OpenDAQ: Automotive Data Logging For The Masses

Richard Park and Seyedhamidreza Tavakolifard

Abstract:

We present OpenDAQ an open source data logging platform for the automobile enthusiast. Acquisition and analysis of driver and vehicle data has previously been the domain of factory sponsored racing teams. We were successful in developing an embedded system which competes with commercial products at a fraction of the cost. Research was conducted into what parameters are important in driver and vehicle development and the hardware required to collect them. OpenDAQ acquires its data from a suite of external sensors and the CAN network that exists in modern vehicles. Although it was developed for motorsports, OpenDAQ is a general purpose device which can easily be integrated into most passenger vehicles.

Introduction:

This project was motivated by a passion for motorsports, competition, and a desire to go faster. Attaching sensors to a race car in order to collect information about exactly what it is doing is not a novel idea. Formula One teams, the highest tier of automobile road racing, have been integrating data acquisition and telemetry into their vehicles since at least the late 1970's. Professional series like Formula One and Nascar are vital in developing technology like traction control and anti-lock brakes which eventually trickle their way down to passenger vehicles. Unfortunately, data acquisition systems which are easily configurable and store data have not yet crossed that barrier, but commercial turn-key devices are available for racers with deep pockets. The majority of racing that occurs around the world is a grassroots effort in which amateurs compete on everything from the same racetracks as Formula One to parking lots where the course is marked out with traffic cones. It may not be as glamorous as professional racing, but the desire to win is just as intense. The goal of our project is to offer a data acquisition system that is low cost without sacrificing usability so that amateur racers gain the full benefits of data analysis in order to set up their cars, develop as drivers, and win more races.

OpenDAQ is a data acquisition unit and not a telemetry device. Telemetry communicates data remotely for analysis in real-time whereas acquisition logs data to a storage medium for later analysis. We did not pursue telemetry as one of our goals for a two main reasons: it would add significant cost to the device in order to develop a system which can maintain a reasonable sampling rate and accurate data stream over the distances which a typical race can cover; telemetry requires a separate team member to monitor the system while the car is being driven, this does not align with our target audience of amateur racers who often act as their own pit crew.

There are several motorsport data logging applications for smartphones which do an excellent job of logging basic parameters. Their achilles heel is that they cannot easily accommodate external sensors which are not already integrated into the phone. With this in mind we decided to base our logger on the Arduino development platform which is cheap and flexible to use.

The embedded hardware consists of the Arduino which is mounted in the vehicle, GPS unit, accelerometer, tire temperature sensors, and a CAN bus transceiver which allows the logger to communicate with a modern vehicles existing computer network. There is a vast amount of data that is available over the CAN bus, but we limited our logger to a core set which were useful to our application. All data is logged to a microSD card in CSV format that can be imported into a software analysis tool.

Software tools which are designed for analyzing the data from an automotive logger exist and some are even free. We decided to develop our own analysis software as a companion to the logging hardware because of deficiencies we identified in the existing tools. All existing tools that we researched are traditional desktop software which does not support collaboration or sharing i.e. there is no social component. We hope to bring motorsport analysis software into the 21st century by developing a cloud based application that is fully integrated into the social app ecosystem.

The rest of the paper is organized into technical material which provides details into the logging hardware and software, a milestones section which covers our experiences leading up to each major progress report, and a conclusion which outlines possible future work.

Technical Material

Embedded Hardware

Microcontroller

The embedded hardware is responsible for acquisition and logging of vehicle parameters. The system is built around an Arduino Mega 2560 development board. It is powered by a 16MHz ATmega 2560 microcontroller which is produced by Atmel. This unit was selected over other Arduino form factors (e.g. Uno) due to the large number of digital and analog I/O pins and multiple serial communication ports. The Arduino proved to be a choice that we did not regret due to the familiar C like programming language, excellent standard embedded libraries, and availability of expansion modules in the form of Arduino Shields that stack on top of the development board. Since the data logger is designed to be powered by a running engine we did not focus on selecting components specifically based on their power consumption. The USB cable plugged into a standard 12v adapter provided sufficient power to supply the Arduino Mega and all other hardware modules. There was initially some uncertainty as to whether the relatively modest computing power of the Arduino would be sufficient for performing the acquisition of data and writing it to a storage medium for later analysis. Were were able to determine that it was able to maintain a consistent sampling rate of 20 Hz which was adequate for all but the shock velocity sensor that we will cover later in this document.

