

UNIVERSIDADE DE SÃO PAULO
ESCOLA DE ENGENHARIA DE SÃO CARLOS
DEPARTAMENTO DE ENGENHARIA ELÉTRICA
E DE COMPUTAÇÃO

**Monitoramento e Gestão de uma Frota de
Veículos Utilizando Sistemas Embarcados**

Autor: Lucas Vieira Pacheco

Orientador: Prof. Dr. Evandro Luis Linhari Rodrigues

São Carlos

2016

Lucas Vieira Pacheco

Monitoramento e Gestão de uma Frota de Veículos Utilizando Sistemas Embarcados

Trabalho de Conclusão de Curso apresentado
à Escola de Engenharia de São Carlos, da
Universidade de São Paulo

Curso de Engenharia Elétrica

ORIENTADOR: Prof. Dr. Evandro Luis Linhari Rodrigues

São Carlos

2016

AUTORIZO A REPRODUÇÃO TOTAL OU PARCIAL DESTE TRABALHO,
POR QUALQUER MEIO CONVENCIONAL OU ELETRÔNICO, PARA FINS
DE ESTUDO E PESQUISA, DESDE QUE CITADA A FONTE.

P657m Pacheco, Lucas Vieira
Monitoramento e Gestão de uma Frota de Veículos
Utilizando Sistemas Embarcados / Lucas Vieira Pacheco;
orientador Evandro Luis Linhari Rodrigues. São Carlos,
.

Monografia (Graduação em Engenharia Elétrica com
ênfase em Eletrônica) -- Escola de Engenharia de São
Carlos da Universidade de São Paulo, .

1. sistemas embarcados. 2. Linux embarcado. 3.
gerenciamento de frotas. 4. sistemas de diagnose de
bordo. 5. OBD-II. I. Título.

FOLHA DE APROVAÇÃO

Nome: Lucas Vieira Pacheco

Título: “Monitoramento e gestão de uma frota de veículos utilizando sistemas embarcados”

Trabalho de Conclusão de Curso defendido e aprovado
em 24 / 11 / 2016,

com NOTA 10,0 (DEZ, ZERO), pela Comissão Julgadora:

Prof. Associado Evandro Luis Linhari Rodrigues - Orientador - SEL/EESC/USP

Mestre Alex Antonio Affonso - Doutorando - SEL/EESC/USP

Mestre Leonardo Mariano Gomes - Doutorando - SEL/EESC/USP

Coordenador da CoC-Engenharia Elétrica - EESC/USP:
Prof. Associado José Carlos de Melo Vieira Júnior

Dedicatória

Dedico este trabalho a minha família que sempre forneceu todo o apoio necessário para que eu pudesse alcançar os meus objetivos.

Lucas Vieira Pacheco

Agradecimentos

Agradeço a minha família pelo apoio e confiança depositados em mim para a execução deste trabalho. Agradeço em especial a meu pai por ter sugerido há alguns anos atrás a ideia de desenvolver uma plataforma para o gerenciamento de um frota de veículos.

Agradeço a meus amigos Artur Loureiro e Leonardo Mariano pelas discussões e sugestões que contribuíram para o desenvolvimento e o aperfeiçoamento deste trabalho.

Agradeço a minha irmã Ana Luísa por sua ajudar na correção e revisão do texto.

Agradeço ao Prof. Evandro pela ajuda fornecida e por acreditar no potencial deste projeto desde o início.

Lucas Vieira Pacheco

Resumo

Este trabalho contemplou o desenvolvimento de um sistema para o gerenciamento de uma frota de veículos. A solução proposta é composta por duas partes: um sistema embarcado e uma plataforma *web* para visualização de dados. O sistema embarcado é comandado por uma placa *Raspberry Pi 3*, sendo capaz de obter dados de geolocalização por GPS e informações disponibilizadas pelo computador de bordo de um veículo como, por exemplo, velocidade e distância percorrida. A comunicação com o computador de bordo é realizada através da porta OBD. Todos os dados coletados são enviados automaticamente para um servidor que hospeda a plataforma *web*. Nela é possível planejar as manutenções que precisam ser realizadas, conferir os trajetos realizados pelos veículos e obter informações sobre como o veículo é conduzido. O projeto desenvolvido simplifica o gerenciamento de uma frota e permite o agendamento de manutenções nos prazos adequados.

Palavras-Chave: sistemas embarcados, Linux embarcado, gerenciamento de frotas, sistemas de diagnose de bordo, OBD-II.

Abstract

This work covers the development of a fleet management system for light vehicles. The proposed solution is composed of two parts: an embedded system and a website for data visualization. The embedded system is based on a Raspberry Pi 3 computer, and is capable of acquiring geolocation data via GPS and data provided by the vehicle ECU, such as speed and traveled distance. The communication between the ECU and the embedded system is performed through the OBD-II interface. Collected data is automatically uploaded to a server. The website allows to schedule maintenance actions, verify the paths traveled by the vehicles and learn more about how the vehicles are driven. The developed project reduces fleet management efforts and allows to schedule and to follow maintenances actions properly.

Keywords: Embedded systems, Linux for embedded systems, fleet management, on-board diagnostics, OBD-II.

Lista de Figuras

2.1	Conector SAE J1962 Fêmea (Fonte: <i>Wiring Diagram</i>)	27
3.1	Raspberry Pi 3 Model B	43
3.2	Descrição dos pinos de entrada e saída da <i>Raspberry Pi 3 Model B</i>	45
3.3	Adaptadores OBD utilizados no projeto	45
3.4	Adaptador OBD instalado no veículo	46
3.5	Módulo GPS utilizado	49
3.6	Organização do <i>software</i> do sistema embarcado	50
3.7	Página <i>web</i> gerada por <i>Flask</i> com o uso de um <i>template</i>	58
3.8	Exemplo da utilização da função <i>jsonify</i> em <i>Flask</i>	59
3.9	Exemplo de gráfico de velocidade produzido com a biblioteca <i>Morris.js</i> para <i>JavaScript</i>	61
3.10	Exemplo do uso da API do <i>Google Maps</i> para <i>JavaScript</i>	63
4.1	Exemplo de dados de geolocalização coletados pelo sistema embarcado e armazenados em um banco de dados <i>SQLite</i>	65
4.2	Exemplo de dados coletados pelo sistema embarcado através da interface OBD de um veículo. Dados estão armazenados em um banco de dados <i>SQLite</i>	66
4.3	Página que contém informações sobre todos os veículos da frota	67
4.4	Página desenvolvida para a inclusão de um novo veículo à frota	69
4.5	Topo da página que contém detalhes sobre um veículo. São mostradas informações sobre o veículo, manutenções, distância percorrida nos últimos dias e alertas gerados pelo sistema.	70
4.6	Trajeto realizado por um veículo e sua velocidade em função do tempo. Os dados foram coletados pelo sistema embarcado e são exibidos pela plataforma <i>web</i>	71

4.7 Velocidade de um veículo e a carga do motor durante um determinado trajeto. Os dados foram coletados pelo sistema embarcado e são exibidos pela plataforma *web*. A carga do motor indica, em termos percentuais, a razão entre o torque instantâneo e o torque máximo disponível. 72

Lista de Tabelas

2.1	Identificação dos pinos do conector SAE J1962	27
2.2	Formato da sentença GGA	29
2.3	Formato da sentença ZDA	30

Sumário

1	Introdução	21
1.1	Motivação	21
1.2	Objetivos	23
1.3	Organização do trabalho	24
2	Embasamento Teórico	25
2.1	OBD	25
2.1.1	Conector SAE J1962	26
2.2	GPS	27
2.2.1	Formato dos dados GPS	27
2.3	SQLite	30
2.3.1	Tipos de dados	31
2.3.2	Alguns comandos <i>SQL</i>	31
2.4	Python	35
2.4.1	<i>SQLite</i> em <i>Python</i>	36
3	Materiais e Métodos	41
3.1	Sistema embarcado	41
3.2	<i>Hardware</i> do Sistema Embarcado	42
3.2.1	Raspberry Pi	43
3.2.2	Adaptador OBD	44
3.2.3	Módulo GPS	48
3.3	<i>Software</i> do Sistema Embarcado	49
3.3.1	<i>Threading</i>	50
3.3.2	<i>GPS_serial</i>	52
3.3.3	<i>OBDMonitor</i>	53

3.3.4	<i>UploadManager</i>	55
3.3.5	<i>VehicleMonitor</i>	55
3.4	Plataforma <i>Web</i>	56
3.4.1	Flask	56
3.4.2	<i>Upload</i> de arquivos	59
3.4.3	Gráficos	60
3.4.4	<i>Google Maps</i> API	61
3.4.5	Troca de dados entre o servidor e <i>JavaScript</i>	63
4	Resultados	65
4.1	Sistema embarcado	65
4.2	Plataforma <i>Web</i>	66
4.2.1	Resumo	66
4.2.2	Adicionando um veículo à frota	67
4.2.3	Dados sobre um veículo	67
5	Conclusões	73
5.1	Conclusões gerais	73
5.2	Trabalhos futuros	74

Capítulo 1

Introdução

1.1 Motivação

Atualmente, muitas empresas necessitam de veículos para desempenharem suas atividades. Uma empresa pode utilizar sua frota para o transporte de bens, de mercadores, ou para o deslocamento de seus colaboradores em suas atividades diárias.

