provide vehicle agnostic decoding, we provide a dictionary-mapping through `topic2msgs` routine. As the DBC file is not standardized, we create dictionary and naming conventions at the software level. While saving the captured CAN data from the vehicle, Libpanda follows the standard convention for data file names that include a timestamp of recording, vehicle's VIN number [18], and the modality of the data such as CAN or GPS. An example of such file name is `2020-05-19-12-56-13_JTMYF4DV1AD018936_CAN_Messages.csv` that includes a timestamp in GMT, a VIN, and the word **CAN** to inform the user that it includes CAN messages. For the legacy reason, if the data file doesn't provide a VIN then `strymread` defaults to a Toyota RAV4 encoding-decoding scheme. A user can also supply its own DBC file with an additional argument `dbcfile` while instantiating `strymread` objects. To create a dictionary, we add a new message/signal pair to a topic of interest (e.g. speed of the vehicle). For example, the Toyota RAV4 speed is found in CAN message with ID 180 and signal ID 1 but for Honda Pilot, the speed is in message ID 344 with signal name XMISSION_SPEED. The constructor of `strymread` initializes dictionary entries that can be queried at runtime to get the correct message/signal pair for the DBC file corresponding to the message file. Abstraction of the appropriate message/signal pair allows researchers to extract valid data in a vehicle agnostic manner. `topic2msgs` takes the topic name as input which returns suitable message/signal pair using dictionary entries. This redirection provides robustness to Strym as DBC files are not standardized.

As of version 0.4.13, `strymread` supports decoding CAN data for two vehicle models: Toyota RAV4 2019, Toyota RAV4 2020, and Honda Pilot 2020. We are working to add support for other models as our requirement grows. In addition to vehicle-agnostic methods, `strymread` also provides a routine to retrieve data with specific message-signal ID/name pair. Appendix A-B, and A-C provide code-snippets on the above use cases. Each vehicle message is returned as pandas timeseries data frame with two columns: *Time* and *Message*.

C. *Data Quality Assessment with* `strymread`

One big concern while working with Cyber-Physical Systems is data quality. Although we earlier motivated Libpanda as a tool to capture high-quality data, there are still reasons to assess data quality in order to understand environmental and deployment challenges when using tools such as Libpanda. `strymread` provides method `ranalyze` for data-health check in the form of rate histogram, rate boxplot, timeseries plot of time-diffs (difference of two consecutive timestamps), and timeseries plot of instantaneous frequency. Rate histogram

provides a distribution of the rate at which each data point for a particular topic such as speed was captured. In an ideal case, we expect the rate histogram to be a narrow peak with one bar. Further, for high-quality consistent timeseries data, the boxplot of data rate would be extremely narrow with an interquartile range close to zero. For such cases, time-diffs should be a straight line parallel to the x-axis. These differences can be observed in Figures 2 and 3 where high-quality consistent data was captured with Libpanda as compared to when data was captured using Python.

D. Mathematical Operations with `strymread`

To perform mathematical operations involving more than one CAN bus signal, we require additional steps. The additional steps involve resampling of CAN bus signals so that each signal consists of message values at the same timestamp. This step is necessary as multiple signals such as relative velocity, radar traces, and acceleration have different data-rate and sampling time is not uniform. Using multiple signals, we can perform a joint analysis that may lead to better decision-making and prediction models. `strymread` provides wrapper functions for resampling and synchronization of multiple timeseries data. Time synchronization of two timeseries data requires finding common timestamps for all data points. We provide `ts_sync` function for time synchronization of two timeseries data. `ts_sync` provides additional options such as desired sampling rate using `rate` argument and method used for interpolation using `method` argument. As of writing this paper, `strymread` supports cubic interpolation, and sample-and-hold interpolation with options `method='cubic'` and `method='nearest'` similar to one supported by Scipy package [19]. In an upcoming release, we are planning to add support for additional interpolation methods such as Lowess and Gaussian Kernel. In addition to timeseries synchronization, an individual timeseries can be resampled using `strymread.resample` method that uses interpolation techniques from Scipy [19] based on the desired sample-time. In addition, we also provide an alternative method of interpolation based on an autoencoder technique [20] that uses a neural network to overfit data samples on dense time-points. These steps are illustrated in Figure 7.

