Software de Análise de Dados Obtidos a partir de um Dispositivo OBD-II

Lucas de Camargo Souza, Edgard Haenisch Porto e Ester Cavalheiro
Universidade Federal de Santa Catarina (UFSC)

***Resumo*—Este trabalho busca mostrar ao estudante de engenharia uma forma alternativa de obter os dados da rede de um veículo utilizando um dispositivo OBD-II e criar um software de análise para estes dados. Os conhecimentos necessários para a confecção deste trabalho são: programação orientada a objetos em C++, Python e noções básicas de desenvolvimento de aplicações na IDE Qt Creator.**

***Palavras-chave*—Redes Veiculares, OBD-II, Análise de Dados, Engenharia Automotiva.**

I. Introdução

A indústria automotiva utiliza sistemas eletrônicos para gerenciar os componentes de um veículo, em particular, redes de comunicação, que realizam a interconexão das Unidades de Controle Eletrônico (*Electronic Control Unit - ECU*) para transmitir mensagens entre as tarefas distribuídas. Para fins de diagnóstico, muitos dos dados transmitidos pela rede automotiva são disponibilizados em um conector padronizado que gera saídas de leitura. Este sistema de autodiagnóstico é denominado de OBD (*On-Board Diagnostic*) e está presente em todos os veículos brasileiros que foram produzidos a partir de 2010. Para ler estes dados, é necessário obter um dispositivo de leitura, como o ELM327, utilizado neste trabalho – com conexão Bluetooth. Ainda, precisa-se utilizar um software capaz de se comunicar com o dispositivo, obter os dados e mostrá-los ao usuário de forma agradável. Entretanto, muitos destes softwares são pagos ou não atendem as expectativas do usuário. Porém, felizmente, encontra-se disponível na internet bibliotecas *open source* de uso livre, que podem ser utilizadas na confecção de um software como citado anteriormente. Desta forma, o objetivo deste trabalho foi desenvolver um software de análise de dados obtidos a partir do dispositivo de leitura OBD-II, utilizando a IDE Qt Creator. Há também o propósito de mostrar ao estudante de engenharia automotiva uma forma alternativa de obter e interpretar os dados que circulam na rede de um carro. Por fim, os testes foram realizados em um veículo Chevrolet Onix Effect em diferentes tipos de rotas.

Todo o trabalho produzido está disponível abertamente como repositório em [1] e é de uso livre.

II. Metodologia

Nesta seção, será investigado o funcionamento básico do dispositivo OBD-II, os métodos de comunicação e obtenção de dados deste equipamento utilizando a linguagem Python, a passagem dos dados para o software Qt Creator e como gerar gráficos a partir das informações obtidas. Por fim, tudo será organizado dentro de um software.
Os requisitos para este trabalho são:

- Computador com sistema operacional Linux
- Dispositivo de leitura OBD-II ELM327 (USB ou Bluetooth, fig. 1)
- Python 2.7.14
- IDE Qt Creator (Qt 5.11)[1].

Figura 1. Modelo do dispositivo adaptador de OBD-II ELM327 usado no trabalho.

É necessário atentar-se ao tipo de comunicação do dispositivo de leitura ELM327, caso seja Bluetooth, como o utilizado neste trabalho, deve-se garantir o módulo de comunicação Bluetooth no computador e seguir os passos de configuração adicionais que serão listados abaixo.

A. Configuração Inicial

Para estabelecer uma conexão com o dispositivo ELM327 e fazer a coleta de dados, utilizou-se a biblioteca de licença livre Python-OBD, documentada e disponível em [2]. A instalação da biblioteca é feita através do *pip*:

```
$ pip install obd
```

Caso o adaptador OBD-II utilize comunicação via Bluetooth, recomenda-se a instalação dos seguintes pacotes:

```
$ sudo apt-get install bluetooth
bluez-utils blueman
```

B. Conectando com o ELM327

Antes de dar início ao desenvolvimento do software, fez-se alguns testes com o dispositivo ELM327 utilizando a biblioteca citada e a linguagem de programação Python (versão 2.7). Os programas de testes foram feitos com base nos exemplos já disponibilizados por [2]. O uso da biblioteca exige que o adaptador OBD-II seja conectado com

[1] https://www.qt.io/qt5-11

o computador e disponibilizado por um driver de dispositivo através de uma porta de comunicação. Caso o adaptador seja USB, a porta será do tipo *tty* e, caso seja Bluetooth, deverá ser do tipo *RFCOMM*. O último caso exigiu os seguintes passos de configuração, uma vez que o adaptador esteja ligado e os pacotes de comunicação Bluetooth devidamente instalados:

```
$ hcitool scan
$ sudo rfcomm bind 0 [endereço do
adaptador OBDII]
```

O primeiro comando irá gerar uma lista com os dispositivos Bluetooth encontrados e seus endereços – como "*AA:BB:CC:11:22:33 - OBDII*". Este endereço deverá ser anexado ao segundo comando, como descrito. Este comando irá gerar um driver para o dispositivo ELM327 no diretório */dev* e, caso não seja possível utilizar o número 0, pode-se tentar com outros (1, 2, 3, ...). Após conectado, o comando abaixo deverá mostrar todos os dispositivos conectados ao computador via porta *RFCOMM*:

```
$ ls /dev | grep rfcomm
```

C. Programa Teste

Uma vez que a conexão foi estabelecida com o adaptador OBD-II, pode-se fazer a primeira comunicação com o dispositivo. Para isso, utilizou-se o programa em Python abaixo:

```
1 import obd
2
3 # conecta automatic. a uma porta USB ou RF
4 connection = obd.OBD()
5
6 # mostra a porta utilizada pela conexao
7 print connection.port_name()
8
9 # mostra o status da conexao
10 print connection.status()
11
12 # mostra o protocolo utilizado
13 # pela conexao com o veiculo
14 print connection.protocol_name()
```

A linha 4 instancia um objeto do tipo *OBD*, que procura os dispositivos disponíveis nas portas USB ou RFCOMM e, caso encontre o adaptador ELM327, conecta automaticamente. Os métodos de conexão desta biblioteca estão melhor documentados na seção "*OBD Connections*" de [2], junto com os métodos disponíveis pela classe *OBD*. Para que o teste seja bem sucedido, é necessário que o adaptador OBD-II esteja conectado à saída de diagnóstico do veículo, que geralmente encontra-se em baixo do volante, ao lado esquerdo do motorista. Ainda, recomenda-se que o veículo esteja com a chave ligada, para acionar o sistema eletrônico – o motor não precisa estar necessariamente ligado.