Visto que os veículos podem representar uma parcela considerável do patrimônio de uma empresa, e visto os altos custos de operação (combustível, seguro, manutenção preventiva, ...), faz-se necessária a gestão eficiente da frota de veículos para a redução de custos e de riscos. É importante ressaltar que os custos e a dificuldade de gerenciamento aumentam de acordo com o número de veículos que a empresa possui.

Para solucionar essa questão, há no mercado diversos *softwares* para a gestão de frotas. Eles são capazes de definir os trajetos de cada veículo a fim de reduzir a distância percorrida, de agendar manutenções preventivas, monitorar o consumo de combustível, e até mesmo de avisar quando é necessário renovar o seguro, entre outras funções. Entretanto, para que um *software* de gestão de frotas funcione corretamente é necessário fornecer manualmente os dados do odômetro de cada veículo para determinar a distância percorrida, e assim, por exemplo, agendar as manutenções. Para o monitoramento do consumo de combustível, é necessário fornecer a quantidade de combustível em cada abastecimento e o nível do tanque do veículo. Isso torna o sistema propenso a erros humanos e exige disciplina para que os dados sejam fornecidos regularmente.

Outra ferramenta disponível para o controle de frotas é o rastreador por GPS. Ele permite consultar a posição de cada veículo através de ferramentas oferecidas pelas empresas de monitoramento. Entretanto, não existe integração entre o rastreador e o *software* de gestão de

veículos.

A proposta desse projeto é desenvolver, de maneira integrada, uma plataforma para gestão de frotas e um sistema embarcado para os veículos monitorados. O sistema embarcado é responsável por coletar dados do veículos e enviá-los automaticamente para um servidor, sem a necessidade de intervenção humana. Isso torna essa solução mais eficiente e mais confiável. Além disso, o sistema embarcado funcionará como um *data logger*, o qual registra de maneira contínua informações sobre o veículo.

O sistema embarcado coleta os dados de duas maneiras: através de um módulo GPS e da leitura de informações do computador de bordo do veículo. O módulo GPS permite determinar a localização do veículo e dessa forma os trajetos percorridos. A comunicação com o computador de bordo do veículo permite a obtenção de informações como velocidade, rotação do motor, nível de combustível no tanque, distância percorrida, entre outras. A comunicação com o computador de bordo é possível graças à interface automotiva OBD. Por se tratar de uma interface padronizada, o sistema embarcado funcionará em qualquer veículo que possua esse recurso.

O sistema OBD, sigla em inglês para *on-board diagnostics*, é um sistema de diagnose de bordo, e atualmente está presente em todos os veículos leves comercializados no Brasil. Esse sistema foi criado nos Estados Unidos para reduzir as emissões veiculares, e posteriormente foi adotado em diversos outros países. Seu objetivo principal é que o veículo seja capaz de medir seus próprios índices de emissão e de reportar falhas automaticamente, sem a necessidade de nenhum equipamento externo de medição. Os dados coletados pelo sistema OBD podem ser lidos através de um equipamento específico. Por se tratar de um sistema padronizado, todos os carros utilizam o mesmo conector e um dos protocolos permitidos. Além disso, os dados são lidos da mesma maneira, independente da marca ou do modelo do veículo.

O sistema proposto, ao medir de maneira conjunta dados de geolocalização e do computador de bordo, obtém muito mais informações sobre o veículo que em qualquer outro sistema atualmente disponível no mercado brasileiro. O envio automático dos dados para um servidor é um diferencial e aumenta a confiabilidade do sistema. O rastreamento do veículo permite determinar o tempo gasto em cada trajeto e detectar se houve algum desvio. O monitoramento da velocidade revela mais detalhes sobre como os motoristas conduzem os veículos. Pode mostrar se o limite de velocidade foi ultrapassado, ou se há acelerações e frenagens bruscas. Por fim, a leitura da distância percorrida permite agendar com antecedência manu-

tenções preventivas sem a necessidade de leitura do odômetro. Tal recurso simplifica a gestão da frota e melhora a capacidade de planejamento da empresa.

1.2 Objetivos

O objetivo deste trabalho é o desenvolvimento, de maneira integrada, de um sistema embarcado para o monitoramento de veículos e de uma plataforma *web* para o gerenciamento de uma frota de veículos monitorados.

Deseja-se desenvolver um sistema embarcado constituído por uma placa *Raspberry Pi*, um adaptador OBD e um módulo GPS. A placa *Raspberry Pi*, que é um computador que utiliza o sistema operacional *Linux*, ocupa papel central no sistema. Ela é responsável por coletar e armazenar os dados obtidos através do adaptador OBD e do módulo GPS. Além disso, ela deve transmitir os dados ao servidor que hospeda a plataforma *web*. O adaptador OBD permite a leitura de dados ao comunicar-se diretamente com o computador de bordo do veículo. O módulo GPS é utilizado para determinar a posição do veículo durante seus deslocamentos.

Desta forma, destacam-se os seguintes objetivos para o desenvolvimento do sistema embarcado:

- utilização de *Linux* embarcado;
- leitura de dados de um veículo através da interface OBD;
- leitura da posição do veículo através do módulo GPS;
- transmissão de dados via *web*

A plataforma *web* deverá fornecer informações sobre todos os veículos, assim como disponibilizar aos usuários os dados coletados pelo sistema embarcado. Destacam-se os seguintes objetivos:

- desenvolvimento de uma plataforma *web* voltada para a gestão de uma frota de veículos;
- comunicação com o sistema embarcado para a transferência de dados;
- permitir a visualização dos dados coletados pelo sistema embarcado;
- agendar manutenções automaticamente a partir dos dados fornecidos pelo sistema embarcado.

1.3 Organização do trabalho

Este trabalho apresenta no capítulo Embasamento Teórico uma explicação sobre todas as tecnologias e técnicas que foram utilizadas no projeto. O capítulo Materiais e Métodos descreve os materiais utilizados no sistema embarcado, e a maneira como foram escritos os códigos do sistema embarcado e da plataforma *web*. O capítulo Resultados avalia o funcionamento do sistema embarcado e mostra como a solução desenvolvida pode ser utilizada para a gestão de uma frota de veículos.

Finalmente, no capítulo Conclusão faz-se uma avaliação do que foi realizado no projeto e os próximos passos a serem seguidos para que ele possa tornar-se um produto disponível para o grande público.

Capítulo 2

Embasamento Teórico

2.1 OBD

O sistema *OBD*, sigla em inglês para *on-board diagnostics*, é um sistema automotivo de autodiagnóstico. A motivação principal para sua criação é o controle de emissões veiculares. Nesse sistema, o próprio veículo é capaz de medir o nível de emissões e de produzir alertas caso os resultados não estejam de acordo com as normas estabelecidas. Esse sistema, conforme sua evolução ao longo dos anos, também passou a fornecer dados que podem auxiliar em outros testes e facilitar na identificação de falhas no veículo.

O primeiro padrão introduzido foi o OBD-I, em 1988, nos Estados Unidos pelo *California Air Resources Board* (CARB) [1]. Capaz de identificar falhas de emissão em veículos, esse padrão foi obrigatório em todos os veículos novos vendidos no estado da Califórnia. A indicação de falhas é feita por lâmpada no painel. O padrão OBD-I, embora limitado se comparado aos sistemas disponíveis atualmente, foi um importante passo na história dos sistemas de autodiagnóstico ao definir um modelo a ser seguido por todos os fabricantes e ao incentivar a produção de veículos mais eficientes.

Entretanto, o padrão OBD-I apresentava um problema que impedia que ele fosse amplamente utilizado: a falta de padronização entre os fabricantes. Cada fabricante poderia utilizar o conector que desejasse, e em alguns casos, as informações do computador de bordo poderiam apenas ser acessadas por ferramentas caras que apenas concessionárias tinham acesso.

O padrão OBD-II surgiu em 1990 através do *Clean Air Act Amendments*, passando a ser obrigatório para todos os veículos leves vendidos nos Estados Unidos a partir de 1996. Esse padrão expandiu a capacidade do padrão OBD-I e teve como prioridade a padronização. Ele estabeleceu o conector a ser utilizado e a função de cada pino, o formato das mensagens

trocadas no sistema, assim como a maneira de reportar medições e problemas diagnosticados pelo sistema. Isso permitiu, por exemplo, o desenvolvimento de ferramentas de testes compatíveis com todos os veículos, independente da marca e do fabricante.

A expansão da capacidade de testes permitiu a detecção de um maior número de falhas no veículo. O padrão OBD-II também permite avaliar a degradação do nível de emissões de veículos ao longo do tempo. Do ponto de vista regulatório, o OBD-II tornou o processo de inspeção mais simples e mais rápido, reduzindo custos. Para os consumidores também houve vantagens. O monitoramento constante avisa o condutor no momento em que uma falha é identificada. Isso permite que o veículo seja encaminhado à manutenção antes que essa falha afete outros componentes, e em alguns casos, antes da expiração do período de garantia.

No Brasil, a regulamentação de sistemas de diagnose de bordo ocorreu por meio da resolução nº 354 do Conselho Nacional do Meio Ambiente (CONAMA), no âmbito do Programa de Controle de Poluição do Ar por Veículos Automotores (PRONCOVE). Ela estabeleceu que a implantação nos veículos novos deveria ocorrer em duas fases, chamadas de OBDBr-1 e OBDBr-2.