We further provides methods for integrating and differentiating timeseries data using `strymread.differentiate` and `strymread.integrate` functions that take a timeseries data as an argument. Additional methods include time-shift, visualization of data distribution using violin plot, and data-denoising.

V. Reading GPS Data with `strymmap`

Comma.ai Panda devices provide GPS data in addition to capturing CAN bus messages. The GPS module of Libpanda reports National Marine Electronics Association (NMEA) formatted strings which are reformatted as CSV files with Columns: GPS Time in GMT, status, longitude, latitude, altitude, horizontal dilution of precision (HDOP), vertical DOP, and position DOP. A GPS reading is valid when GPS status is

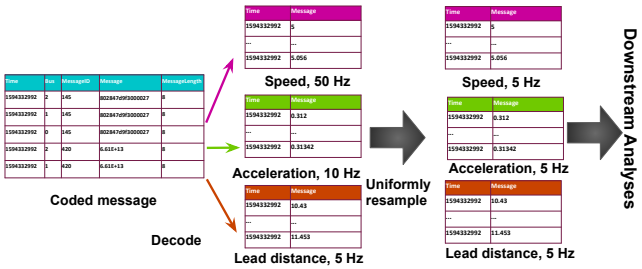

Fig. 7: A data analysis pipeline for decoding CAN messages. For joint-analysis of multiple signals, we are first required to perform time-synchronization and resample them to a fixed sample-time. Once resampling is done, we can perform downstream analyses such as denoising, smoothing, training a machine learning model, and making prediction.

set as A. We use this information to determine when a valid GPS signal is acquired. Libpanda stores GPS data in a CSV file with a naming convention similar to CAN bus message CSV file except, instead of **CAN**, the word **GPS** is used which is also useful if we want to correlate certain information obtained from CAN buses to GPS messages. For example, we can also measure speed from GPS data and compare the quality of speed with one obtained from CAN buses.

`strymmap` produces a visualization of driving routes based on GPS data using Mapbox service. Upon instantiation, `strymmap` creates a map with a driving route overlaid in png and HTML format. Since `strymmap` uses widgets, `strymmap` can only be used with an ipython notebook. An example driving route on the map is shown in Figure 8. A code-snippet to produce GPS visualization is provided in Appendix A-F.

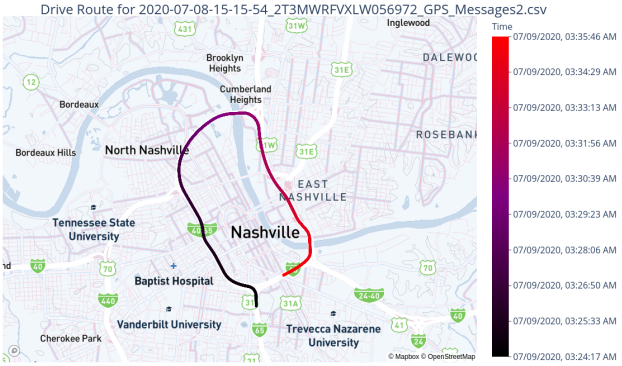

Fig. 8: An example driving route using GPS data produced with `strymmap`.

VI. Phasespace Analysis with `phasespace`

In dynamical system theory [21], phasespace analysis often reveals interesting properties of a system. A visual representation of two signals on two-dimensional phasespace provides a trajectory of the evolution of a system with respect to time or

any other dependent variable. In the two-dimensional plane, phasespace clustering provides insight into how a system may evolve from one state to another. With Strym's `phasespace` module, we can perform phasespace analysis. Currently, clustering quality assessment of phasespace is supported through `phasespace` but we are in the process of adding more methods to support several different applications related to the study of a dynamical system. An example of a phasespace plot for the velocity-acceleration curve is shown in Figure 9. A code snippet to reproduce the phasespace plot is provided in Appendix A-G.

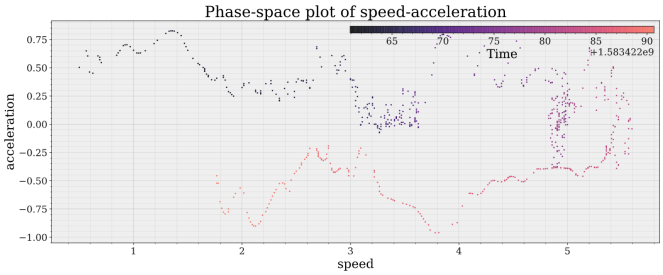

Fig. 9: An example phase space plot for velocity-acceleration curve that uses messages from CAN buses is shown with time as colormap. Speed is shown in $\frac{m}{s}$, while acceleration is shown in $\frac{m}{s^2}$.