D. Obtendo Dados

Uma vez que a execução do programa de teste foi bem sucedida, pode-se começar a obter dados do veículo. Para isso, deve-se primeiro entender como um comando de solicitação de dados é enviado ao veículo. A documentação [2] possui uma seção "*Command Tables*" com todos os comandos que podem ser interpretados pela porta de diagnóstico OBD-II, tais comandos são padronizados nesta porta para todos os tipos de comunicação veicular. Na página de documentação, os comandos são apresentados em uma tabela contendo os campos *PID, Name, Description* e *Response Value*. Segundo [3], o campo *PID* significa os identificadores de parâmetros usados para solicitar dados de um veículo e são padronizados pela norma J1979 da SAE (*Society of Automotive Engineers*). O campo *Name* representa o nome daquele comando utilizado na biblioteca e *Descrption* contém uma breve descrição do comando. O campo *Response Value* é de grande importância, pois informa como o dado solicitado será retornado ao usuário programador, sendo que alguns tipos de respostas são características da biblioteca e estão melhor descritos na seção *Responses*. Ainda, as respostas de valores numéricos, que serão as mais relevantes para a análise gráfica de dados, podem ser convertidas em valor flutuante através do método de classe *value*. Para obter um dado do veículo, basta fazer a chamada abaixo, que envia o comando solicitado ao veículo e retorna o valor lido do barramento de diagnósticos:

```
resposta = connection.query(<NAME>)
```

Em que *<NAME>* especifica o nome do comando solicitado, como descrito pela tabela citada anteriormente.

Após analisar a tabela de comandos disponíveis, é importante saber quais deles são aceitos pelo veículo utilizado, para isso existem três comandos que retornam esta informação: *PIDS_A, PIDS_B* e *PIDS_C*. Estes comandos retornam um vetor de números binários com o valor 1 representando que o comando naquela posição, partindo do PID 01, é aceito pelo veículo. Por exemplo, caso a o comando *PID_A* retorne o valor 10101..., significa que os comandos de PID 01, 03 e 05 são aceitos pelo veículo e os comandos de PID 02 e 04 não são aceitos.

Conhecendo o funcionamento dos comandos citados logo acima, foi feito um programa em Python que identifica todos os comandos aceitos pelo veículo e os imprime na tela em forma de lista, com seus respectivos PID, nome e descrição. Este programa esta disponível em [1], dentro do diretório *OBD Comandos*.

Por fim, a seção *Async Connections* da documentação [2] merece uma devida atenção, pois mostra como é feita uma comunicação assíncrona com o dispositivo ELM327 para obtenção de dados. Isto possibilita obter vários dados do adaptador de uma forma mais eficiente do que a citada anteriormente. Tal forma resume-se em dizer ao programa quais os dados que se deseja obter do veículo e especificar qual método deve ser chamado quando estes dados estiverem disponíveis. Por exemplo, ao solicitar o dado da velocidade do veículo, o adaptador irá colocar está informação na rede de diagnóstico e esperar por uma resposta para enfim retorná-la ao programador. O método assíncrono faz com que enquanto ele espera por uma resposta, pode executar outros métodos até que a resposta chegue, fazendo com que haja uma interrupção no fluxo de execução e a resposta seja finalmente retornada ao usuário. Este método foi crucial para o desenvolvimento deste trabalho, pois deseja-se obter continuamente uma quantidade pré-estabelecida de diferentes dados. O código abaixo exemplifica uma obtenção periódica dos dados de velocidade e rotação (RPM) do veículo:

```python
import obd
import time

# estabelece uma conexao assinc.
connection = obd.Async()

# func que sera chamada qdo o dado
# estiver disponivel
def new_rpm(v):
  print v.value

def new_speed(v):
  print v.value

# registra os comandos solicitados
connection.watch(obd.commands.RPM,
  callback=new_rpm)
connection.watch(obd.commands.SPEED,
  callback=new_speed)

# inicia a obtencao de dados
connection.start()

# recebe dados por 60seg
time.sleep(60)

# encerra a conexao
connection.stop()
```

A execução do programa acima fará com que os dados sejam obtidos e mostrados na tela por 1 minuto.
Entretanto, o objetivo do trabalho não é criar uma aplicação em Python que retorna ao usuário dados pelo terminal do Linux, mas sim criar uma aplicação com interface utilizando o Qt Creator. Para isso, deve-se aplicar uma forma de passar os dados obtidos pelo programa em Python para uma aplicação em Qt, feita utilizando a linguagem C++, e de forma rápida. A próxima seção irá abordar uma forma de que isso pode ser realizado.

E. D-Bus

Deve-se ter em mente que um programa em Python e um programa feito em C++ utilizando o Qt Creator caracterizam dois processos diferentes, logo cada um deles possui o seu próprio espaço de endereçamento e as informações de um não está diretamente disponível para o outro. Desta forma, a passagem de dados entre processos define um tópico em computação chamado de IPC (*Inter-Process Communication*), que são mecanismos utilizados pelo sistema operacional para operar com dados compartilhados os entre processos. Aqui não será discutido o funcionamento de um IPC e nem as diferentes formas de realizar esta operação, pois tal discussão possui uma certa complexidade computacional. Logo, a abordagem será direta: optou-se por utilizar um recurso disponível em sistemas operacionais Linux chamado de D-Bus. Como dito em [4], este recurso define uma forma simples de comunicar diferentes aplicações no sistema e é caracterizado por ser uma abstração de um barramento de comunicação, porém que conecta processos.
O D-Bus é também referenciado pela própria documentação do Qt, disponível em [5]. Ainda, o Qt disponibiliza uma forma simplificada de operar com este recurso e ferramentas para facilitar a implementação. O funcionamento deste barramento resume-se na troca de mensagens entre processos que estão conectados à rede por meio de um nome de serviço, que consiste em um endereço de como pode ser referenciado, fazendo uma analogia aos endereços de IP e *hostnames*.
Por exemplo, o serviço D-Bus é definido por *freedesktop.org* e pode ser encontrado pelo nome de serviço:
`org.freedesktop.DBus`
Utilizando este recurso, é possível instanciar um objeto no barramento e fazer com que outros processos usem de seus recursos. No Qt, as classes que disponibilizam recursos no barramento D-Bus são chamadas de adaptadores. Esta classes tendem a ser leves tal que sua principal função seja retransmitir de e para um outro objeto, validando ou convertendo as entradas vindas do barramento.
O D-Bus também pode ser facilmente utilizado pela linguagem Python, dentre as bibliotecas disponíveis para operar com este recurso, optou-se pela PyDbus, disponível em [6] e que pode ser instalada através do comando:

```
$ pip install
  git+https://github.com/LEW21/pydbus.git
```