O sistema OBDBr-1 possui as características mínimas para a detecção de falhas para a avaliação de funcionamento dos sistemas de ignição e de injeção de combustível. Sua implantação iniciou-se em 2007, quando pelo menos 40% dos veículos leves de passageiros, produzidos no Brasil ou importados, deveriam atender suas especificações. A partir de 1º de janeiro de 2009, tornou-se obrigatório em todos os veículos novos.

O sistema OBDBr-2 é uma expansão do sistema OBDBr-1, podendo detectar um número maior de falhas. Sua implantação iniciou-se em 2010, sendo obrigatória em todos os veículos leves de passageiros ou comerciais a partir de 1º de janeiro de 2011.

2.1.1 Conector SAE J1962

A norma SAE J1962 define o conector a ser usado pelo sistema OBD-II, figura 2.1. Esse também é o conector usado nos sistemas OBDBr-1 e OBDBr-2. A norma especifica que o conector deve estar posicionado no interior do veículo, em um painel que seja facilmente acessível a partir do banco do motorista. Caso haja alguma tampa ou proteção no conector ou no painel, esta deve ser facilmente removível sem o uso de nenhuma ferramenta.

O conector SAE J1962 possui 16 pinos, havendo pinos reservados para os cinco protocolos automotivos previstos na norma OBD-II: SAE J1850 PWM, SAE J1850 VPW, ISO 9141-2, ISO 14230 e ISO 15765. Há também um pino que fornece alimentação diretamente

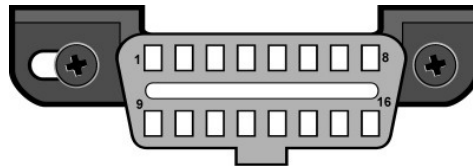


Figura 2.1: Conector SAE J1962 Fêmea (Fonte: *Wiring Diagram*)

da bateria do veículo. A tabela 2.1 contém a descrição completa dos pinos desse conector.

Tabela 2.1: Identificação dos pinos do conector SAE J1962

Pino	Função
1	Reservado ao fabricante
2	Sinal positivo SAE J1850 PWM e VPW
3	Reservado ao fabricante
4	Terra do chassi
5	Terra do sinal
6	CAN High ISO 15765
7	K-Line ISO 9141-2 e ISO 14230
8	Reservado ao fabricante
9	Reservado ao fabricante
10	Sinal negativo SAE J1850 PWM
11	Reservado ao fabricante
12	Reservado ao fabricante
13	Reservado ao fabricante
14	Can Low ISO 15765
15	L-Line ISO 9141-2 e ISO 14230
16	Terminal positivo da bateria

2.2 GPS

2.2.1 Formato dos dados GPS

Os dispositivos GPS existentes atualmente utilizam um formato padronizado criado pelo órgão norte-americano NMEA (sigla em inglês para *National Marine Electronics Association*). O formato NMEA é uma especificação que define a comunicação entre os diversos

dispositivos eletrônicos utilizados em embarcações [2]. Desta forma, equipamentos de diferentes fabricantes podem ser utilizados em conjunto, e programas computacionais são compatíveis com todos os dispositivos que adotam este formato.

No formato NMEA, as informações são enviadas através de mensagens codificadas em caracteres ASCII. As mensagens são iniciadas pelo caractere \$, que é seguido por uma sequência de campos separados por vírgulas. No final de cada mensagem, há o caractere *, um *checksum* e o marcador de fim de linha (caracteres ASCII "\r" e "\n"). O *checksum* é utilizado para garantir a integridade das mensagens recebidas. Ele corresponde a operação XOR (ou exclusivo), feita *bit a bit* em todos os caracteres da mensagem, incluindo as vírgulas que separam os campos, porém excluindo os caracteres \$ e * no início e no final das mensagens.

O primeiro campo da mensagem é composto por cinco caracteres. Ele indica o tipo de dispositivo que a enviou (dois primeiros caracteres) e o tipo de mensagem (3 caracteres seguintes). Mensagens de dispositivos GPS sempre se iniciam com "GP". Os fabricantes também podem utilizar mensagens proprietárias que não estão previstas no formato NMEA. Nesse caso, o primeiro campo da mensagem começa com o caractere P, seguido por três caracteres que correspondem ao código do fabricante e outros caracteres usados para identificar o tipo de mensagem.

Podemos tomar como exemplo a seguinte mensagem:

```
$GPGLL,2200.67129,S,04754.97951,W,030250.00,A,A*68
```

O primeiro campo de dados inicia-se com GP, pois foi transmitida por um dispositivo GPS, e trata-se de uma mensagem do tipo GLL (*Geographic Position - Latitude and Longitude*). Por fim, há o caractere * e *checksum* correspondente aos caracteres ASCII "68".

A especificação NMEA prevê que as mensagens sejam enviadas através do protocolo RS422 a uma taxa de 4800 *bits* por segundo. Entretanto, a maioria dos dispositivos GPS emprega comunicação serial RS232 a uma taxa de 9600 *bits* por segundo, com 8 *bits* de dados, nenhum *bit* de paridade e um *stop bit*.

Os dispositivos GPS que funcionam de acordo com a especificação NMEA enviam dados de maneira contínua, sem haver necessidade de requisitá-los. No caso de um sistema embarcado que trabalha com um módulo GPS, basta ler as mensagens transmitidas através da porta serial e interpretá-las para determinar a posição. Entretanto, a especificação NMEA prevê que os dispositivos também podem receber mensagens, o que normalmente é utilizado para configurações.

Algumas sentenças NMEA para dispositivos GPS

Descreve-se aqui duas mensagens utilizadas por dispositivos GPS. Elas mostram como informações sobre geolocalização, data e hora podem ser obtidas. Dentre as mensagens previstas na especificação NMEA, destaca-se a GGA por conter informações sobre latitude, longitude, altitude e hora.

GGA: fornece informações sobre a posição em 3D (latitude, longitude e altitude) e sobre a precisão das medidas

Tabela 2.2: Formato da sentença GGA

Campo	1	2	3	4	5	6	7	8
Formato	\$--GGA	hhmmss.ss	llll.ll	a	yyyy.yy	a	x	xx x.x
Campo	9	10	11	12	13	14	15	
Formato	x.x	x	x.x	x	x.x	xxxx	*hh	

Os campos descritos correspondem na tabela 2.2 a:

1. Horário UTC (horas, minutos, segundos e dezenas de segundos)
2. Latitude em graus, composta por 6 dígitos, sendo 2 decimais
3. Norte (N) ou sul (S)
4. Longitude em graus, composta por 7 dígitos, sendo 2 decimais
5. Leste (E) ou Oeste (W)
6. Indicador da qualidade da leitura
7. Números de satélites em vista, entre 0 e 12
8. Diluição horizontal de precisão em metros
9. Altitude da antena acima ou abaixo do nível do mar
10. Unidade da altitude da antena
11. Separação geoidal
12. Unidade da separação geoidal

13. Idade dos dados de GPS diferencial (DGPS). Em branco se DGPS não é utilizado
14. ID da estação diferencial de referência, entre 0 e 1023
15. *checksum*

ZDA: fornece informações sobre data e horário

Tabela 2.3: Formato da sentença ZDA

Campo	1	2	3	4	5	6	7
Formato	\$--ZDA	hhmmss.ss	xx	xx	xxxx	xx	xx *hh

Os campos descritos correspondem na tabela 2.3 a:

1. Horário UTC (horas, minutos, segundos, dezenas de segundos)
2. Dia, entre 01 e 31
3. Mês, entre 01 e 12
4. Ano utilizando 4 dígitos
5. Descrição da zona local, entre 00 e +- 13 horas
6. Descrição da zona local em minutos
7. *checksum*

Pode-se encontrar na referência [2] uma lista completa de todas as mensagens para dispositivos GPS da especificação NMEA, assim como a descrição de cada mensagem.

2.3 SQLite

SQLite [3] é uma implementação de banco de dados *SQL*. Todo o código encontra-se em domínio público, e pode ser usado gratuitamente, inclusive para fins comerciais. Ao contrário de outros bancos de dados, como por exemplo *MySQL* e *PostgreSQL*, *SQLite* não utiliza o modelo servidor-cliente. Isso significa que não há um processo separado para o servidor, e a biblioteca *SQLite* lê e escreve diretamente arquivos no disco. Outra característica importante é que, por não empregar um servidor, *SQLite* não precisa ser configurado antes de seu uso (*zero-conf*).

Embora seja escrito em *C*, há extensões que permitem a utilização de *SQLite* em outras linguagens de programação. Pode-se citar, por exemplo, *C++*, *java*, *javascript*, *python*, *R*, *php*, entre outros.

SQLite é a implementação de banco de dados mais utilizada no mundo. Está presente em aparelhos celulares *Android*, *iPhones*, computadores *Mac*, e em programas, como por exemplo, *iTunes*, *Skype*, *Firefox* e *Google Chrome*.

Trata-se de uma excelente opção para sistemas embarcados por se tratar de uma biblioteca compacta. Em alguns casos, ocupa menos de 500KB de espaço em disco. Além disso, por não haver um servidor, não há a necessidade de um administrador ou de intervenção humana.

2.3.1 Tipos de dados

Os dados armazenados em um banco de dados *SQLite* podem ter cinco tipos diferentes:

INTEGER: armazena números inteiros, positivos e negativos, utilizando 1, 2, 3, 4, 6 ou 8 *bytes*;

REAL: armazena números no formato de ponto flutuante. São utilizados 8 *bytes*;

TEXT: armazena texto utilizando as codificações UTF-8, UTF-16BE ou UTF-16LE;

BLOB: armazena valores binários. Pode ser utilizado, por exemplo, para armazenar arquivos;

NULL: utilizado para indicar que um valor é desconhecido.