GPS Module

For the logging data to be useful in analysis of a driver and cars performance it is critical that the sensor data can be matched up to a location and time. The time relative to the beginning of logging is obtained from the Arduino's internal clock and the location data is provided by a Venus638FLPx GPS Receiver that is mounted to a breakout PCB by Sparkfun. In order to capture accurate and precise location data at the higher speeds which are attained in a performance driving environment, the GPS hardware requirement for our data logger was a position fix update rate of at least 10 Hz. The Venus is able to provide up to a 20 Hz update rate with the right configuration. The Venus was setup to communicate with the Arduino over a dedicated serial line. It outputs data messages, known as sentences, in a format that adheres to the NMEA GPS standard. There are over a dozen types of NMEA sentences and each one encodes a subset of parameters that are available to the GPS receiver. By default the Venus outputs 4 different types of NMEA sentences at an update rate of 1Hz. We discovered that only 1 sentence type can be enabled at update rates faster than 10Hz. We reconfigured the unit to send only RMC type sentences since it includes latitude, longitude, time, and speed data. In testing we found that the Venus was not able to maintain a consistent 20Hz update rate, but never dipped below 10 Hz. A magnetic antenna that is mounted to the roof of the vehicle connects to the receiver via an sma connector on the breakout board. We did not find cold or hot fix times to be a limiting factor. The quality of fixes, measured in the number of satellites that were connected, was consistently high during all of our testing sessions. Overall we were satisfied with it's performance given the mid range price point at which it is available.

Figure 1 is a map of Willow Springs International Raceway which is the closed circuit track that the final testing of the logger was performed at. The map is generated entirely from interpolation of gps data which was collected by the openDAQ. Figure 2 is a Google Maps satellite view of the track facility and confirms the accuracy of our positional data using GPS.