F. Classe OBData

O primeiro passo para a implementação do software é desenvolver uma classe principal que irá abrigar e disponibilizar os dados obtidos da OBD-II, através da aplicação em Python, para o programa em Qt. A estrutura desta classe deve ser muito definida, ela consiste em métodos *getters* e *setters* para todos os dados solicitados do veículo. Por exemplo, uma vez que se deseja obter e analisar o dado de velocidade do veículo, esta classe deverá conter as seguintes declarações para a variável *Speed*:

```
class OBData : public QObject
{
public:
  double getSpeed() const;

signals:
  void Speed_changed(double);

public slots:
  void setSpeed(double val);

private:
  double Speed;
};
```

Desta forma, uma vez que a aplicação em Python receber um valor para a velocidade do veículo, esta deverá fazer a chamada do método *OBData.setSpeed()*, alterando o valor da velocidade na classe principal, de modo que a alteração do valor irá gerar o sinal *Speed_changed()*, notificando todos os métodos conectados a este sinal que a velocidade foi alterada, passando o novo valor como argumento. Este simples método pode ser estendido para quantos dados forem necessários, basta replicar o padrão para a variável *Speed*. Como exemplo, a versão final da classe *OBData* está disponível no apêndice B.
Para possibilitar o uso do D-Bus, tomou-se como exemplo a referência [7], em que o autor implementa uma aplicação do barramento que consiste em uma luz e um controle, tal

que ambos são processos individuais, mas que se comunicam entre si, tornando possível apagar e acender a luz através do processo do controle. Assim, a classe *OBData* assemelha-se com a classe que representa a luz, já a aplicação em Python representa o controle, pois deve enviar dados.
O primeiro passo para configurar o D-Bus é gerar um arquivo XML que consiste das informações da classe OBData que serão agregadas ao barramento. Para isso, executa-se a chamada
`$ qdbuscpp2xml obdata.h > obdata.xml`
que irá gerar uma saída contendo um código em XML. Este código deve ser agregado ao arquivo *.pro* da aplicação em Qt da seguinte forma:

```
# OBData.pro
QT       += dbus
DBUS_ADAPTORS += obdata.xml
```

O segundo passo da configuração consiste em criar uma instância da classe *OBData* no barramento D-Bus, que foi feito durante a inicialização do programa, ao longo da execução do construtor da classe *MainWindow*, que representa toda a interface gráfica da aplicação. Isso foi feito da forma abaixo:

```
#include "obdata_adaptor.h"

MainWindow::MainWindow(QWidget *parent) :
  QMainWindow(parent)
{
 // ...
 // Cria um interface adaptor
 new OBDataAdaptor(&obdata);

 // Conecta ao session bus
 QDBusConnection connection =
     QDBusConnection::sessionBus();
 if (!connection.isConnected()) {
   qWarning("DBus Error.");
 }
 connection.registerObject("/OBData", &obdata);
 connection.registerService("local.OBData");
}
```

Nota-se que a classe foi registrada no barramento utilizando o nome de serviço *local.OBData*, que foi retirado do arquivo *.xml* que contém a linha especificando este endereço
`<interface name="local.OBData»`
Ainda, a instância desta classe está registrada como */OBData*.
Após configurar o D-Bus para a classe *OBData*, deve-se implementar o programa de coleta de dados em Python. A abordagem para conectar-se com a classe implementada no barramento é direta e necessita-se apenas de poucas linhas:

```
# obdpy.py
from pydbus import SessionBus

bus = SessionBus()
obdata = bus.get("local.OBData", "/OBData")
```

A linha 2 importa a classe necessária para conectar-se com o barramento e a linha 4 cria um objeto para esta classe. Agora é apenas necessário localizar a classe *OBData* e sua instância no D-Bus utilizando o seu nome de serviço *local.OBData* e o endereço do objeto */OBData*. Desta forma, a aplicação em Python está pronta para se comunicar com o programa em Qt. Logo, resta enviar os dados coletados e para isso basta uma modificação no código já apresentado em II-D, pois ao invés de mostrar os dados na tela, pode-se passá-los como argumento para os métodos da classe *OBdata*. O código completo da aplicação em Python foi agregado a este relatório e está disponível no apêndice A. Por fim, a execução do programa em Qt seguida da aplicação em Python irá criar uma comunicação entre os dois processos, de forma que os dados da OBD-II serão repassados para o Qt. Logo, decidiu-se executar o programa em Python durante a execução do programa em Qt utilizando a classe *QProcess*. Este método é executado na aplicação final ao clicar-se em *Conectar*, na aba *Configuração* – o programa está disponível em [1].