2.3.2 Alguns comandos *SQL*

São apresentados na sequência alguns comandos *SQL* utilizados para manipular um banco de dados. Embora não seja uma descrição completa de todos os comandos disponíveis em *SQLite*, eles cobrem tudo o que foi utilizado neste trabalho, de forma a permitir que o leitor compreenda o código que foi escrito.

Criação de uma tabela

Em um banco de dados, as informações são armazenadas sob o formato de tabelas. Um banco de dados pode conter uma ou mais tabelas. A cada coluna de uma tabela estão associados um nome e o tipo de dados que ela armazena. Também é possível determinar se a coluna

aceita valores do tipo *NULL*, ou se ela tem a propriedade de *primary key*, isto é, não podem haver valores repetidos.

Para criar uma tabela, utiliza-se o comando *CREATE TABLE*. A sintaxe utilizada é a seguinte:

```
CREATE TABLE nome_da_tabela (
    coluna1 datatype ,
    coluna2 datatype ,
    ...
)
```

Por exemplo, deseja-se criar uma tabela contendo o cadastro de pessoas. É necessário armazenar os seguintes dados: nome completo, idade, número do documento e cidade de origem. O número do documento é único para cada pessoa e não pode ter dois ou mais valores iguais na tabela. Para isso, será associado a ele a propriedade *primary key*. Com exceção da cidade de origem, todos os campos são obrigatórios, o que é especificado por *NOT NULL* na declaração de uma coluna. Tal tabela pode ser criada a partir do seguinte comando:

```
CREATE TABLE cadastro (
    nome TEXT NOT NULL,
    idade INTEGER NOT NULL,
    documento INTEGER PRIMARY KEY NOT NULL,
    cidade TEXT
)
```

Informações sobre uma tabela

As informações sobre uma tabela podem ser obtidas através do comando *PRAGMA table_info(tablename)*. Esse comando retorna para cada coluna da tabela as seguintes informações: índice da coluna, nome da coluna, tipo de dado, se a coluna não aceita valores do tipo *NULL*, valor padrão da coluna e se tem a propriedade de *primary key*. O exemplo a seguir ilustra o uso desse comando:

```
>PRAGMA TABLE_INFO(example)
0|col1|text|0||0
1|col2|real|0||0
2|col3|text|0||0
```


Inserindo dados em uma tabela

Para inserir uma linha em uma tabela, utiliza-se o comando *INSERT INTO*. A sintaxe deste comando é a seguinte:

```
INSERT INTO nome_da_tabela (coluna 1, coluna 2, ..., coluna n)
      VALUES (valor 1, valor 2, ..., valor n)
```

Caso todas as colunas sejam utilizadas, não é necessário declará-las. Entretanto, a ordem dos valores deve ser a mesma ordem que a ordem das colunas na tabela. Colunas que não possuem a propriedade *NOT NULL* podem ser omitidas quando os dados são inseridos na tabela. Valores do tipo texto devem ser declarados entre aspas.

O exemplo a seguir mostra como inserir dados na tabela criada no exemplo anterior. No primeiro comando, todas as colunas são utilizadas. No segundo, omite-se a coluna cidade pois ela não possui a propriedade *NOT NULL*. No terceiro comando, as colunas não são especificadas.

```
INSERT INTO cadastro (nome, idade, documento, cidade)
      VALUES ("Lucas", 24, 1234, "Sao Carlos")
INSERT INTO cadastro (nome, idade, documento)
      VALUES ("Hilma", 55, 5678)
INSERT INTO cadastro VALUES ("Ana", 21, 1357, "Guaratingueta")
```

Lendo dados de uma tabela

Para obter dados de uma tabela, é utilizado o comando *SELECT*. A sintaxe deste comando é a seguinte:

```
SELECT coluna 1, coluna 2, ..., coluna n FROM nome_da_tabela
```

Para obter dados de todas as colunas, a lista de colunas pode ser substituída pelo caractere *:

```
SELECT * FROM nome_da_tabela
```

O exemplo a seguir ilustra a aplicação deste comando na tabela *cadastro*, criada anteriormente.

```
> SELECT nome, idade FROM cadastro;
Lucas|24
Ana|21
Hilma|55
```

```
> SELECT * FROM cadastro;
Lucas|24|1234|Sao Carlos
Ana|21|1357|Guaratingueta
Hilma|55|5678|
```

Filtrando dados de uma tabela

É possível filtrar os dados de uma tabela através da cláusula *WHERE*. Ela pode ser utilizada com comandos como *SELECT*, *UPDATE* e *DELETE*. No caso do comando *SELECT*, a sintaxe é a seguinte:

```
SELECT coluna 1, ..., coluna n FROM nome_da_tabela WHERE [condicao]
```

A condição é declarada através dos operadores disponíveis em *SQLite*. Pode-se, por exemplo, selecionar todas as linhas em que o conteúdo de uma coluna corresponda a um valor especificado através do operador "=". Outra possibilidade é filtrar dados de acordo com valores numéricos através dos operadores >, <, <= e >=. Pode-se combinar duas ou mais condições através dos operadores *AND* e *OR*.

O exemplo a seguir mostra como filtrar dados na tabela cadastro.

```
> SELECT * FROM cadastro WHERE cidade = "Sao Carlos"
Lucas|24|1234|Sao Carlos

> SELECT nome, cidade FROM cadastro WHERE idade > 18 AND idade < 22;
Ana|Guaratingueta
```

Modificando dados de uma tabela

O comando *UPDATE* é utilizado para modificar valores de uma tabela. Sua sintaxe é a seguinte:

```
UPDATE nome_da_tabela
SET coluna 1 = valor 1, ..., coluna n = valor n WHERE [condicao]
```

A condição é especificada da mesma maneira como no exemplo anterior. Todas as linhas selecionadas terão seus valores modificados.

Como exemplo, vamos atualizar na tabela cadastro a idade e a cidade de uma pessoa:

```
> SELECT * FROM cadastro WHERE nome = "Lucas";
Lucas|24|1234|Sao Carlos
```

```
> UPDATE cadastro
SET idade = 25, cidade = "Guaratingueta" WHERE nome = "Lucas";
> SELECT * FROM cadastro WHERE nome = "Lucas";
Lucas|25|1234|Guaratingueta
```

Listando tabelas

Não há um comando específico para listar todas as tabelas presentes em um arquivo. Nesse caso, a solução é recorrer a tabela *sqlite_master*. Ela está presente em todos os arquivos criados por *SQLite* e armazena o esquema completo do banco de dados. Essa tabela é criada automaticamente por *SQLite* através do comando:

```
CREATE TABLE sqlite_master(
  type text,
  name text,
  tbl_name text,
  rootpage integer,
  sql text
);
```

O nome de todas as tabelas no banco de dados pode ser obtido através do seguinte comando:

```
SELECT name FROM sqlite_master WHERE type = "table"
```

2.4 Python

Python [4] é uma linguagem de programação interpretada desenvolvida pela *Python Software Foundation*. Ela fornece suporte à programação orientada a objetos.

Atualmente, é uma linguagem de programação bastante versátil. Pode ser utilizada em diversos tipos de aplicações, graças a uma extensa lista de bibliotecas disponíveis. Podemos citar aplicações, como por exemplo, aplicações *Web*, cálculo científico e numérico, gráficos, *Machine Learning*, *Data Science*, visualização de dados, desenvolvimento de interfaces gráficas, entre outras.

Neste trabalho, foram utilizadas bibliotecas que permitem a criação de *threads*, o acesso a bancos de dados *sqlite*, o uso de expressões regulares e o *back-end* de um servidor. Essas bibliotecas e os seus respectivos usos serão discutidos nas próximas seções.

2.4.1 SQLite em Python

Python possui nativamente a biblioteca *sqlite3* [5] que permite a utilização de bancos de dados *sqlite*. É possível conectar-se a um banco de dados, executar comandos *SQL* e salvar alterações.

Conexão ao banco de dados e execução de comandos

Primeiramente, é necessário importar a biblioteca *sqlite3*. Em seguida, deve-se criar uma conexão com um banco de dados e criar um cursor para essa conexão a fim de executar comandos *SQL*. Isso é demonstrado no código a seguir:

```
import sqlite3
conn = sqlite3.connect("exemplo.db")          # cria uma conexao
cur = conn.cursor()                          # cria um cursor
```

Caso o arquivo utilizado com argumento da função *connect* não existir, ele será criado.