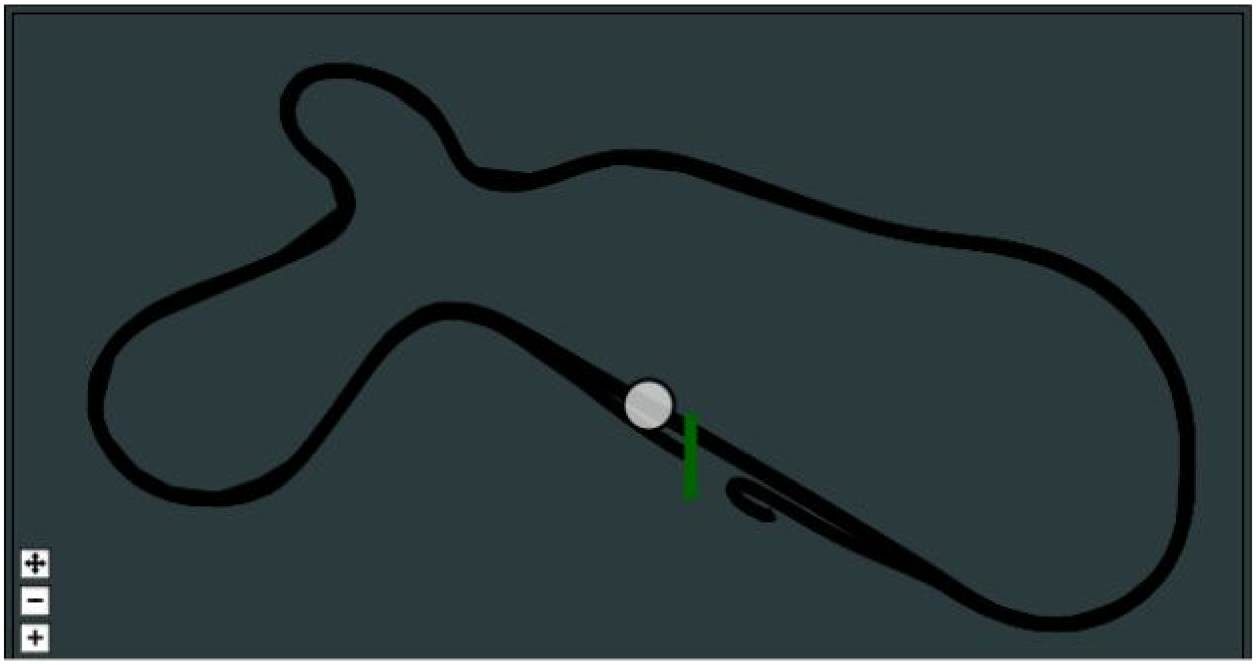


Fig. 1 Track map generated from interpolating GPS latitude and longitude

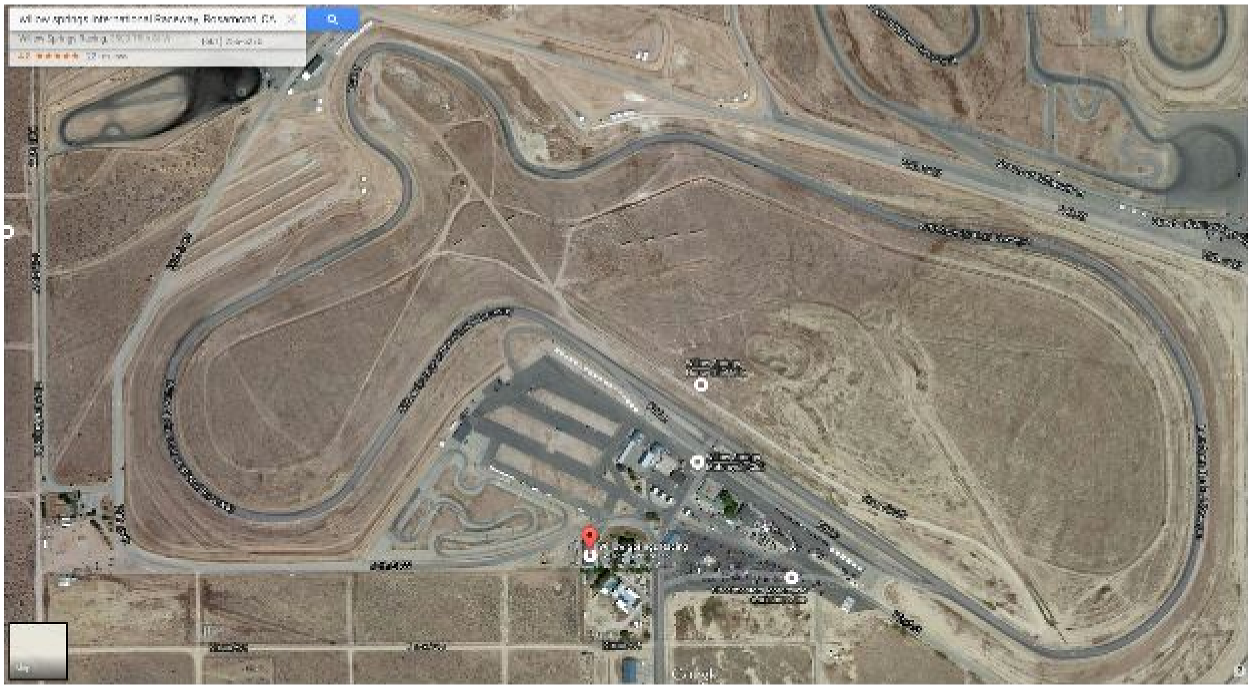


Fig. 2 Google Maps satellite view of Willow Spring International Raceway

Accelerometer

Lateral and longitudinal acceleration forces act upon a vehicle during cornering, braking, and acceleration. Logging this information in units of G-force reveals how effectively a driver is using their inputs to control the vehicle given the traction that is available. For example, if a scatter plot of g-force data shows that peak g's are not being reached then it can diagnose a driver who is not making use of all the grip available due to a lack of aggressiveness. G-forces were calculated using acceleration data provided by a 3-Axis accelerometer. We selected the ADXL345 designed and manufactured by Analog Devices. It is mounted to a breakout board by Sparkfun. The ADXL345 is a popular device due to it's low power consumption and multiple modes of operation. The dynamic range of measurement can be set to ±2g, ±4g, ±8g, and ±16g. It can be configured to communicate over the SPI or I2C protocols. A modern Formula One car is capable of 6 lateral g's. Unfortunately, our test vehicle is only capable of pulling approximately one lateral g which is typical of a modern sports car on race compound tires. With that in mind we set the accelerometer range to ±2g. Vehicles can experience much greater g-forces in extreme circumstances (e.g. collision with a solid object, airborne situations, upside down, etc), but we are mostly concerned with logging data for analysis of driver and vehicle performance under normal high performance driving conditions. We decided upon the I2C protocol for two reasons: it only requires 2-wires to communicate versus SPI which would need 4 and the IR temperature sensors that we decided to use communicate solely over I2C. We set the resolution of data for each axis to 10-bits. This provides 8-bits of resolution per g-force and is more than sufficient for our needs. The accelerometer integrated well with the rest of our system, but they are inherently noisy devices and a means of damping the output data via physical means (vibration isolation) or signal processing (low pass filter) would have provided a cleaner output signal.

Figure 3 plots longitudinal g-forces (Y axis) against lateral g-forces (X axis) for data collected over a twenty minute session of driving on a closed circuit track. The data confirms that the driver was using the available grip for the vehicle during cornering (approx 1.1g). The longitudinal data points of the y-axis are asymmetric between the positive axis (acceleration) and negative axis (braking). This is due to the much greater braking forces which can be generated as compared to acceleration. The particular vehicle that was used for testing has 200 horsepower. This is a relatively modest amount of power as compared to a fully prepared race car and explains the large disparity between braking and acceleration. The values which far exceed the g-forces for which the car is capable are data points which are produced due to the oversensitivity of the sensor (e.g. sudden jolts due to imperfections in the road surface). As mentioned before these outliers can be eliminated by filtering them or physically damping the sensor.