G. Gerando Gráficos em Tempo Real

Para visualizar os dados obtidos da OBD-II, optou-se por criar uma aplicação capaz de gerar gráficos interativos em tempo real. O método para desenvolver esta ideia utilizando apenas os recursos fornecidos pelo Qt tornaria a confecção do trabalho muito mais complicada e estaria fora do escopo do tema principal abordado aqui. Felizmente, existe uma biblioteca de código livre para gerenciar gráficos de análise de dados, chamada *QCustomPlot*, disponível e documentada em [8]. Desta forma, tomou-se como base de desenvolvimento o exemplo também disponível na documentação da biblioteca, referenciado em [9]. Este exemplo mostra a utilização da classe *QCustomPlot* para o desenvolvimento de um gráfico que se atualiza e recebe dados em tempo real. A ideia é utilizar um *timer* que invoca um método responsável atualizar o gráfico periodicamente. Antes de prosseguir com o desenvolvimento, é necessário ler e entender os passos para a configuração da classe *QCustomPlot*, disponível em [10]. Nota-se na página de configuração que o arquivo *.pro* do programa sofrerá uma pequena alteração e que os gráficos são tratados como *Widgets* dentro da aplicação.
Tendo em vista o funcionamento básico da classe citada acima, para facilitar o desenvolvimento, foi implementada uma classe chamada *RTPlot*, que herda da classe *QCustomPlot*. Tal implementação teve como objetivo disponibilizar para o programa principal uma interface simplificada de chamadas de métodos para a plotagem dos gráficos em tempo real, de forma que as informações passadas para estes gráficos serão aquelas repassadas pela aplicação em Python já discutida acima. Os detalhes da implementação da classe *RTPlot*, assim como a da interface gráfica, não serão discutidos aqui, pois tornariam este trabalho muito extenso. Entretanto, suas funcionalidades básicas serão apresentadas.
A classe *RTPlot* gera uma plotagem de dados que pode ser inicializada, pausada e reiniciada. Estes dados são plotados a medida que são recebidos da aplicação em Python pelo programa em Qt, para isto, é necessário haver uma conexão entre *signals* e *slots*, que são conceitos básicos de desenvolvimento de software em Qt. Resumidamente, quando um dado é recebido pela classe *OBData*, ela dispara um sinal, que é reconhecido pela classe principal *MainWindow*, tal que os dados são repassados e plotados para a classe *RTPlot*. Como exemplo, uma conexão entre o recebimento dos dados da velocidade do veículo e o gráfico correspondente é feito

pelo método:

```
connect(&obdata, SIGNAL(Speed_changed(double)),
        ui->Speed_RTP, SLOT(plot(double)));
```

Os dados então são plotados e salvos pelo objeto *Speed_RTP*, da classe *RTPlot*. Ainda, foi implementado um método que é capaz de calcular a média destes dados recebidos dentro de um intervalo de tempo, tal que um gráfico desta média é mostrado junto ao gráfico dos dados. Utilizando a interface gráfica, o usuário pode ainda configurar esta média da forma que bem entender. Outro método de análise implementado foi uma função da classe *RTPlot* que retorna a integral calculada a partir dos dados do gráfico em um intervalo de tempo também escolhido pelo usuário. Para isso, utilizou-se o método numérico de integração via trapézios compostos, que é suficiente para trabalhar com dados discretos. Desta forma, é possível integrar os dados da velocidade, obtendo a distância percorrida pelo veículo. Estes dados estão disponíveis na aba *Resumo* da aplicação final, que fornece ao usuário alguns resultados dos testes realizados.

Um detalhe importante e prático é que os dados obtidos podem ser salvos em um arquivo de extensão *.log*, que consiste de todos os dados coletados pelo programa, disponibilizados no formato de *valores separados por vírgula (CSV)*, tal que podem ser carregados novamente para a aplicação uma outra hora.

H. Seleção de Dados

Antes da seleção dos dados adquiridos pelo software, houve a definição dos comandos da tabela *"Mode 1"* da biblioteca [2] para o estudo. O veiculo aceita 36 comandos que obtém os dados de gradezas físicas e químicas. Os comandos escolhidos refere-se ao sistemas de motor e transmissão são:

- *MAF*:obtém dados do sensor de fluxo de ar do sistema de arrefecimento automotivo 5, retornando o valor lido em g/s. Com ele é possível saber a quantidade total de ar admitido pelo motor dentro de um intervalo de tempo utilizando o método de integração disponível.
- *RPM* obtém dados do sensor de rotação6, em alguns casos esse sensor não permite a ligação do carro. O valor retornado é apresentado em rpm.
- *SPEED* refere-se a velocidade do veiculo, retorna o valor em km/h.
- *FUEL LEVEL* obtém o nível de combustível através do sensor que esta na tampa do tanque6. O valor é apresentado em porcentagem.
- *INTAKE TEMP* obtém a temperatura do ar de admissão no motor. O valor é retornado em $^{\circ}$C.
- *THROTTLE POS* obtém dados do sensor de posição do acelerador monitora até que ponto a válvula de aceleração (ou lâmina) está aberta, o que é determinado pela distância que o pedal do acelerador foi pressionado. O valor é retornado em porcentagem.
- *COOLANT TEMP* obtém dados do sensor de temperatura do líquido de arrefecimento do motor. O valor é apresentado em $^{\circ}$C.

A equipe decidiu por realizar os testes para a coleta de dados, no dia 15 de novembro, quinta-feira, por volta das 17 horas, na cidade de Joinville. Todas as trajetórias foram consideradas sem congestionamento. Foram ao todo 3 caminhos, respectivamente em rodovia, perímetro urbano e um caminho intermediário.

O caminho 1 é do ponto 34A da BR101 (em uma rotatória da zona industrial) em direção ao Centro de Convenções Expoville (Rua XV de Novembro, 4315), como pode ser visto na figura 2. Esse percurso é puramente rodoviário com distância aproximada de 5km.

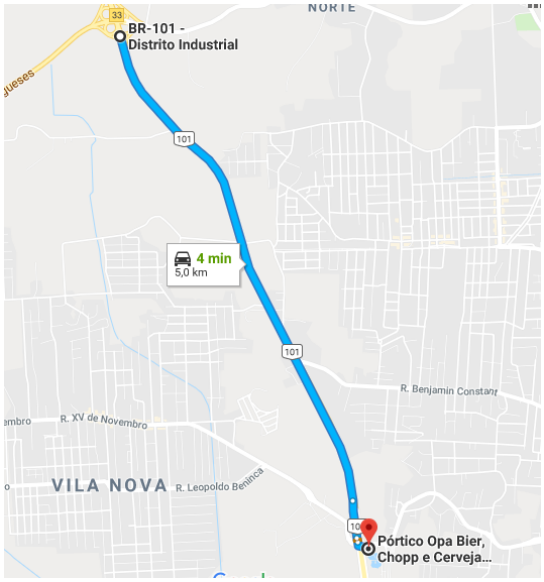

Figura 2. Caminho 1, rodoviário

O caminho 2 é do Centro de Convenções Expoville para a Avenida Blumenau 1700, sua distância é de aproximadamente 5km. O percurso é puramente urbano, como pode ser visto na figura 3.