VII. Additional tools in Strym

In addition to `strymread`, `strymmap`, and `phasespace` modules, Strym provides tools for gathering metadata about driving from a list of files, finding a subset of data with common characteristics, finding appropriate timeshift between two CAN datasets obtained from two vehicles following each other—both fitted with Panda devices, Raspberry Pi4 and Libpanda library. Further, we also provide plotting libraries based on matplotlib [22] for quick visualization of CAN messages as timeseries. Up-to-date documentation on Strym API and tutorials can be found on [23].

VIII. Use Cases

A. Relationship between Signals

We performed vehicular data analysis to gain a number of insights in identifying anomalous events, traffic flow estimation, designing controllers and choosing suitable parameters. By aggregating CAN data from multiple drives, we were able to visualize a relationship between steering angle in degree vs speed for Toyota RAV4 in $\frac{km}{h}$ shown in Figure 10. The figure indicates that usually driving maneuvers have a low steering angle at higher speed under normal driving conditions. With such an insight, an anomalous case can be detected where the driver tries to maneuver at a large steering angle at a higher speed.

B. Estimation and Validation of a Signal

Front-facing radar provides an instantaneous estimation of lead vehicle relative velocity and distance. We performed a coordinated two-vehicle test while recording data from

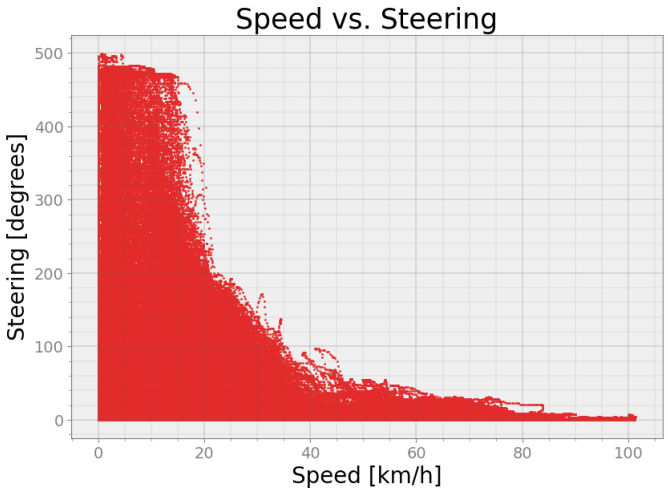

Fig. 10: Relationship between steering wheel angle, in degrees, and speed in $\frac{km}{h}$ for a Toyota RAV4 aggregated over 450 km of driving.

the lead vehicle (a Honda Pilot) and a following vehicle (Toyota Rav4). Strym provided a means to validate the relative speed measurement obtained from the rear vehicle by constructing lead speed and comparing it to the recorded speed from the lead car. To construct the leader's velocity, we first differentiated the lead distance signal to obtain relative speed where the differentiated signal was smoothed using an AutoEncoder interpolation technique [20]. A comparison of actual lead speed and reconstructed leader speed is shown in Figure 11. This use case is suitable for the case where a vehicle controller's purpose is to follow a human-driven car. Such a controller needs to have a good estimation of the leader's velocity in absence of inter-vehicle communication to minimize estimation error for safety and fulfilling other objectives.

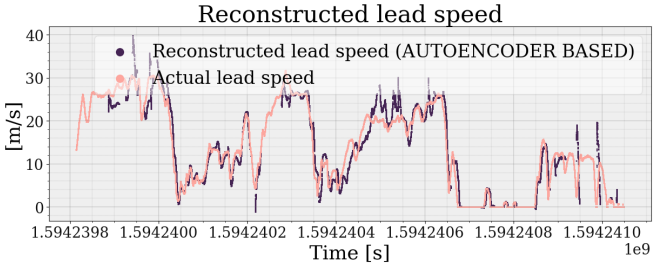

Fig. 11: Leader's velocity estimation using CAN messages from follower car using Autoencoder based interpolation provided by Strym package.