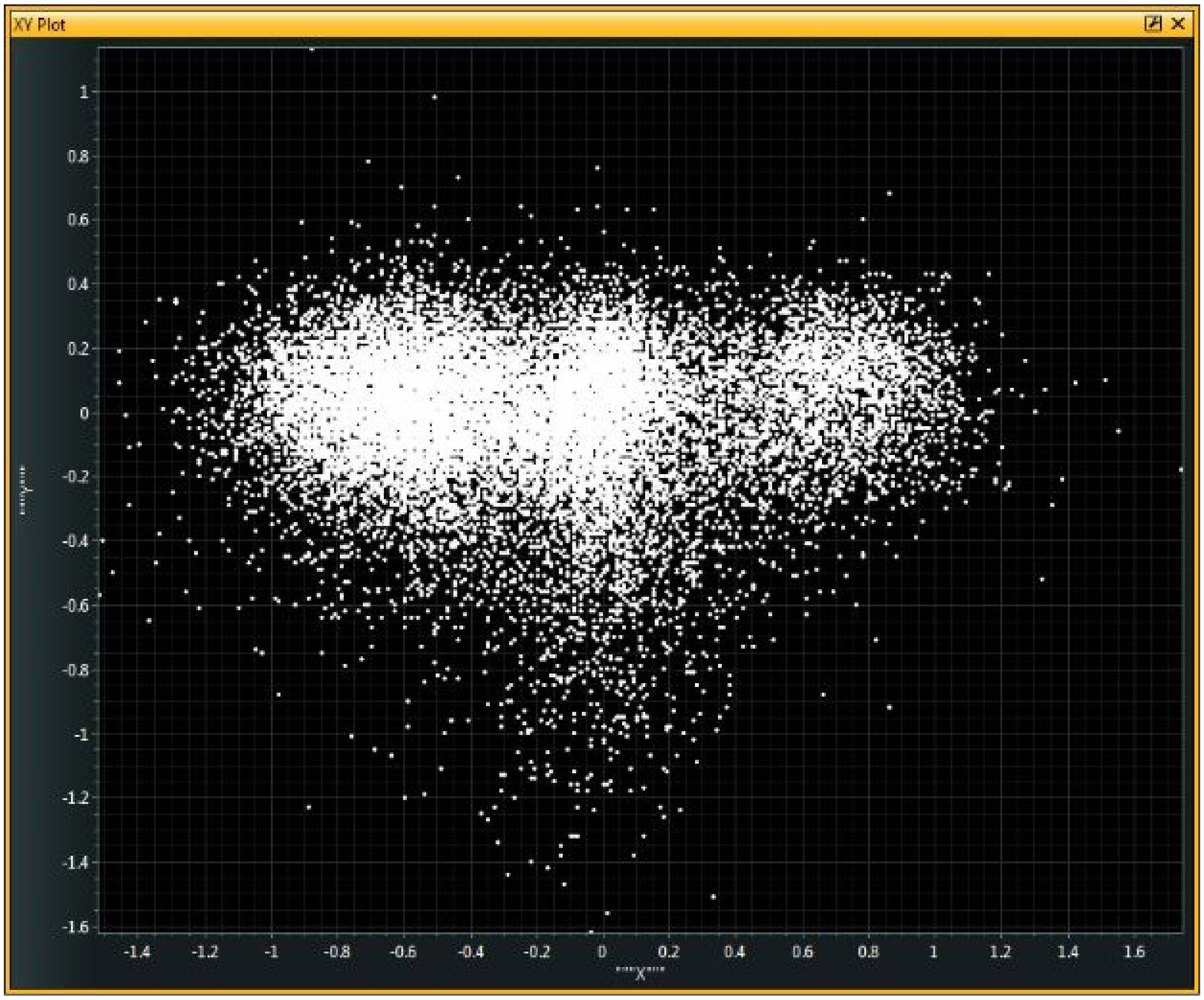


Fig. 3 X-Y scatter plot of lateral and longitudinal g-forces

CAN Bus Module

Without some form of logging the driver has to rely on their memory and sense of feel. Driving at the limit of traction requires an intense level of focus which does not spare much mental bandwidth for committing things accurately to memory. In order for a driver to evaluate their performance precisely, it is important to know exactly what kind of inputs they were giving the car at specific moments in time. By accessing the vehicles existing set of sensors through the CAN network the driver no longer has to rely on feel, but can instead perform a quantitative analysis based on fact.

Communicating with the CAN network of our test vehicle was enabled through an Arduino CAN shield which is built and sold by Sparkfun. The CAN shield integrates two chips designed and produced by Microchip: a high speed CAN transceiver chip (MCP2551) and a stand alone CAN controller chip (MCP2515). The transceiver chip converts data from the CAN controller into signals which can be transmitted or received from the physical layer of the CAN bus. The controller chip acts as a CAN bus node and translates between data communicated from the Arduino over SPI protocol and valid CAN bus frames.

OBD2 is a specification for accessing vehicle diagnostic data that is required to be implemented in modern vehicles. The connecter type and location are standardized. The OBD2 specification supports 5 different signaling protocols and CAN is among them. Due to the ease of access and availability of a common connector type the physical interface from the logger CAN bus shield to the vehicles CAN bus network is the OBD2 diagnostics port which is located under the steering wheel.