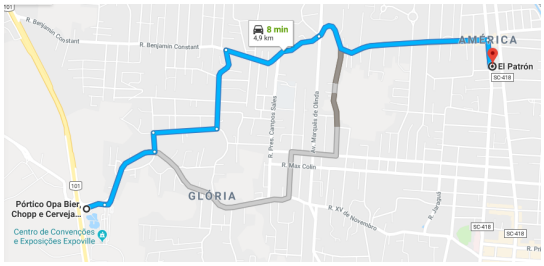

Figura 3. Caminho 2, urbano

O caminho 3 é do Terminal Norte até a UFSC (Centro Tecnológico de Joinville). O percurso é de 7km e é intermediário, com características de vias mais variadas, como pode ser mostrada na figura 4.

1) Razões de obtenção de velocidade: As leituras de velocidade podem ajudar a avaliar, de maneira mais prática, a determinação de limites de velocidade. Se o veículo pode realizar uma trajetória a uma determinada velocidade, considerando padrões psicomotores humanos de condução normais, um novo limite de velocidade pode ser estabelecido em rodovias e em perímetros urbanos, o que pode abrir possibilidades de melhoria de logística.

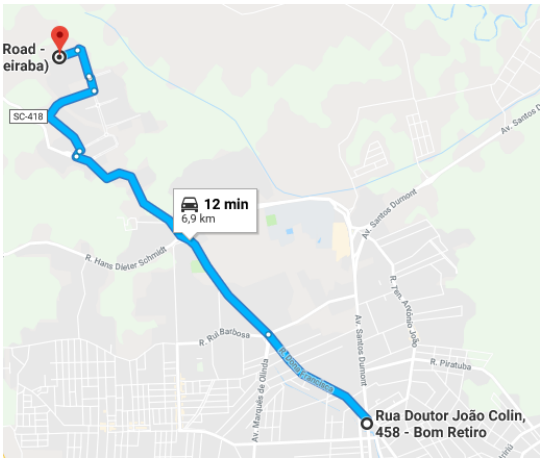

Figura 4. Caminho 3, urbano

2) Razões de obtenção de dados relacionados ao consumo de combustível: Foram utilizados quatro parâmetros de leitura na tentativa de avaliar o consumo de combustível do veículo, as rotações por minuto, nível do combustível no tanque, MAF - índice de fluxo de massa de ar (em gramas por segundo) e a posição relativa do acelerador.

Sobre as condições em que o motor estaria a altas rotações, não significa que há necessariamente o maior consumo de combustível. Considerando, por exemplo, que o uso do freio motor (em condições de declívio), o acelerador está em uma posição em que não permite uma taxa alta de injeção de combustível, mesmo que as rotações por minuto sejam altas; bem como não poderíamos considerar que o momento de maior consumo seria quando o acelerador está posicionado em seu limite máximo, pois se estiver em marcha alta ou em aclive, as rotações por minuto seriam baixas. Assim, assume-se que os pontos onde há um maior consumo seriam os que as rotações por minuto estão altos, juntamente com altos níveis da posição relativa do acelerador.

O sensor de fluxo de massa de ar pode auxiliar nas conclusões de desempenho, quando se trata da temperatura ambiente. Se a temperatura estiver alta demais, a densidade do ar também estará alta e, consequentemente, menos ar (que contém oxigênio) estará disponível para a combustão (a eficiência volumétrica cai). As leituras do sensor de fluxo e as de rotações por minuto podem estar intimamente ligadas, se não houver variação significativa da temperatura (ou seja, da densidade do ar).

3) Razões de obtenção de dados relacionados às temperaturas no motor: Foram coletados os dados das temperaturas de admissão e do fluido de arrefecimento para checar as condições normais de operação e, assim, disponibilizar uma ferramenta que possa alertar ao motorista de leituras anormais de temperatura que possa indicar determinados problemas.

Um deles é a questão da temperatura de válvulas, o aumento anormal de temperatura nesta região pode ser ocasionada pela deposição, decorrente da composição do combustível ou aditivos. Outras causas podem ser a combustão incompleta do combustível ou parâmetros de desenho do motor, causando problemas como pior dirigibilidade, aumento do consumo de combustível, emissões, perda de potência e diminuição da vida útil do motor. Velocidades mais baixas do motor também contribuem para a deposição. As consequências de deposição nas válvulas de admissão podem diminuir o fluxo de ar para dentro da câmara de combustível, diminuindo a eficiência volumétrica do motor, diminuir a qualidade de vaporização do combustível (quando o motor ainda está frio), como mostra a fonte [11].

A importância das leituras das temperaturas do liquido refrigerante é que o sistema de arrefecimento está ligado ao balanço energético do motor. Quanto maior a taxa do combustível queimado, maior transmissão de calor por conta da combustível, resultando em mais rotações por minuto, o que também gera transmissão de calor por atrito. A leitura do MAF pode diminuir enquanto as temperaturas do radiador podem aumentar. Relembrando o conceito de máquina térmica, quanto maior a diferença de temperaturas, melhor a eficiência. Mesmo se o sistema de arrefecimento ter uma melhor capacidade de retirar energia térmica, maior porção da energia consumida do combustível será destinada para o mesmo ao invés da locomoção do veículo.

III. Resultados

Utilizando a versão final do programa desenvolvido, disponível em [1], conectou-se o dispositivo ELM327 ao veículo Chvrolet Onix Effect 1.4 (ano 2014) e obteve-se o protocolo de comunicação *ISO 15765-4 (CAN 11/500)*, através da porta de conexão */dev/rfcomm0*. A seguir, são apresentados os resultados das leituras referentes à velocidade, consumo e temperaturas.

A. Leituras de velocidade

Com relação ao caminho 1, as velocidades tendem a ser, em média, 70km/h, como pode ser visto na figura 5, em que há um ponto indicando tal velocidade média. Os momentos em que a velocidade passou de 90km/h foi em percurso mais retilíneo. O tempo de percurso durou 4 minutos.