C. Obtaining Insights on Driving Behavior

In another use case, we analyzed the aggregated data from multiple drives to study the distribution of the maximum and minimum acceleration from driving data performed by a specific user. The knowledge of maximum/minimum acceleration was used for training of a reinforcement learning-based controller for vehicle control [24]. Histogram of maximum and minimum accelerations observed across all driving dataset

is shown in Figure 12 and Figure 13. We found out that maximum acceleration for a vehicle never crossed beyond $5\frac{\mathrm{m}}{\mathrm{s}^2}$ and minimum acceleration stayed below $-6\frac{\mathrm{m}}{\mathrm{s}^2}$ under normal driving conditions. These were used as bounds for training a reinforcement learning controller for longitudinal motion as described in the paper [24].

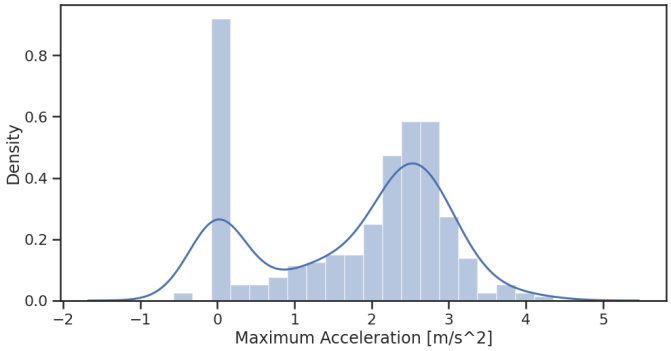

Fig. 12: Distribution of minimum acceleration found from each drive collected over the span of 10 months from the same vehicle by the same driver. Histogram is plotted with bin size of 20.

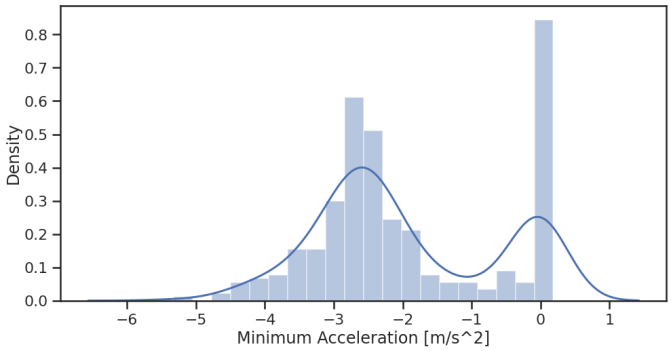

Fig. 13: Distribution of minimum acceleration found from each drive collected over the span of 10 months from the same vehicle by the same driver. Histogram is plotted with bin size of 20.

D. Multi-modal Analysis to Search for an Unknown Signal

In a different kind of use case, we sought to find out a correlated signal from the CAN bus given ground truth data. Particularly, we were interested in finding fuel usage information to train a reinforcement-learning controller [24]. The objective of such a controller is to optimize fuel usage while autonomously controlling a vehicle. To carry out the exercise of identifying fuel information, we piggybacked our CAN data collection with data collection from the Toyota Techstream device. Toyota Techstream provides fuel usage which can be used to perform correlation with a guessed message/signal ID from the CAN bus. However, the use of Techstream for obtaining fuel usage is not scalable as the device itself is costly. Further, when Techstream is in use, the vehicle doesn't allow control inputs for autonomous control.

The problem of finding fuel information was broken down into two parts: (1) finding the correct time-shift between Techstream data and CAN data using a common signal present in both modes; (2) finding a signal in CAN data that has a maximum correlation with fuel data obtained from Techstream. We were required to find time-shift because Techstream and CAN bus captured data independently of each other with their own clock. Further, the issue of calculating time-shift becomes non-trivial as Techstream needs to be restarted after a certain duration while CAN data capture doesn't get interrupted under normal operation. Hence entirety of CAN data may not have a one-to-one match with Techstream data. We identified that speed signal was present in both modes and was used for finding time-shift using `strymread`'s `time_shift` function. Identified time-shift was applied to the time-axis of CAN data and then correlation was performed with candidate message/signal from CAN data with fuel usage from Techstream. Figure 14 provides a plot demonstrating automatic alignment of speed data from two modes. The output of `time_shift` function is a time-shift in seconds that can be globally applied to one of the two modes. In the next step, we performed a correlation of Techstream's fuel data for the duration that is common in both modes with candidate message/signal ID (called Proxy). We were able to identify that message ID 865 provides data maximally correlated with Techstream's fuel data as shown in Figure 15. The use of message 865 for fuel rate was previously unknown to the open-source community.