The vehicle parameters that we log are vehicle speed, engine speed, throttle position, brake position and steering angle. Since this information is of use to many of the existing CAN bus nodes they are typically broadcast over the bus on a short interval of between 10ms and 100ms. The identifiers which correspond to this information is proprietary information that differs between manufacturers and models of vehicles. In order to discover the CAN identifiers, which are 11-bits, we wrote a small Arduino program which scans all traffic and outputs only the data for identifiers which are changing values. Using this we were able to find the correct CAN id's for throttle position, brake position, and steering angle through a process of elimination (i.e. change one input at a time and see what corresponding data changed along with it). Finding the id's for engine speed and vehicle speed proved to be much more difficult. We were able to narrow the id's down to a few candidates, but the problem was compounded by the fact that we did not know the byte ordering, bit width, and those values are scaled using a proprietary formula. Due to time constraints we opted for a different approach using OBD2. The OBD2 specification provides a means for obtaining some vehicle parameters by requesting the data with a standardized identifier known as a PID and waiting for the response. We implemented this method for the two speed parameters because their PID's are standardized. A drawback of this approach is that the request and reply nature of accessing these values introduce a latency which proved to be the bottleneck of our sampling rate.

The CAN bus shield also includes a 5 function joystick and microSD card slot. The joystick was mapped to the following functions: calibrate the accelerometer so that it's resting state is centered at zero g's for each axis; disable CAN bus logging which is a useful feature for debugging external sensors without having to plug into the OBD2 port; initiate a new logging session. Logging sessions are stored on a microSD card in CSV format for later analysis.

Figure 4 plots accelerator position, brake position, and speed in the seconds leading up to Turn 3 at Willow Springs. Turn 3 requires the heaviest braking input on that particular track and is a good place for analyzing technique and vehicle setup. The graphs give show how well the driver transitions between inputs (e.g. there is minimal amount of time going between the gas pedal and brake), the driver's use of the brake pedal (e.g. achieving maximum brake pressure in a short period of time followed by releasing pressure gradually to modulate weight transfer), and metrics like entry speed into the corner and exit speed out of the corner. The simultaneous application of brake and throttle near the end of the braking event is due to a technique known as heel-toe shifting in which the gas pedal is “blipped” in order to facilitate a down shift gear change while maintaining brake pressure. This ensures that the car is settled and in the proper gear make an effective corner exit.