O método *execute*, do objeto cursor, é utilizado para executar um comando *SQL*. O método *executescript* permite a execução de mais de um comando por vez. Os comandos podem estar contidos em uma string, devendo ser terminados pelo caractere ";", ou podem estar em um arquivo de texto. O exemplo a seguir ilustra como os métodos *execute* e *executescript* podem ser utilizados.

```
# Execução de um comando utilizado-se "execute"
cur.execute(''CREATE TABLE example
            (col1 text, col2 real, col3 text)''')

# Execução de múltiplos comandos contidos em uma string
script = '''create table if not exists example_2 (
            id integer primary key autoincrement,
            col1 text,
            col2 real,
            col3 integer);
            insert into example_2 (col1, col2, col3) values ("A", 1.0, 2);
            insert into example_2 (col1, col2, col3) values ("B", 2.0, 3);
            insert into example_2 (col1, col2, col3) values ("C", 3.0, 4);
            '''
cur.executescript(script)

# Execução de múltiplos comandos a partir de um arquivo de texto
with open("script.db", "r") as f:
```

```
cur.execute_script(f.read())
```

Sempre que uma modificação é feita em um banco de dados, como por exemplo, a criação de uma nova tabela ou a inserção de uma linha, é necessário salvar essas alterações para que elas estejam disponíveis futuramente. Isso é realizado com o método *commit* do objeto conexão:

```
conn.commit()
```

É importante ressaltar que objetos do tipo *connection* e *cursor* podem apenas ser usados na *thread* em que foram criados. Caso seja necessário fazer uso de um banco de dados *sqlite* em uma *thread* que é executada em paralelo ao programa principal, a conexão com o banco de dados deve ser realizada nessa mesma *thread*.

Lendo dados do banco de dados

Para obter os dados de um banco de dados após a execução de um comando *SELECT*, há três possibilidades: utilizar o objeto cursor como um iterador, utilizar o método *fetchone()* ou utilizar o método *fetchall()*. O método *fetchone()* retorna apenas uma linha por vez, enquanto que *fetchall()* retorna todas as linhas disponíveis. Uma linha é representada por um *tuple*, em que cada elemento corresponde a uma coluna, na mesma sequência das colunas na tabela em questão ou na mesma sequência em que foram especificadas no comando *SELECT*. O comando *fetchall()*, pelo fato de poder retornar mais de uma linha, retorna sempre um objeto do tipo *list* que contém todas as linhas em questão.

No caso de usarmos um iterador, o código a seguir produz o seguinte resultado:

```
>>>cur.execute("select * from example_2")
>>>for row in cur:
    print row

(1, u'A', 1.0, 2)
(2, u'B', 2.0, 3)
(3, u'C', 3.0, 4)
```

No caso dos métodos *fetchone()* e *fetchall()*, temos:

```
# Exemplo com fetchone()
>>> cur.execute("select * from example_2")
>>> x = cur.fetchone()
>>> print x
```

```
(1, u'A', 1.0, 2)

# Exemplo com fetchall()
>>> cur.execute("select * from example_2")
>>> x = cur.fetchall()
>>> print x
[(1, u'A', 1.0, 2), (2, u'B', 2.0, 3), (3, u'C', 3.0, 4)]
```

Valores dos tipos *integer* e *real* em *sqlite3* são representados em *Python*, respectivamente, por variáveis do tipo *int* e *float*. Valores do tipo *text* no banco de dados correspondem à variáveis *unicode* em *Python*, podendo ser facilmente ser convertidas para *strings*, caso necessário.

Row objects

Um dos inconvenientes da representação das linhas de uma banco de dados como *tuples* é que não há um mapeamento direto entre um elemento de um *tuple* e a coluna a qual ele pertence. Elementos podem ser acessados apenas através de índices, e cabe ao programador determinar a relação entre os índices e as colunas de uma tabela. Isso requer maior esforço para a escrita do código, além de aumentar a probabilidade de erros.

Para contornar esse problema, podem ser utilizados objetos do tipo *Row*. Eles permitem mapear seus elementos através de índices, como já era feito com os *tuples*, ou através do nome das colunas, de forma similar a como é feito com dicionários em *Python*. Inclusive, há o método *keys()* que lista o nome das colunas de um objeto *Row*. Para utilizar objetos *Row* no lugar de *tuples*, deve-se proceder na seguinte maneira:

```
conn = sqlite3.connect("exemplo.db")
conn.row_factory = sqlite3.Row
cur = conn.cursor()
```

Os resultados desta alteração podem ser vistos a seguir:

```
>>> cur.execute("select * from example_2")
>>> x = cur.fetchone()
>>> print type(x)
<type 'sqlite3.Row'>
>>> print x.keys()
['id', 'col1', 'col2', 'col3']
>>> print x["col1"]
A
>>> print x[2]
```

1.0

Capítulo 3

Materiais e Métodos

3.1 Sistema embarcado

Foi desenvolvido neste projeto um sistema embarcado que é instalado em veículos leves com o objetivo de monitorá-los e enviar os dados coletados para um servidor. O projeto contemplou desde a escolha dos componentes a serem utilizados até o desenvolvimento do *software* que comanda o sistema embarcado.

O primeiro passo tomado foi definir as funcionalidades e as características que o sistema deveria possuir. Desejou-se desenvolver uma solução que funcionasse de maneira autônoma, sem a necessidade de intervenção humana. Uma vez instalado no veículo, o sistema deve sempre estar em operação. Por questões de segurança, o condutor não deve ter nenhum tipo de interação com os equipamentos instalados a fim de evitar distrações desnecessárias. Por se tratar de um sistema automotivo, decidiu-se que a alimentação deve ser fornecida pela bateria do veículo.

Dentro no contexto de gestão de frotas, foi estipulado que ao menos as seguintes informações devem ser coletadas: velocidade do veículo, rotação do motor, carga do motor, distância percorrida e geolocalização. Os três primeiros valores são utilizados para analisar como o veículo é conduzido. O monitoramento da velocidade pode indicar se os limites de velocidade são respeitados e se há acelerações e frenagens bruscas. A rotação do motor combinada com a velocidade pode mostrar se o veículo opera engrenado e se as marchas estão sendo utilizadas corretamente. Valores de rotação muito elevados que podem danificar o motor também seriam detectados. A carga do motor está associada ao consumo de combustível. A distância percorrida deve ser medida por ser fundamental para o agendamento de manutenções sem a necessidade de consultar o odômetro. Por fim, a geolocalização serve para registrar os traje-

tos realizados. Assim, é possível determinar se eles podem ser substituídos por outros mais curtos e se há desvios nas rotas estabelecidas.

O sistema embarcado deve coletar os dados de maneira contínua enquanto o veículo estiver em operação e armazená-los no cartão de memória da *Raspberry Pi*. É importante que seja possível definir o intervalo de amostragem de cada tipo de medição. No caso da velocidade, é fundamental medi-la em intervalos curtos de tempo, pois ela pode variar rapidamente. Por outro lado, ler a distância percorrida apenas uma vez por dia já seria o suficiente para o controle das manutenções.

As informações recolhidas devem ser posteriormente enviadas ao servidor que hospeda a plataforma de gestão de frotas. O envio dos dados deve ser feito de maneira automática através de uma conexão *Wi-Fi* no momento em que o veículo retornar a garagem. Isso garante que o frotista trabalhe com dados atualizados.

Após definir as características do sistema embarcado, iniciou-se o processo de escolha dos componentes a serem utilizados, para em seguida escrever o *software*.

3.2 *Hardware* do Sistema Embarcado

Basicamente, o sistema embarcado deve obter dois tipos de dados: geolocalização e dados do veículo.

Para a geolocalização, decidiu-se usar um módulo GPS destinado a sistemas embarcados. Este módulo, descrito na seção 3.2.3, permite determinar a posição do veículo e opera de acordo com o formato NMEA. Para seu funcionamento, é necessário apenas alimentá-lo e conectá-lo ao sistema embarcado. Os dados são transmitidos através de comunicação serial.

No caso dos dados sobre o veículo, é necessário medir pelo menos a velocidade, a rotação do motor e a distância percorrida. Esses valores são medidos em todos veículos que possuem sistema OBD e podem ser obtidos diretamente do computador de bordo, eliminando a necessidade de instalação de sensores. Optou-se por utilizar um adaptador OBD com comunicação *Bluetooth* descrito na seção 3.2.2. Trata-se de um interpretador multi-protocolo que funciona com todos os protocolos previstos no padrão OBD. Desta maneira, não é necessário o desenvolvimento de um *hardware* específico para a comunicação com o computador de bordo. O adaptador OBD também simplifica o programa do sistema embarcado, pois ele funciona da mesma maneira, independente do protocolo utilizado pelo veículo.

Para comandar o sistema embarcado foi escolhida a placa *Raspberry Pi 3 Model B*, des-

crita na seção 3.2.1, e que funciona com o sistema operacional *Linux*. Possui *bluetooth* e uma porta serial, podendo assim comunicar-se com o adaptador OBD e com o módulo GPS. Ela é responsável por ler os dados desses dois dispositivos, armazená-los e enviá-los para um servidor. O uso de *Linux* embarcado traz algumas vantagens em relação ao uso de um microcontrolador. É possível escrever o código usando diversas linguagens de programação como, por exemplo, *C++* e *Python*, assim como armazenar em bancos de dados as informações coletadas. Além disso, o sistema operacional permite a execução de vários processos em paralelo e possui as ferramentas necessárias para a conexão com a internet e para o uso de *bluetooth*.

3.2.1 Raspberry Pi

Para controlar o *hardware* embarcado, foi escolhido o computador *Raspberry Pi Model 3*, figura 3.1, pertencente à família *Raspberry Pi*. Desenvolvidos pela *Raspberry Pi Foundation* no Reino Unido, os dispositivos da família *Raspberry Pi* são computadores de baixo custo e do tamanho de um cartão de crédito. São constituídos de apenas uma placa, na qual todos os componentes estão montados. Por isso, são ótimas opções para sistemas embarcados.