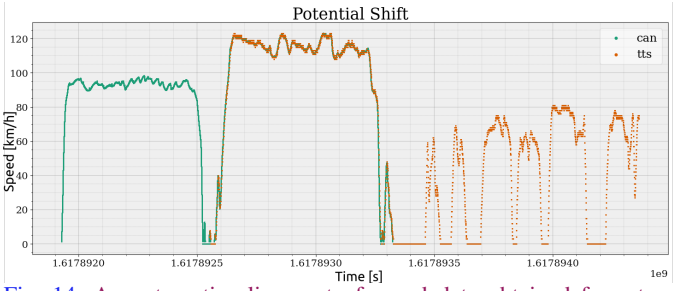

Fig. 14: An automatic alignment of speed data obtained from two modes: Techstream Device and CAN bus. Alignment can be performed even if only a portion of data matches from both modes.

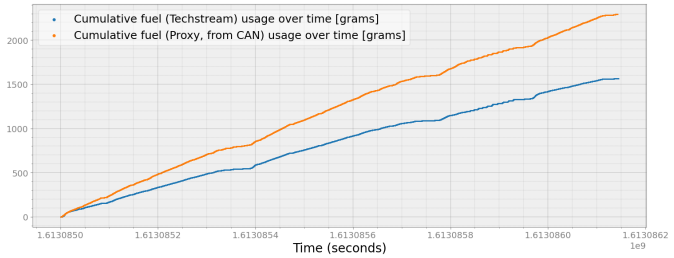

Fig. 15: Proxy CAN message 865 maximally correlate with Techstream Fuel Data.

IX. Discussions and Future Prospects

The need for Strym arose from having software tools that provide a standard way of dealing with timeseries data obtained from multiple modalities such as modern vehicle CAN buses, GPS, LiDAR, radar, cameras, etc. Currently, there

is neither a standard way of storing, analyzing, and post-processing vehicular data from multiple modalities nor there is a benchmark in terms of fusing multi-modal vehicular data. This paper is an attempt to provide an open-source software tool for analyzing multi-modal data but also an attempt in the direction of setting the standard on how to gather, store and analyze vehicular data. Currently, Strym operates only on CAN bus messages and GPS data but we are moving in the direction of combining vehicular data from other modalities such as dashcam, LiDAR, and other augmented sensors. Such multi-modal data has potential in areas such as improving autonomous vehicle control, studying human driving behavior for novel insights, and creating new traffic flow theories.

X. Source Code

Source code for Strym is available at https://github.com/jmscslgroup/strym under the MIT license.

XI. Acknowledgement

This material is based upon work supported by the U.S. Department of Energy's Office of Energy Efficiency and Renewable Energy (EERE) under the Vehicle Technologies Office award number CID DE-EE0008872. The views expressed herein do not necessarily represent the views of the U.S. Department of Energy or the United States Government. We offer additional thanks to our many collaborators in the CIRCLES project who provided thoughtful feedback on the usability of Strym for research application at scale.

References

[1] The Society of Automotive Engineers. Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles. *SAE Standards J3016*, 202104, 2021.

[2] Raphael E Stern, Shumo Cui, Maria Laura Delle Monache, Rahul Bhadani, Matt Bunting, Miles Churchill, Nathaniel Hamilton, Hannah Pohlmann, Fangyu Wu, Benedetto Piccoli, et al. Dissipation of stop-and-go waves via control of autonomous vehicles: Field experiments. *Transportation Research Part C: Emerging Technologies*, 89:205–221, 2018.

[3] Vittorio Giammarino, Simone Baldi, Paolo Frasca, and Maria Laura Delle Monache. Traffic flow on a ring with a single autonomous vehicle: An interconnected stability perspective. *IEEE Transactions on Intelligent Transportation Systems*, 22(8):4998–5008, 2020.

[4] Shumo Cui, Benjamin Seibold, Raphael Stern, and Daniel B Work. Stabilizing traffic flow via a single autonomous vehicle: Possibilities and limitations. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1336–1341. IEEE, 2017.