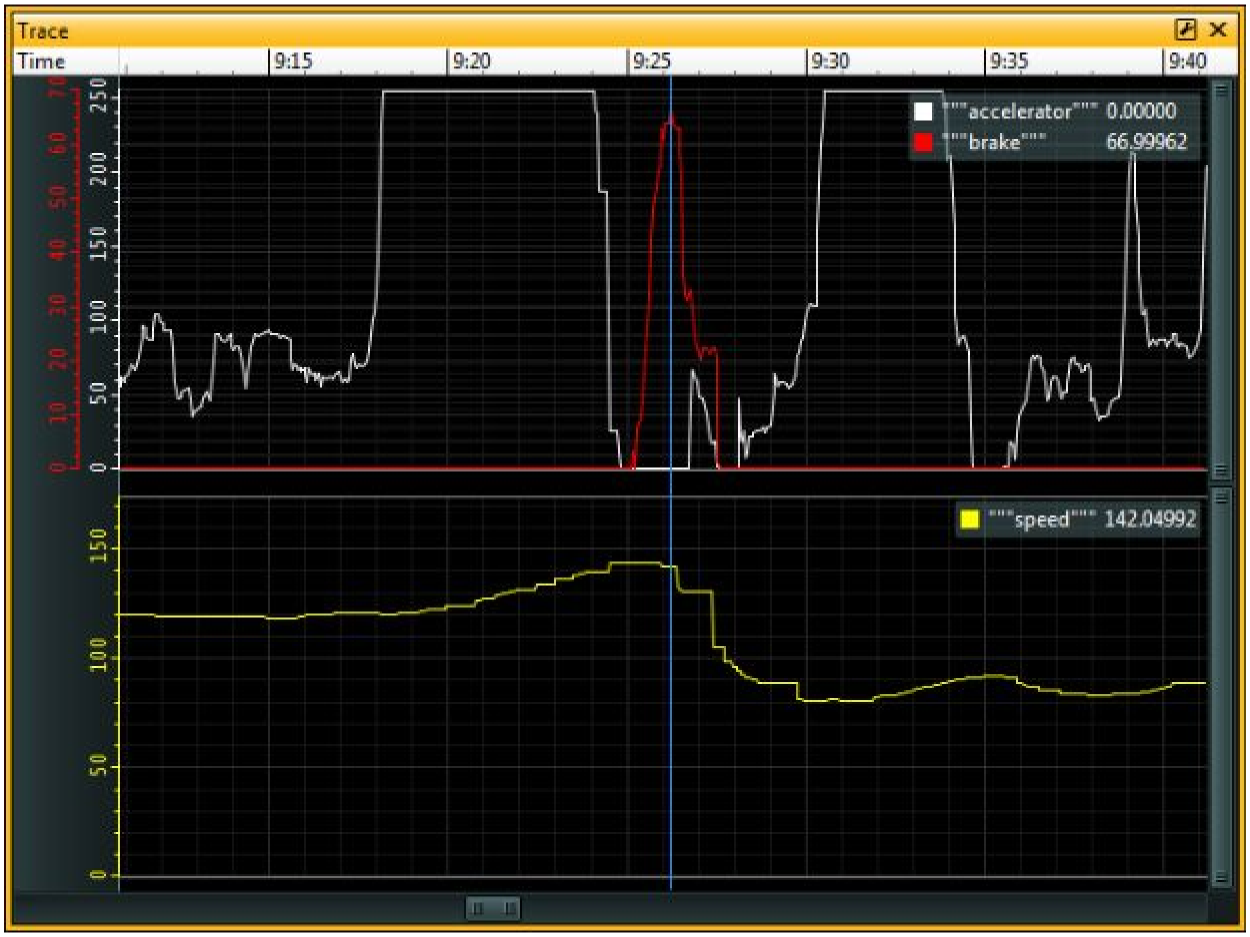


Fig. 4 Plot of accelerator position, brake position, and speed in a heavy braking zone

Temperature Sensor Array

The feedback gained from analyzing the temperature gradient across a tire can give insight into suspension setup, inflation pressures, driving performance, and tire compounds. We developed an array of three infrared temperature sensors to obtain a record of how this gradient changes over a logging session. The sensors that we chose to work with are the Melexis' MLX90614ESF-BAA. They offer a resolution of 0.02 degrees celsius in the I2C communication mode. We chose I2C over the optional pulse width modulation (PWM) mode in order to reduce the number of wires that would need to be run out to each wheel. Our test setup only instrumented one wheel (rear passenger side), but a full setup at each wheel would require 12 sensors in total. The IR sensor obtains a reading by averaging the temperatures in its field of view (FOV). Melexis produces several models of sensors with different FOV's, but we selected the 90 degree model due to its lower cost and proximity to the wheel when it is mounted. In practice we found that a 90 degree FOV provided a well balanced reading when compared against the readings from a pyrometer which is a tool that uses a sharp probe to measure temperatures at a point just below a tire's surface. The 3 sensors were soldered to a perfboard which has a 4pin connector that delivers power to the sensors and the I2C clock and data lines. This array was was mounted to the wheel liner using double sided molding tape. The mounting point it at a slight angle which is less than optimal because there is an approximately 2 inch difference in distance from the tire between the outboard sensors. Calibration of the array showed that this produced a 2.5 degree celsius differential between the two sensors at ambient temperature of about 70 degrees fahrenheit. A planned task for this array is to fabricate a mount which attaches to the hub via an adjustable arm which can be positioned precisely parallel to the surface of the tire.

Figure 5 plots two graphs. The upper graph is the temperatures for the rear passenger side tire at the outside shoulder (Rt), middle (Mt), and inside shoulder (Lt). The lower graph is a plot of the lateral g-forces experienced during cornering of the car. The negative g-forces correspond to right turns and the positive to left turns. During cornering tire temperatures go up and during straightaways they come back down. This is clearly visible in the two graphs of lateral g's and tire temperatures. The graphs also show that the outside shoulder of the tire is less affected by right hand turns due to less weight being exerted on that side as the vehicle's mass naturally rolls to the outside of the corner. This reduces the amount of mechanical friction between the tires tread and road surface which is the primary cause of tire heating. The consistently higher temperature at the middle of the tire's surface indicates that the full width of the tire is being underutilized during hard cornering and a lower inflation pressure may lead to lower lap times.