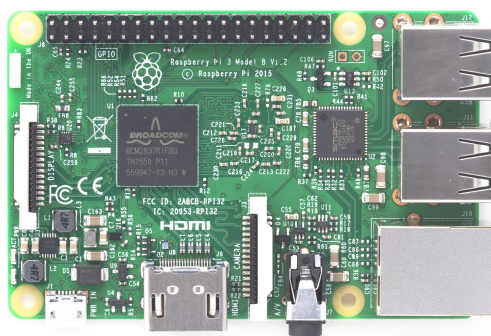


Figura 3.1: Raspberry Pi 3 Model B

A *Raspberry Pi 3 Model B*, lançada em fevereiro de 2016, é, no presente momento, o dispositivo mais recente de sua linha de produtos, e é comercializada pelo preço de 35 dólares americanos. Sua escolha para esse projeto justifica-se pelo fato de possuir os periféricos necessários para a comunicação com os outros elementos do projeto, e por funcionar com o sistema operacional *Linux*. O *Linux* permite, entre outras coisas, a escrita do código fonte na linguagem de programação *Python*, o uso de banco de dados, além da facilidade para a

conexão à *internet* e a outros computadores em rede.

Dentre as principais características da *Raspberry Pi 3 Model B*, pode-se destacar:

- processador ARMv8 *quad-core* com *clock* 1.2GHz;
- 1GB de memória *RAM*;
- bluetooth 4.1;
- Adaptador *wireless* 802.11n;
- 40 pinos de entrada/saída digitais;
- 4 portas USB;
- porta ethernet

A descrição dos pinos de entrada e de saída digitais é mostrada na figura 3.2. Há terminais de alimentação de 5V e de 3,3V. São também suportados os seguintes protocolos de comunicação digital: UART, SPI e I2C. A porta UART está disponível nos pinos GPIO 14 e GPIO 15. Ela é utilizada para a comunicação com o módulo GPS. É importante ressaltar que o nível lógico alto corresponde a 3,3V, e que níveis de tensão superiores a esse valor podem danificar a placa.

O módulo *Bluetooth* é utilizado para a comunicação com o adaptador OBD. O adaptador *wireless* permite que o sistema embarcado envie os dados automaticamente ao servidor, sem a necessidade de conexão de cabos ou de intervenção humana.

Neste projeto, a distribuição *Linux* utilizada é a *Raspbian*, que é baseada na distribuição *Debian*.

3.2.2 Adaptador OBD

O adaptador OBD, figura 3.3, é o dispositivo utilizado para a comunicação com o computador de bordo do carro. Ele é instalado diretamente na porta OBD do veículo, conforme mostrado na figura 3.4. Visto que a porta OBD disponibiliza dois terminais de alimentação provenientes da bateria, não há necessidade de alimentação externa.

A troca de dados entre o adaptador e a *Raspberry Pi* é realizada através de *Bluetooth*. O adaptador utilizado é baseado no circuito integrado ELM327, produzido pela empresa canadense *ELM electronics*. O ELM327 é um interpretador multi-protocolo projetado para

Raspberry Pi 3 GPIO Header				
Pin#	NAME		NAME	Pin#
01	3.3v DC Power	⬇	DC Power 5v	02
03	GPIO02 (SDA1 , I ² C)	⬇	DC Power 5v	04
05	GPIO03 (SCL1 , I ² C)	⬇	Ground	06
07	GPIO04 (GPIO_GCLK)	⬇	(TXD0) GPIO14	08
09	Ground	⬇	(RXD0) GPIO15	10
11	GPIO17 (GPIO_GEN0)	⬇	(GPIO_GEN1) GPIO18	12
13	GPIO27 (GPIO_GEN2)	⬇	Ground	14
15	GPIO22 (GPIO_GEN3)	⬇	(GPIO_GEN4) GPIO23	16
17	3.3v DC Power	⬇	(GPIO_GEN5) GPIO24	18
19	GPIO10 (SPI_MOSI)	⬇	Ground	20
21	GPIO09 (SPI_MISO)	⬇	(GPIO_GEN6) GPIO25	22
23	GPIO11 (SPI_CLK)	⬇	(SPI_CE0_N) GPIO08	24
25	Ground	⬇	(SPI_CE1_N) GPIO07	26
27	ID_SD (I ² C ID EEPROM)	⬇	(I ² C ID EEPROM) ID_SC	28
29	GPIO05	⬇	Ground	30
31	GPIO06	⬇	GPIO12	32
33	GPIO13	⬇	Ground	34
35	GPIO19	⬇	GPIO16	36
37	GPIO26	⬇	GPIO20	38
39	Ground	⬇	GPIO21	40

Rev. 2
29/02/2016

www.element14.com/RaspberryPi

Figura 3.2: Descrição dos pinos de entrada e saída da *Raspberry Pi 3 Model B*

funcionar com todos os protocolos automotivos de veículos de passeio previstos na especificação OBD-II. O público alvo deste circuito integrado são hobbistas ou pequenas empresas que desenvolvem projetos que requerem comunicação com o computador de bordo de um veículo, mas que desejam abstrair as especificidades dos diferentes protocolos automotivos existentes.



Figura 3.3: Adaptadores OBD utilizados no projeto

O adaptador OBD utilizado já possui todos os componentes externos necessários para a utilização do ELM327, além de possibilitar a conexão através de *Bluetooth*, o que reduz significativamente o custo e o tempo de execução deste projeto. Um dispositivo como esse pode ser adquirido no mercado brasileiro por preços a partir de 30 reais.



Figura 3.4: Adaptador OBD instalado no veículo

ELM327

O circuito integrado ELM327 [6] funciona com os seguintes protocolos automotivos:

- SAE J1850-PWM;
- SAE J1850-VPW;
- ISO 9141-2;
- ISO 14230-4;
- ISO 15765-4;
- SAE J1939.

Esses protocolos cobrem todos os veículos que atendem à especificação OBD-II. Além disso, é possível sua utilização em caminhões e em outros veículos pesados que empregam o protocolo SAE J1939.

O ELM327 atua como uma ponte entre os diferentes protocolos automotivos e a porta de comunicação serial RS232. Numa aplicação típica, um microcontrolador ou sistema embarcado envia um comando ao ELM327 através da porta serial, e esse comando é repassado ao computador de bordo utilizando-se o protocolo automotivo pertinente.

Destaca-se, em particular, a capacidade deste circuito integrado em detectar automaticamente o protocolo automotivo utilizado pelo veículo, o que facilita o desenvolvimento de aplicações com o mesmo. Outra característica importante é o fato de possuir um conversor analógico-digital que permite medir o nível de tensão da bateria, independente desse dado ser fornecido pelo computador de bordo do veículo. O ELM327 permite algumas configurações,

que são efetuadas através de comandos específicos (*AT commands*) listados no *datasheet* pelo fabricante.

Comunicação com o ELM327

Para a comunicação com o ELM327, é utilizada a porta serial RS232. A taxa de envio de dados pode ser configurada em 9600bps ou 38400bps através do pino *Baud Rate*. Além disso, a comunicação serial utiliza 8 bits de dados, 1 stop bit e nenhum bit de paridade.

Os comandos enviados ao ELM327 podem ser códigos OBD, que são repassados ao veículo, ou instruções destinadas ao próprio circuito integrado. Todos os comandos enviados ao ELM327 devem ser terminados por um *carriage return* (código ASCII 0x0D), caso contrário, ele não será considerado. O ELM237 não diferencia letras maiúsculas de minúsculas e espaços são desconsiderados. No caso de códigos OBD, eles devem ser enviados na forma de caracteres ASCII, e apenas caracteres hexadecimais são permitidos. Instruções destinadas ao ELM327 sempre começam com os caracteres "AT". Respostas fornecidas pelo ELM327 sempre são terminadas com um *carriage return*.

Após terminar de processar um comando e enviar uma resposta, se for o caso, o ELM327 envia o caractere '>' indicando que está ocioso e pronto para receber novos comandos. Por exemplo, caso seja enviado o comando "AT I", que solicita a versão do circuito integrado, a resposta fornecida será:

```
ELM327 v2.1
```

```
>
```

No caso de um código OBD, caso seja feita uma leitura da rotação do motor, correspondente ao código 01 0C, a resposta será:

```
41 0C 00 00
```

```
>
```

Configurando e utilizando o ELM327

Antes de enviar códigos OBD direcionados ao computador de bordo, é necessário inicializar o ELM327 para que ele possa utilizar um protocolo automotivo. Primeiramente, é necessário informar ao circuito integrado qual protocolo deverá ser utilizado. Isso pode ser realizado através do comando "AT SP 0", que define o protocolo a ser utilizado. As letras *SP*,

em inglês, significam *set protocol*, ou, define protocolo. O número 0, usado como argumento, indica ao circuito integrado que ele deve descobrir automaticamente qual protocolo é utilizado pelo veículo. Isso é particularmente útil para se desenvolver uma solução que funcione em diferentes modelos de veículos.

A busca pelos protocolos disponíveis ocorre apenas quando o primeiro código OBD é transmitido ao ELM327. Inicia-se então uma varredura e o circuito integrado envia a mensagem "SEARCHING...". Esse processo pode levar alguns segundos. Uma vez definido qual protocolo será empregado, o código OBD é enviado e o ELM327 retorna a resposta do computador de bordo.