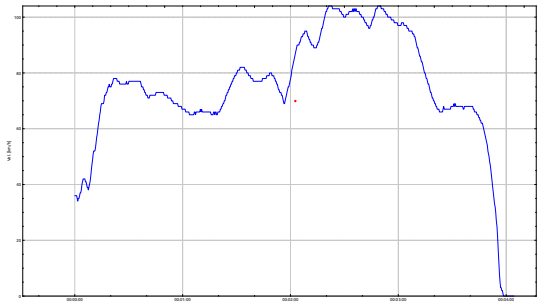

Figura 5. Leituras de velocidade por tempo do caminho 1

Ao observar as velocidades do caminho 2, disponível na figura 6, é possível notar que a velocidade varia bastante, devido ao maior número de semáforos e cruzamentos. O tempo do percurso durou 8 minutos, o dobro em relação ao percurso rodoviário.

O gráfico de velocidades do caminho 3 está disponível na figura 7. As médias total de velocidades nos caminhos 2 e 3

não são adequadas para conclusões, visto que há muitos pontos que permitem velocidades variadas do veículo. Portanto, foram capturadas as médias de intervalos de 20 segundos para cada uma delas, permitindo limites de velocidades mais particulares para cada parte do trajeto.

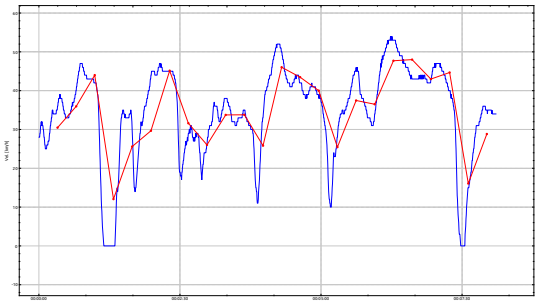

Figura 6. Leituras de velocidade por tempo do caminho 2

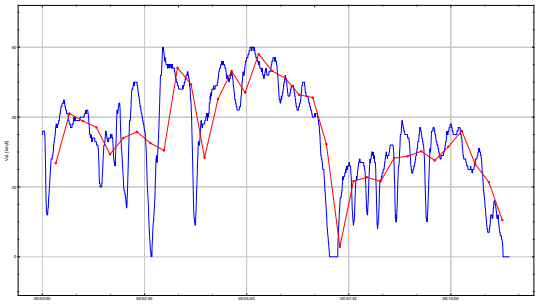

Figura 7. Leituras de velocidade por tempo do caminho 3

B. *Leituras de consumo*

Considerando então as rotações por minuto e a posição do acelerador mais altas, no caminho 1, os dois momentos em que provavelmente houve maior consumo de combustível foram os primeiros 30 segundos e entre 1 minuto e 55 segundos e 2 minutos e 23 segundos. Em ambos os momentos, respectivamente, foram admitidos 0.614kg e 0.775kg de ar, com maior porção de massa de combustível, como podem mostrar as figuras 8 e 9.

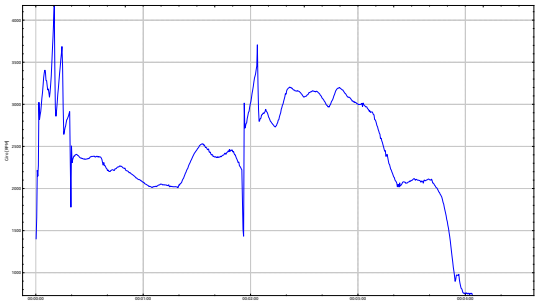

Figura 8. Leituras de rotação por minuto do caminho 1

No caminho 2, o momento onde houve provavelmente o maior consumo foi o tempo entre 1 minuto e 20 segundos e 2 minutos, com a admissão de 0.335kg de ar, como mostram as figuras 10 e 11.

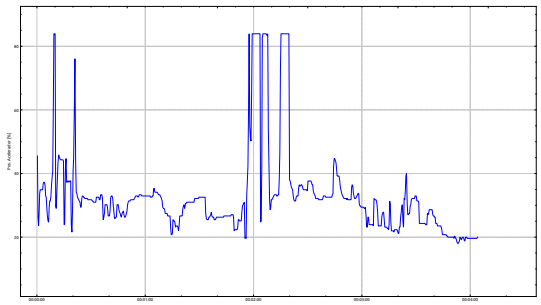

Figura 9. Leituras de posição do acelerador por tempo do caminho 1

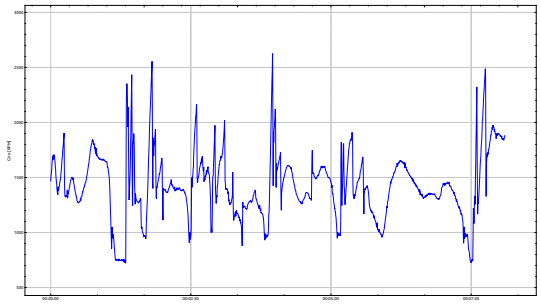

Figura 10. Leituras de rotação por minuto do caminho 2

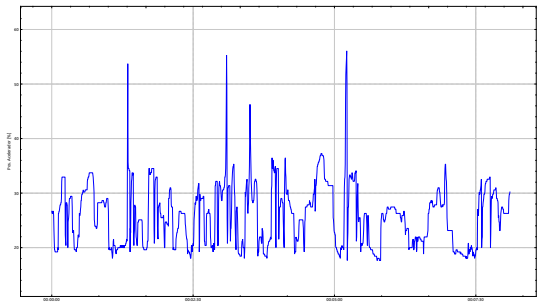

Figura 11. Leituras de posição do acelerador por tempo do caminho 2

Quanto ao caminho 3, houve um maior consumo entre os 2 minutos e 40 segundos e 3 minutos, totalizando uma admissão de 0.372kg de ar, como mostram as figuras 12 e 13.