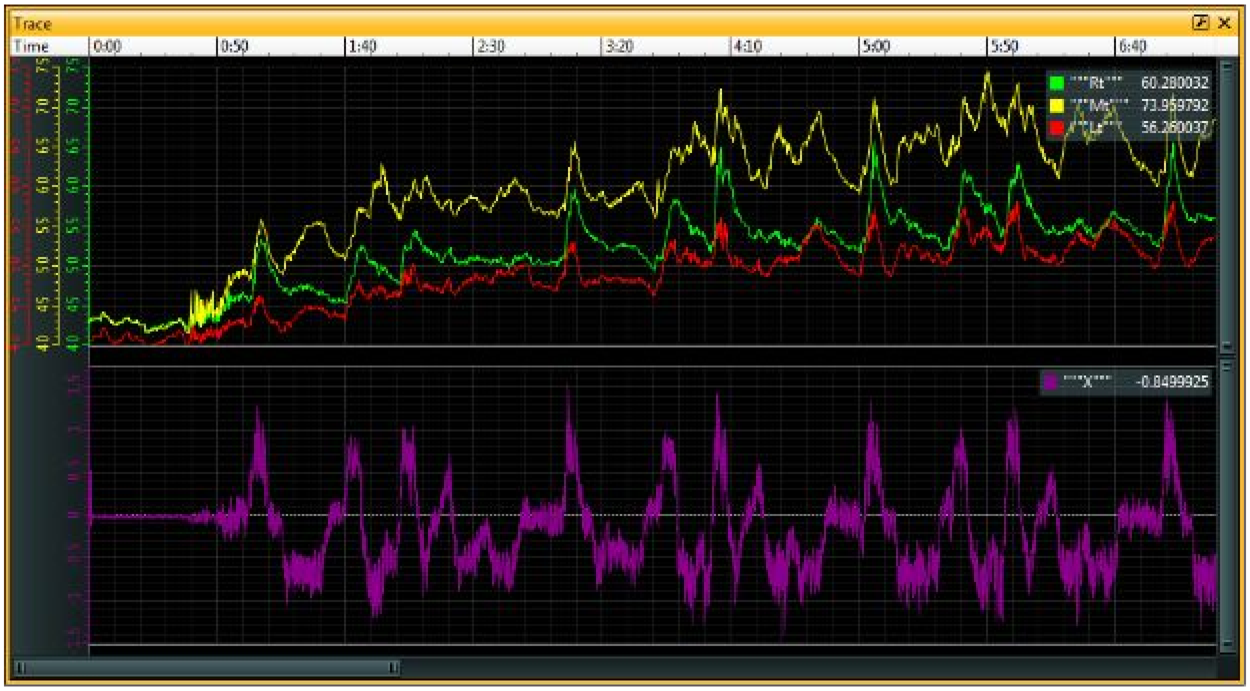


Fig. 5 Plots of tire temperature data (Rt, Mt, Lt) and lateral g-forces (X)

Software

The software will be responsible for visualizing the data that the embedded hardware recorded. The software does all the visualization that the drivers needs to understand her driving style. Also drives should be able to easily share the data with others, so it should be able to easily gets connected to internet. The software should be user friendly so the driver can easily find the menus and learn how to navigate it.

Programming language

Our programming language of choice should not be low level, because we do not have any system level tasks that should be performed in the software part. Also one of the tasks that we were expected from our program is to share the race data with others so the programming language should be able to easily get connected to cloud in order to transfer our data to cloud and that data can gets shared easily. We decided to choose python as a programming language. Python is a simple high level programming language. Python is easy to understand, so many developers can simply help extend the software functionality.

UI library

After choosing the right programming language we have to choose the library that will be responsible for UI. At the beginning we started using Tkinter, but this library was not as expandable as we are looking for. For instance it did not have an easy way to show the web content. It also did not provide a native OS look. Thus we changed our UI framework from TKinter to PyQt. Although learning PyQt involved some learning curve, at the end PyQt is more robust.

Mapping

Mapping was probably the most challenging part in the software part. Driver needs the map to understand his whole path without map he has know idea about his turns and positions. Only by mapping data to the map the driver can understand his driving style. The CSV file that we get from hardware should get converted to the map of dots. The hardware gives us the points as latitude and longitude coordinates. The easiest way is to draw these points would be to use Google maps. Simply feed the points to Google maps and it gives you the results. Thus we started to embed Google maps with our software. Given that python is relatively new language it gets updated more frequently in compare to more stable languages like c++ or java. Google maps API for embedding into our UI was written in python 2.7 while we were using python 3.0 to write our program. Version inconsistency introduces a delay in our development. We decided to manually import Google maps without its library and directly call Google maps URL for this reason.

After successfully importing Google maps into our program we feed our data to it, based on the Google documentation we feed coordinates with comma separation. Interestingly Google maps freezed! Our first impression was maybe we did something wrong in our programming part that leads to this issue but after testing the same data directly through Google maps website and also through Google earth we realize that this problems is not from our software. We did not have time to fully debug the reason for why Google maps did not work in our case, but we think one hypothesis for this failure would be because the amount of data points that we are giving him to process is a lot our first set of data point that we were giving to Google maps was 35972 point. Another hypothesis would be because we have some inconsistency between our gps unit and main Adriano we got repetitive data point with the same coordinates, so when we feed these data to Google maps we have the same coordinate repeated for couple of times.

Given that Google maps did not work we had to come up with our own custom solution. The solution should be able to draw large amount of data in relatively short period of time. Also we had to convert latitudes and longitudes to the x,y coordinates. Given that our car race will be in relatively flat area we ignored Z in our mapping.

For converting lat/long to x,y we used the following formula:

R = 6371000 *Earth radius in meters*

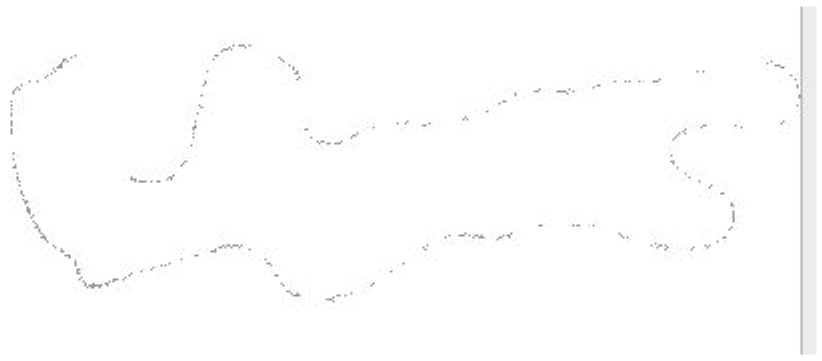
*X = R*cos(lat)*

*y = R*cos(long)*

The above formula gave us good precision in drawing our map.