O usuário também pode escolher qual protocolo deve ser utilizado. Por exemplo, o comando "AT SP 1" seleciona o protocolo SAE J1850 PWM, enquanto que o comando "AT SP 3" seleciona o protocolo ISO 9141-2. Para uma lista completa das opções disponíveis, ver a referência [6]. Caso o usuário escolha um protocolo que não é suportado pelo veículo, os comandos OBD não funcionarão e o circuito integrado retornará a mensagem "NO DATA".

No caso do sistema elétrico do veículo estar desligado, na maioria dos veículos, o computador de bordo não responderá a nenhum código OBD. Portanto, nesse caso, independente da opção utilizada no comando "AT SP X", o ELM327 sempre retornará "NO DATA".

Após feita a inicialização do protocolo automotivo no ELM327, ele está pronto para enviar qualquer código OBD. Sempre que o circuito integrado for desligado, o que no caso do adaptador OBD ocorre quando o veículo é desligado, é necessário inicializar o protocolo novamente. É importante salientar que este circuito integrado é apenas uma ponte entre os protocolos automotivos e a porta serial. Qualquer comando que não seja iniciado por "AT" e que contenha apenas caracteres hexadecimais é repassado diretamente ao computador de bordo. Nenhum tipo de verificação é feita, e cabe ao usuário certificar-se que os códigos passados estão corretos.

3.2.3 Módulo GPS

Foi utilizado o módulo GPS GY-NEO6VM2, figura 3.5, para a determinação da posição dos veículos. Este módulo utiliza o circuito integrado NEO-6M-0-001, do fabricante suíço *U-blox*, e é capaz de fornecer sua localização a uma taxa de 1Hz.

A comunicação com este módulo é feita através da porta serial RS232 a uma taxa de 9600 *bits* por segundo. O módulo é capaz de fornecer uma nova leitura de posição a cada 1 segundo. A tensão de alimentação deve estar compreendida na faixa de 3V a 5V. O nível

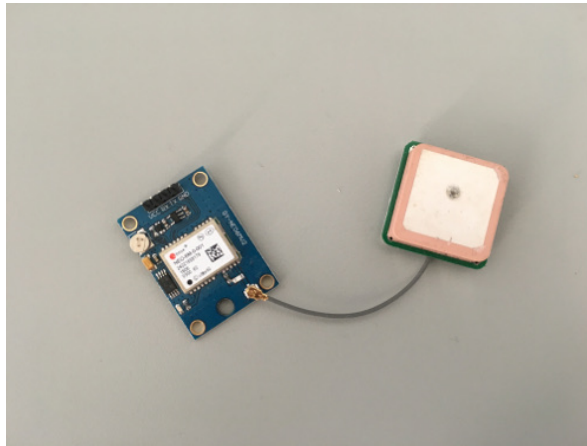


Figura 3.5: Módulo GPS utilizado

lógico utilizado pelo circuito integrado é de 3,3V. Os dados enviados pelo módulo seguem a especificação *NMEA*, conforme mostrado na seção 2.2.1.

3.3 *Software* do Sistema Embarcado

Optou-se por desenvolver o código do sistema embarcado através da linguagem de programação *Python*. É uma linguagem de alto nível e que oferece suporte a programação orientada a objetos. Além disso, há um grande número de bibliotecas disponíveis para aplicações como, por exemplo, utilização com bancos de dados, envio de arquivos por *HTTP* e execução de processos em paralelo.

O programa desenvolvido para o sistema embarcado foi dividido em quatro partes. Elas são: leitura da geolocalização através do módulo GPS, leitura dos dados do veículo através da interface OBDII, envio dos dados coletados para o servidor, e por fim, um programa para integrar esses três blocos.

A comunicação com o GPS é feita por um programa chamado de "*GPS_serial*" e a leitura dos dados do veículo pelo programa "*OBDMonitor*". Ambos armazenam os dados coletados em bancos de dados *SQLite*. O programa "*UploadManager*" controla o envio dos dados ao servidor. O programa "*VehicleMonitor*" recebe as configurações do usuário e é responsável por executar os três programas citados anteriormente. A estrutura do código do sistema embarcado é mostrado na figura 3.6.

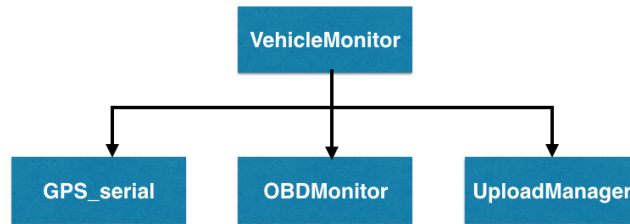


Figura 3.6: Organização do *software* do sistema embarcado

3.3.1 Threading

Os programas *OBDMonitor* e *GPS_serial* foram escritos na forma de classes. Visto que é necessário coletar dados de geolocalização e do veículos de maneira simultânea, eles precisam ser executados em paralelo. Essa seção discute como isso pode ser realizado.

Python possui a biblioteca *threading* [7], que permite a execução de processos, ou *threads*, de maneira concorrente. O exemplo a seguir mostra como utilizar essa biblioteca para executar uma função em paralelo.

```

import threading
import time

def my_function(id):
    count = 1
    while count <= 3:
        time.sleep(1)
        print id + ": " + str(count) + " seconds"
        count += 1

t1 = threading.Thread(target = my_function, args = ("Thread 1",))
t1.start()
t2 = threading.Thread(target = my_function, args = ("Thread 2",))
t2.start()
  
```

Uma *thread* é iniciada ao criar-se um objeto da classe *threading.Thread*, onde é definida a função que será executada e os argumentos que serão passados. A execução é iniciada com o método *start()*. O resultado deste exemplo é mostrado abaixo:

```

Thread 1: 1 seconds
Thread 2: 1 seconds
Thread 1: 2 seconds
  Thread 2: 2 seconds
Thread 1: 3 seconds
  
```

```
Thread 2: 3 seconds
```

Também é possível executar classes em paralelo. Isso é feito ao criar subclasses da classe *Thread*. Nessa classe, o conteúdo do método *run()* é executado em paralelo. O exemplo a seguir mostra como criar uma subclasse a partir da classe *Thread*.

```
import threading
import time

class my_class(threading.Thread):
    def __init__(self, id):
        self.id = id
        threading.Thread.__init__(self, group = None, \
                                   target = None, name = None, verbose = None)
        return

    def run(self):
        count = 1
        while count <= 3:
            time.sleep(1)
            print self.id + ": " + str(count) + " seconds"
            count += 1

t1 = my_class("Thread 1")
t1.start()
t2 = my_class("Thread 2")
t2.start()
```

É necessário realizar a inicialização da classe *Thread* no método "*__init__*". A execução da *thread* é iniciada através do método *run()*, da mesma maneira como no exemplo anterior. Este exemplo produz o seguinte resultado:

```
Thread 1: 1 secondsThread 2: 1 seconds

Thread 2: 2 seconds
Thread 1: 2 seconds
Thread 2: 3 seconds
Thread 1: 3 seconds
```

Essa técnica foi empregada nas classes *OBDMonitor* e *GPS_serial*.

3.3.2 *GPS_serial*

A classe *GPS_serial* é responsável por obter dados de geolocalização fornecidos pelo módulo GPS. A comunicação entre este módulo e a *Raspberry Pi* ocorre por meio de comunicação serial.

Comunicação serial

Para a leitura dos dados da porta serial, foi utilizada a biblioteca *pySerial* [8]. Esta biblioteca não está incluída na instalação de *Python* e precisa ser instalada separadamente. Isso pode ser feito, entre outras maneiras, com o auxílio do gerenciador de pacotes *pip*, através do seguinte comando:

```
sudo pip install pyserial
```

No caso da *Raspberry Pi 3*, a porta serial é disponibilizada em `"/dev/ttyS0"`. A "abertura" de uma porta serial na biblioteca *pySerial*, comunicando-se a *9600 bits* por segundo, é feita da seguinte maneira:

```
import serial
port = serial.Serial("/dev/ttyS0", 9600, timeout = 0.1)
```

O argumento *timeout* corresponde ao tempo máximo que deve ser esperado para a leitura de uma linha ou de um caractere.

A leitura de uma caractere é feita através do método *read* e a leitura de uma linha completa (terminada pelo caractere ASCII `"\n"`) através do método *readline()*.

```
c = port.read()           # le um caractere
s = port.read(10)        # le ate 10 caracteres
t = port.readline()      # le uma linha
```

Funcionamento de *GPS_serial*

A classe *GPS_serial* monitora de maneira contínua todos os dados enviados pelo módulo GPS. Valores de latitude, longitude e altitude são extraídos diretamente da sentença *GGA*, seção 2.2.1, e armazenados em um banco de dados, juntamente com o horário e a data em que foram lidos.

Mostra-se a seguir um exemplo dos dados que são continuamente enviados pelo módulo GPS:

```

$GPGSV,4,1,13,05,08,141,31,10,22,335,25,12,13,027,23,13,03,102,23*70
$GPGSV,4,2,13,15,19,075,21,16,02,215,27,18,54,345,21,20,58,125,32*77
$GPGSV,4,3,13,21,55,217,33,25,49,011,10,26,29,225,29,29,50,126,25*7D
$GPGSV,4,4,13,31,15,284,21*40
$GPGLL,2200.67129,S,04754.97951,W,030250.00,A,A*68
$GPGST,030250.00,35,,,,6.4,5.7,10*7A
$GPZDA,030250.00,15,09,2016,00,00*6A
$GPRMC,030251.00,A,2200.67146,S,04754.97953,W,0.039,,150916,,,A*7B
$GPVTG,,T,,M,0.039,N,0.072,K,A*2C
$GPGGA,030251.00,2200.67146,S,04754.97953,W,1,10,0.79,834.4,M,-6.1,M,,*49
$GPGSA,A,3,20,21,18,29,26,10,15,31,12,05,,,1.46,0.79,1.23*09

```

A sentença *GGA* indica que a leitura foi realizada às 03:02:51, e que a latitude corresponde a $-22^{\circ} 00,67146'$, a longitude a $-47^{\circ} 54,97953'$ e a altitude a 834,4 metros. O horário da leitura é obtido através do horário fornecido pelos satélites GPS.