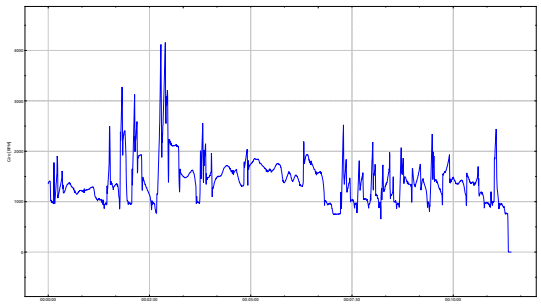

Figura 12. Leituras de posição do acelerador por tempo do caminho 3

C. *Leituras de temperaturas*

A seguir serão apresentadas duas partes: sobre as temperaturas de admissão e sobre temperaturas do líquido do sistema

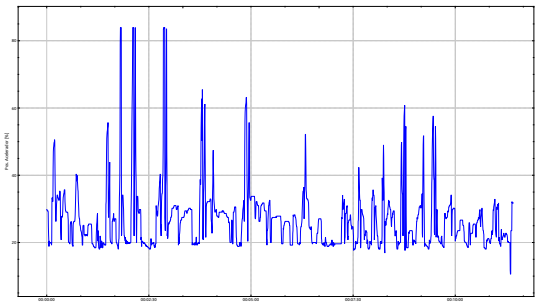

Figura 13. Leituras de posição do acelerador por tempo do caminho 3

de arrefecimento.

1) Temperaturas de admissão: As temperaturas de admissão adquiridas no caminho 1 foram praticamente constantes, com a temperatura de 32ºC, como é mostrado na figura 14.

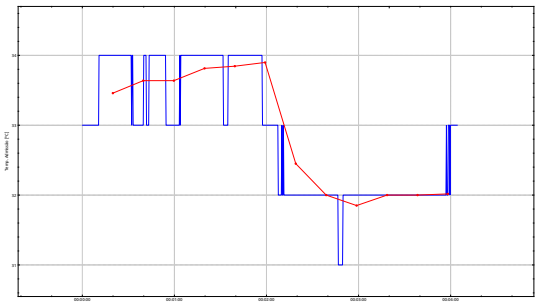

Figura 14. Leituras de temperatura de admissão do caminho 1

Entretanto, quando o veículo começou a realizar o percurso do perímetro urbano (caminho 2), a temperatura aumentou por volta de 38ºC, como mostrado na figura 15.

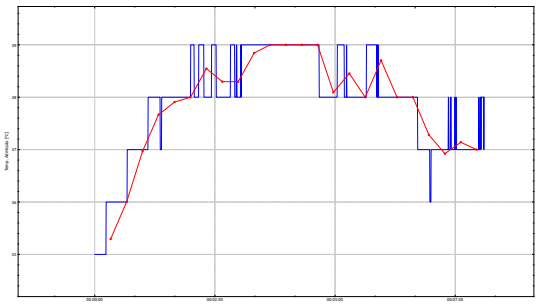

Figura 15. Leituras de temperatura de admissão do caminho 2

Por fim, quando o veículo percorreu o caminho 3, que é intermediário, saindo do perímetro urbano, a temperatura estava ainda maior (por volta de 43ºC) e voltou a abaixar para 38ºC, como pode ser visto na figura 16.

2) Temperaturas de líquido refrigerante: Como podem ser observadas nas figuras 17, 18 e 19, a temperatura de refrigerante que mais prevaleceu foi de 82.0 ºC no caminho 1, enquanto que o do caminho 2, 80.5 ºC e, por último, no caminho 3, a temperatura média foi de 81.8 ºC.

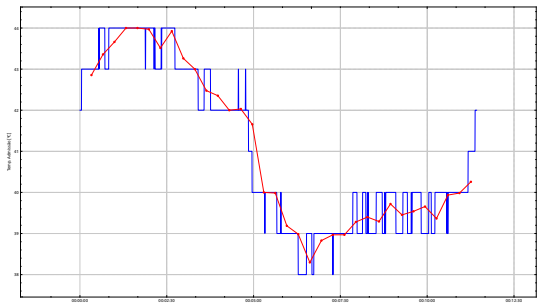

Figura 16. Leituras de temperatura de admissão do caminho 3

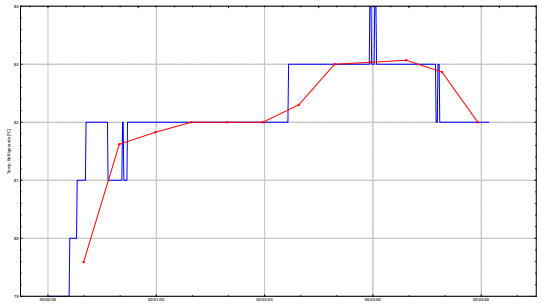

Figura 17. Leituras de temperatura de liquido refrigerante do caminho 1

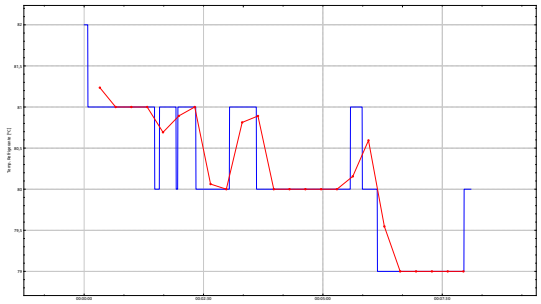

Figura 18. Leituras de temperatura de liquido refrigerante do caminho 2

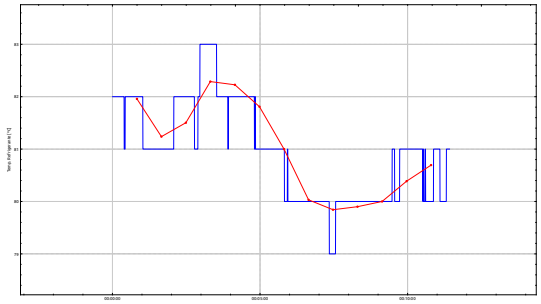

Figura 19. Leituras de temperatura de liquido refrigerante do caminho 3

IV. Conclusão

As considerações finais prezam por aperfeiçoamento de metodologias. A aplicação computacional desenvolvida poderia ser estendida para fornecer mais opções de cálculos analíticos para facilitar a interpretação dos dados. Tal modificação exigiria um tempo maior de desenvolvimento e talvez um projeto de software mais elaborado. Entretanto, o objetivo de introduzir a ideia de uma confecção simplificada de um programa de aquisição de dados de um dispositivo OBD-II foi cumprido. Uma vez que o código disponível em [1] é de uso livre, o leitor, caso interessado, pode dar continuidade ao desenvolvimento do programa da forma que preferir, estando convidado a compartilhar suas modificações com o desenvolvedor, de forma a trazer melhorias ao software.