After conversion it was time to actually draw the map using python and PyQt drawing tools. At the first we used PyQt, QGraphicsPathItem class. This class was able to draw the intended map for us. The problem was this class was so slow and it actually draws the path point by point, so it took about 5 minutes each time for the map to get drawn. We had to choose a better library in order to first get all the path info and then draw it.

We use the pyqtgraph and specifically graph item class for this purpose. By giving this class the same data point it was be able to draw the map a lot faster and smother. Below is the final map:



Graphing other data

After our maps get drawn, we have to graph related data. For this purpose we used the same pyqtgraph library. Given the time pressure we did not implement the native user interface for sensors which means that if we had a speed parameter for our date we show it as regular line chart based on the time for now. This part like the last past should process the data quickly in relatively short period of time.

Connecting everything together

Now we have everything in place, the last step for us is to connect all the charts together. Which means when a user hover on one chart all the related data on all the other charts should gets highlighted as well.

The first and most important part is to have an id for each data point. As mentioned above we have repetitive coordinate in our data so we cannot use coordinates as our ID. In order to select the id we decide to choose coordinate time as our id. Since we draw our map and convert our coordinates to the x,y plane we are still looking into the way to connect to our map.

We are able to connect all of our charts together, but our algorithm to detect point in our map is still under development.

Milestones

Update 1:

We were able to achieve all the planned goals for the embedded hardware and software. At the time of this update we had a functioning logger that had been tested in the field during an Autocross practice session. Autocross is a form of low speed motorsport that takes place on tight technical tracks which are marked by traffic cones. The process of reverse engineering CAN bus ID's was very tedious and we were not as successful in decoding the identifier for vehicle and engine speed. As described in the previous CAN module technical section, we were able to determine a few candidate identifiers. A major hurdle was converting the raw can bus values into an engine speed in revolutions per minute and vehicle speed into kilometers per hour or miles per hour without knowledge of the conversion formula that is used by the vehicle manufacturer.

we were be able to import our data into the software and also implemented google maps into the software, but we did not get to fully test it. We saw some inconsistency with google maps. Map can load sample data points easily, but when we feed the whole data set into it will freeze.

Final Update:

We were able to implement and test a functioning tire temperature array at the end of this milestone. Since we used I2C to communicate with the three ir sensors, they each needed a unique address. This proved to be a bit of a challenge because Melexis chose not to adhere strictly to the I2C standard. The standard Arduino library for I2C worked fine for reading from the devices, but not for writing to them. After some searching, we were able to find a third party library online that communicates with the devices properly and were able to set the I2C address of each sensor to a unique value by writing a simple program.

We were not successful in implementing a working position sensor for our logger. We tried several iterations that were based around a linear slide potentiometer. Integrating a sensor into the Arduino which detects the position based on voltage from the slide pot was not an issue. The difficulty was in housing the potentiometer so that it was protected from the elements and modifying the potentiometer with a system of articulating rods that would enable it to be mounted to fixed points on the suspension. Initial tests were conducted using zip ties to fix the rod ends to coils on the rear spring. That test failed due to the difference in travel between the slide potentiometer and rear shock absorber. We plan to pursue this until we are able to come up with a viable solution. The next version will reduce the amount of articulation in the sensor to eliminate problems with binding. We will also try mounting the sensor to suspension points located on the control arms which are located inboard of the shock absorber and experience less travel. Measuring the geometry of the suspension components will allow us to convert the readings take at the control arm to travel at the shock absorber. The data that we seek to analyze with the shock position sensor is the velocity at each sample point. Our current logger runs at approximately 20 Hz which is sufficient for the current set of parameters. In order to produce a velocity reading at each sample point we will require that the shock position is updated at a rate of at least 40 Hz. This is currently not possible with the Arduino running our embedded logging software. We will explore three possible options in the future: switch to a microcontroller with a higher clock speed, use a dedicated Arduino microcontroller for the shock sensors, remove the current software bottleneck by reverse engineering the CAN id's for engine speed and vehicle speed.

The software portion can successfully visualize the data and show the map. We are still looking in a way to integrate our data point with the rest of our data. As mentioned in the software portion we will most probably have to use the combination of latitude and longitude and times as each points uniq id. After we solve this problem we will move forward by making the UI more user friendly and implement the actual gauges for graphs.

Conclusion

We achieved a majority of our project goals this quarter. The OpenDAQ logger is a fully functional device which provides accurate data that can be used to improve a vehicle and driver's performance. Looking forward we would like to continue the development of the software analysis tool so that it is a cloud based application with all the features of competing desktop tools. The bill of materials for building the current version of OpenDAQ is well under that of a commercial product, but the barrier to wide adoption of the logger is that it requires some technical experience in programming and assembling circuits. In order to create a closer to turn-key type of system we plan to layout custom PCB's for the circuits, refactor the source code, and fabricate mounts for external sensors. Through the experience of developing this project we gained experience in motorsports specific data analysis and the process of creating a product that others would want to use. The feedback we received during presentations was invaluable and we hope to apply the skills we learned in class to our future endeavors.