[5] Nianfeng Wan, Chen Zhang, and Ardalan Vahidi. Probabilistic anticipation and control in autonomous car following. *IEEE Transactions on Control Systems Technology*, 27(1):30–38, 2017.

[6] Sebastian Thrun, Mike Montemerlo, Hendrik Dahlkamp, David Stavens, Andrei Aron, James Diebel, Philip Fong, John Gale, Morgan Halpenny, Gabriel Hoffmann, et al. Stanley: The robot that won the darpa grand challenge. *Journal of field Robotics*, 23(9):661–692, 2006.

[7] Guna Seetharaman, Arun Lakhotia, and Erik Philip Blasch. Unmanned vehicles come of age: The darpa grand challenge. *Computer*, 39(12):26–29, 2006.

[8] Sebastian Thrun. Winning the darpa grand challenge. In *European Conference on Machine Learning*, pages 4–4. Springer, 2006.

[9] Rainer Makowitz and Christopher Temple. Flexray-a communication network for automotive control systems. In *2006 IEEE International Workshop on Factory Communication Systems*, pages 207–212. IEEE, 2006.

[10] Ryosuke Okuda, Yuki Kajiwara, and Kazuaki Terashima. A survey of technical trend of adas and autonomous driving. In *Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test*, pages 1–4. IEEE, 2014.

[11] Fabius Steinberger, Ronald Schroeter, and Diana Babiac. Engaged drivers–safe drivers: gathering real-time data from mobile and wearable devices for safe-driving apps. In *Automotive user interfaces*, pages 55–76. Springer, 2017.

[12] Alireza Asvadi, Luis Garrote, Cristiano Premebida, Paulo Peixoto, and Urbano J Nunes. Multimodal vehicle detection: fusing 3d-lidar and color camera data. *Pattern Recognition Letters*, 115:20–29, 2018.

[13] Uwe Kiencke, Siegfried Dais, and Martin Litschel. Automotive serial controller area network. *SAE transactions*, pages 823–828, 1986.

[14] Josef Wenger. Automotive radar-status and perspectives. In *IEEE Compound Semiconductor Integrated Circuit Symposium, 2005. CSIC'05.*, pages 4–pp. IEEE, 2005.

[15] Sam Abbott-McCune and Lisa A Shay. Techniques in hacking and simulating a modem automotive controller area network. In *2016 IEEE International Carnahan Conference on Security Technology (ICCST)*, pages 1–7. IEEE, 2016.

[16] Gaya Haciane, Rassaniya Lerdphayakkarat, Papit Meteekotchadet, Juliana Kutch, Jon Roschke, and Hakan Kutgun. Comma. ai marketing plan. 2018.

[17] Matthew Bunting, Rahul Bhadani, and Jonathan Sprinkle. Libpanda - a high performance library for vehicle datacollection. In *The Workshop on Data-Driven and Intelligent Cyber-Physical Systems (DI-CPS), CPS-Week, 2021*, 2021.

[18] Atanu Mondal and Sulata Mitra. Identification, authentication and tracking algorithm for vehicles using vin in centralized vanet. In *International Conference on Advances in Communication, Network, and Computing*, pages 115–120. Springer, 2012.

[19] Pauli Virtanen, Ralf Gommers, Travis E Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, et al. Scipy 1.0: fundamental algorithms for scientific computing in python. *Nature methods*, 17(3):261–272, 2020.

[20] Rahul Bhadani. Autoencoder for interpolation. *arXiv preprint arXiv:2101.00853*, 2021.

[21] Jürgen Moser. Dynamical systems, theory and applications. *Dynamical Systems, Theory and Applications*, 38, 1975.

[22] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95, 2007.

[23] Rahul Bhadani, Matt Bunting, Matt Nice, Ngoc Minh Tran, Safwan Elmadani, Dan Work, and Jonathan Sprinkle. Strym: A python package for real-time can data logging, analysis and visualization to work with usb-can interface.

[24] Eugene Vinitsky, Nathan Lichtlé, Matt Nice, Benjamin Seibold, Dan Work, and Alexandre Bayen. Deploying traffic smoothing cruise controllers learned from trajectory data. *International Conference on Robotics and Automation*, 2022.