Os teste, em geral, podem ser feitos em outros horários, com maior congestionamento, para estudos de logística, eficiência veicular etc. Uma sugestão seria realizar testes com limites superiores de velocidade (com medidas de segurança respeitadas), para redefinir quais poderiam ser os novos limites em diversas áreas de uma cidade, considerando parâmetros psicomotores humanos normais. Ainda, os dados coletados poderiam ser agregador em um servidor de *Big Data*, para futuras análises de tráfego urbano.

Do veículo utilizado (Chevrolet Onix Effect), obteve-se que os valores de temperatura do líquido refrigerante lidos apresentaram-se como o oposto ao esperado, visto que o veículo teve que sair de inércia diversas vezes em um trajeto urbano e que esteve exposto a menores correntes de ar para o sistema de arrefecimento. Porém as rotações por minuto na rodovia estiveram mais altas do que no perímetro urbano.

APÊNDICE A
APLICAÇÃO EM PYTHON PARA DE COLETA DE DADOS DA OBD-II

```
# obdpy.py
# !/usr/bin/env python
# -'''- coding: utf-8 -'''-

from pydbus import SessionBus
import obd
import time


bus = SessionBus()
obdata = bus.get("local.OBData", "/OBData")

connection = obd.Async()

obdata.setConnectionStatus(connection.status())
obdata.setProtocolName(connection.protocol_name())
obdata.setPortName(connection.port_name())


def new_rpm(v):
    obdata.setRPM(v.value.magnitude)

def new_speed(v):
    obdata.setSpeed(v.value.magnitude)

def new_fuel_level(v):
  obdata.setFuelLevel(v.value.magnitude)

def new_intake_temp(v):
  obdata.setIntakeTemp(v.value.magnitude)

def new_maf(v):
  obdata.setMAF(v.value.magnitude)

def new_throttle_pos(v):
  obdata.setThrottlePos(v.value.magnitude)

def new_coolant_temp(v):
  obdata.setCoolantTemp(v.value.magnitude)

connection.watch(obd.commands.RPM, callback=new_rpm)
connection.watch(obd.commands.SPEED, callback=new_speed)
connection.watch(obd.commands.FUEL_LEVEL, callback=new_fuel_level)
connection.watch(obd.commands.INTAKE_TEMP, callback=new_intake_temp)
connection.watch(obd.commands.MAF, callback=new_maf)
connection.watch(obd.commands.THROTTLE_POS, callback=new_throttle_pos)
connection.watch(obd.commands.COOLANT_TEMP, callback=new_coolant_temp)

connection.start()

while(True):
  time.sleep(1)
```

APÊNDICE B
CLASSE OBDATA UTILIZADA PARA TROCA DE DADOS

```
//obdata.h
#ifndef OBDATA_H
#define OBDATA_H

#include <QObject>

class OBData : public QObject
{
    Q_OBJECT
public:
    explicit OBData(QObject *parent = nullptr);

    QString getConnectionStatus() const;
    QString getProtocolName() const;
    QString getPortName() const;

    double getRPM() const;
    double getSpeed() const;
    double getFuelLevel() const;
    double getIntakeTemp() const;
    double getMAF() const;
    double getThrottlePos() const;
    double getCoolantTemp() const;

signals:
    void ConnectionStatus_changed();
    void ProtocolName_changed();
    void PortName_changed();

    void RPM_changed(double);
    void Speed_changed(double);
    void FuelLevel_changed(double);
    void IntakeTemp_changed(double);
    void MAF_changed(double);
    void ThrottlePos_changed(double);
    void CoolantTemp_changed(double);

public slots:
    void setConnectionStatus(QString s);
    void setProtocolName(QString s);
    void setPortName(QString s);

    void setRPM(double val);
    void setSpeed(double val);
    void setFuelLevel(double val);
    void setIntakeTemp(double val);
    void setMAF(double val);
    void setThrottlePos(double val);
    void setCoolantTemp(double val);

private:
    QString ConnectionStatus;
    QString ProtocolName;
    QString PortName;

    double RPM;
    double Speed;
    double FuelLevel;
    double IntakeTemp;
    double MAF;
    double ThrottlePos;
    double CoolantTemp;
};

#endif // OBDATA_H
```

REFERÊNCIAS

[1] L. de Camargo Souza, "Um software em qt para aquisição e plotagem de dados de um adaptador obd-ii," https://github.com/lucasdecamargo/OBData, 2018, (Accessed on 11/24/2018).

[2] "python-obd," https://python-obd.readthedocs.io/en/latest/, (Accessed on 11/24/2018).

[3] "Obd-ii pids - wikipedia," https://en.wikipedia.org/wiki/OBD-II_PIDs, (Accessed on 11/24/2018).

[4] "Dbus," https://www.freedesktop.org/wiki/Software/dbus/, (Accessed on 11/24/2018).

[5] "Qt d-bus," http://doc.qt.io/qt-5/qtdbus-index.html, (Accessed on 11/24/2018).

[6] "Lew21/pydbus: Pythonic dbus library," https://github.com/LEW21/pydbus, (Accessed on 11/24/2018).

[7] "[qt dbus] qdbus simple example - the brown box," http://www.hoangvancong.com/2017/11/24/qt-dbus-qdbus-simple-example/, (Accessed on 11/25/2018).

[8] "Qt plotting widget qcustomplot," https://www.qcustomplot.com/, (Accessed on 11/25/2018).

[9] "Qt plotting widget qcustomplot - realtime data demo," https://www.qcustomplot.com/index.php/demos/realtimedatademo, (Accessed on 11/25/2018).

[10] "Qt plotting widget qcustomplot - setting up," https://www.qcustomplot.com/index.php/tutorials/settingup, (Accessed on 11/25/2018).

[11] Z. Stępień, "Intake valve and combustion chamber deposits formation – the engine and fuel related factors that impacts their growth," *NAFTA-GAZ*, vol. 70, no. 4, pp. 236–242, 2014. [Online]. Available: http://archiwum.inig.pl/inst/nafta-gaz/nafta-gaz/Nafta-Gaz-2014-04-04.pdf