APPENDIX A
EXAMPLE CODE-SNIPPETS

A. Reading CAN data

```
from strym import strymread
cancsvfile = "2020-05-19-12-56-13_JTMYF4DV1AD018936_CAN_Messages.csv"
r =strymread(csvfile=cancsvfile)
if r.success:
    print("Encoded CAN bus dataframe is:\n")
    print(r.dataframe)
```

B. Retreiving Trip Data

```
r.count() # produces the histogram of message counts per message ID
all_msg_ids = r.messageIDs() # get the list of all message IDs available in captured csv file
f = r.frequency() # retrieves frequency & relevant statistics of each message in captured csv file
st = r.start_time() # Get the start time of the capture
et = r.end_time() # Get the end time of the capture
tp = r.triptime() # total duration of the captured csv file
tl = r.triplength() # returns total distance travelled while logging CAN data.
```

C. Access Vehicle Data in Timeseries Format

Using vehicle-agnostic functions:

```
import numpy
speed = r.speed() # returns speed with SI unit as per DBC file
accelx = r.accelx() # returns acceleration in longitudinal direction
accely = r.accely() # returns acceleration in lateral direction
steer_torque = r.steer_torque() # returns steering torque
yaw_rate = r.yaw_rate() # returns yaw rate
steer_rate = r.steer_rate() # returns steering rate

# Returns longitudinal distance of track ID from 0 to 15
long_dist = r.long_dist(track_id = numpy.arange(0, 16))

# Returns lateral distance of track ID from 0 to 4
lat_dist = r.lat_dist(track_id = numpy.arange(0, 5))

# Returns relative velocity of each track with ID from 0 to 4
relative_vel = r.rel_velocity(track_id = numpy.arange(0, 5))

# Returns ACC state of the vehicle
acc_state = r.acc_state()
```

Using vehicle-specific message IDs:

```
# get_ts function can take two arguments:
## msg name or ID, and signal name or ID
## Usually it is easy to infer msg name and
## signal name from DBC file

msg869 = r.get_ts(msg=869, signal=6)
msg869 = r.get_ts(msg="DSU_CRUISE", signal=6)
msg869 = r.get_ts(msg="DSU_CRUISE", signal="LEAD_DISTSNCE")
```

D. Assessing Quality of Data

```
from strym import strymread
cancsvfile = "2020-05-19-12-56-13_JTMYF4DV1AD018936_CAN_Messages.csv"
r =strymread(csvfile=cancsvfile)
speed = r.speed()
strymread.ranalyze(speed, title ="MessageID 180", savefig=True)
```

E. Time-Synchronization of Two Signals

```
from strym import strymread
cancsvfile = "2020-05-19-12-56-13_JTMYF4DV1AD018936_CAN_Messages.csv"
r =strymread(csvfile=cancsvfile)
```

```
ts_yaw = r.yaw()
ts_speed = r.speed()
interpolated_speed, interpolated_yaw = strymread.ts_sync(ts_speed, ts_yaw, rate = 20, method='nearest')
```

F. GPS Coordinates Visualization

```
import strym
from strym import strymmap

# Setup API Key for Mapbox
# API key can be obtained from account.mapbox.com
key = os.getenv('MAP_BOX_API')
if key is None:
    api_key = input("Enter API Key: ")
    !echo "export MAP_BOX_API={api_key}" >> ~/.env

# Configure map properties
strym.config["mapheight"] = 700
strym.config["mapwidth"] = 1250
strym.config["mapzoom"] = 12.20

gpsfile = "2020-05-19-12-56-13_JTMYF4DV1AD018936_GPS_Messages.csv"

# creates a png and html file displaying driving routes
g = strymmap(csvfile=gpsfile)

# To plot the route directly in the ipython notebook
fig = g.plotroute(interactive=True)
```

G. Phase-space Plotting

```
import strym
from strym import strymread
from strym import phasespace

cancsvfile = "2020-05-19-12-56-13_JTMYF4DV1AD018936_CAN_Messages.csv"
r =strymread(csvfile=cancsvfile)

accelx = r.accelx()
speed = r.speed()
speed['Message'] = peed['Message']*0.277777778 # Convert km/h to m/s
re_speed, re_accelx = strymread.ts_sync(speed, accelx, rate=20)
ps = phasespace(dfx=re_speed, dfy=re_accelx)
ps.phaseplot(title='Phase-space plot of speed-acceleration', xlabel='speed', ylabel='acceleration')


```