Utilização

A classe *GPS_serial* toma como argumentos a porta serial, velocidade da porta serial (*baud rate*), *timeout* da porta serial e o arquivo de banco de dados a ser utilizado. A leitura dos dados é iniciada através do método *start()*, conforme discutido na seção 3.3.1. O exemplo a seguir mostra como ela é utilizada neste projeto:

```

gps = GPS_serial("/dev/ttyS0", 9600, 0.1, "some_file.db")
gps.start()           # inicia leitura dos dados

```

3.3.3 OBDMonitor

A classe *OBDMonitor* é responsável pela leitura de dados do veículo através da interface OBD. A comunicação entre o adaptador OBD e a *Raspberry Pi* é feita através de *Bluetooth*.

Comunicação por *Bluetooth*

A biblioteca *PyBluez* foi utilizada para permitir a comunicação por *bluetooth* com o adaptador OBD [9]. Ela oferece suporte aos protocolos *RFCOMM* e *L2CAP*. A troca de dados é feita através de *sockets*. Da mesma forma como a biblioteca *pySerial*, ela não faz parte da instalação de *Python* e precisa ser instalada separadamente. Utilizando-se o gerenciador de pacotes *pip*, isso pode ser feito da seguinte maneira:

```

sudo pip install pybluez

```

O código a seguir mostra como é estabelecida a conexão com outro dispositivo *bluetooth* a partir do seu endereço. O adaptador OBD utiliza o protocolo *RFCOMM*.

```
import bluetooth
sock = bluetooth.BluetoothSocket(bluetooth.RFCOMM)
baddr = "00:1D:A5:68:98:8D" # bluetooth address
port = 1
sock.connect((baddr, port))
```

Uma vez criado o *socket* a ser utilizado, é possível enviar e receber dados através dos métodos *send* e *recv*, respectivamente.

```
sock.recv(1) # recebe 1 caractere
sock.recv(10) # recebe 10 caracteres
sock.send("ola!!!") # envia uma mensagem
```

Funcionamento

A comunicação com o adaptador OBD é realizada conforme foi descrito na seção 3.2.2. A inicialização do ELM327, seção 3.2.2, é feita automaticamente. São enviados de maneira periódica os códigos OBD correspondentes aos sensores a serem lidos. Os dados coletados são armazenados em um banco de dados. É possível configurar individualmente o período de leitura de cada sensor.

Utilização

A classe *OBDMonitor* toma como argumentos o endereço *bluetooth* do adaptador OBD, uma lista com dados dos sensores a serem lidos, o arquivo do banco de dados e o tempo de *timeout*. O exemplo a seguir mostra como ela deve ser utilizada.

```
baddr = "00:1D:A5:68:98:8D"
sensors_read = [
    {"sensors": [ "Speed",
                  "Engine RPM",
                ],
      "period": 0
    },
    {"sensors": [ "Engine coolant temperature" ],
      "period": 15
    },
    {"sensors": [ "Distance traveled malfunction",
```

```

        "Distance traveled codes cleared",
    ],
    "period": 120
},
]
obd_monitor = OBDMonitor(baddr, sensors_read, "some_file.db", 0.1)
obd_monitor.start()

```

O período é definido em segundos, sendo que um período igual a zero indica que os sensores devem ser lidos tão rapidamente quanto for possível.

3.3.4 *UploadManager*

A classe *UploadManager* é responsável pelo envio dos dados coletados pelo sistema embarcado ao servidor. Ela é informada pelo *VehicleMonitor* quando novos dados são acrescentados aos bancos de dados. O envio dos dados é realizado apenas quando o veículo retorna a empresa e o sistema embarcado possui acesso a uma rede *WiFi*.

Envio de arquivos através de *HTTP*

A biblioteca *Requests* permite a utilização do protocolo *HTTP* em *Python* de maneira simples [10]. Ela foi empregada neste projeto com o objetivo de enviar arquivos do sistema embarcado ao servidor.

Sua instalação pode ser feita através do seguinte comando:

```
pip install requests
```

O exemplo a seguir demonstra como enviar um arquivo a um endereço *http*.

```

import requests
url = "http://192.168.0.111:5000/upload"
files = {"file": open("obd_data.db", "rb")}
r = requests.post(url, files = files)

```

3.3.5 *VehicleMonitor*

A classe *VehicleMonitor* controla todo o sistema embarcado. Ela é executada após a inicialização do sistema operacional. Ela inicia e controla a execução dos três blocos discutidos anteriormente: *GPS_serial*, *OBDMonitor* e *UploadManager*.

Ela é responsável por monitorar as conexões *WiFi*. Caso seja detectada uma rede *WiFi* disponível na garagem do veículo, a aquisição de dados é interrompida e é feito o envio dos dados coletados ao servidor. A partir do momento que o veículo é ligado novamente, a aquisição de dados é reiniciada.

Optou-se por criar um arquivo de banco de dados para cada dia, de forma a evitar que o sistema embarcado trabalhe com arquivos muito grandes e para facilitar o envio dos dados ao servidor. Isso também é controlado pela classe *VehicleMonitor*.

3.4 Plataforma Web

A plataforma *web* é utilizada para realizar a gestão de uma frota. Ela fornece informações sobre todos os veículos e permite visualizar os dados coletados pelo sistema embarcado. Permite ainda realizar o agendamentos de manutenções. É a principal ferramenta de que o frotista dispõe para gerenciar seus veículos.

A plataforma *web* foi desenvolvida usando quatro linguagens de programação: *HTML*, *CSS*, *JavaScript* e *Python*. As três primeiras são empregadas na parte de *front-end*, ou seja, o código que é executado pelo navegador de *internet* do usuário. O *HTML* define a estrutura de uma página *web*, enquanto que o *CSS* é utilizado para definir o estilo, isto é, tipo e tamanho de fonte, cores, formato dos elementos da página, entre outros. Já o *javascript* é responsável por alterar o conteúdo de uma página, requisitar dados do servidor e gerar gráficos, como será discutido nas próximas seções.

O *Python* foi empregado na parte de *back-end* da aplicação, isto é, o código que é executado no servidor. Ele fornece uma página *web* a um usuário quando solicitado, além de receber e armazenar os dados dos sistemas embarcados. Foi utilizada a biblioteca *Flask*, que é específica para a criação de aplicações *web*. Ela é discutida em mais detalhes na seção 3.4.1. Como no caso do sistema embarcado, *SQLite* foi escolhido como solução de banco de dados.

3.4.1 Flask

Flask é uma biblioteca escrita em *Python* destinada ao desenvolvimento de aplicações *web* [11]. Ela é baseada em outras duas bibliotecas: *Jinja2* [12], utilizada para a criação de *templates*, e *Werkzeug* [13], responsável pela interface com servidores.

Trata-se de uma biblioteca de fácil utilização e que permite o uso de todos os recursos

disponíveis em *Python*. É uma alternativa a biblioteca *Django*, também escrita em *Python*, mas com a vantagem de ser mais simples. Possui um servidor que é destinado a testes durante o processo de desenvolvimento. Entretanto, ele não é recomendado para produção pelo fato de não ser escalável e por servir apenas uma página por vez .

Exemplo de aplicação

O exemplo a seguir mostra como utilizar esta biblioteca para servir uma página *web* e como utilizar *templates* para customizar páginas.

```

from flask import Flask
from flask import render_template
app = Flask(__name__)

@app.route('/hello_world')
def hello_world():
    titulo = "Meu Titulo"
    lista1 = ["item 1", "item 2", "item 3"]
    lista2 = None

    return render_template("hello.html", titulo = titulo ,
                           lista1 = lista1)

```

O conjunto de um comando *app.route()* e da função que lhe está associada é chamado de *view* e define o comportamento do servidor para um determinado endereço. Para testar esse exemplo de aplicação, é necessário inicializar o servidor de testes do *Flask*. Isto é feito através dos seguintes comandos no terminal:

```

$ export FLASK_APP=exemplo.py
$ python -m run flask run
* Running on http://127.0.0.1:5000/

```

O comando *@app.route('hello_world')* define que a página será servida no endereço *http://127.0.0.1:5000/hello_world*. Quando o navegador solicita esta página, o servidor executa a função *hello_world* que renderiza o arquivo *hello.html* utilizando a biblioteca *Jinja2* através da função *render_template*. O conteúdo de *hello.html* é mostrado a seguir:

```

<h1>{{ titulo }}</h1>
<h3>Lista:</h3>
<ul>
    {% for item in lista1 %}

```

```

        <li>{{ item }}</li>
    {% endfor %}
</ul>

```

Na renderização, as variáveis passadas como argumentos na função `render_template` podem ser utilizadas, e é possível, inclusive, utilizar estruturas de controle como, por exemplo, laços do tipo *for*. A documentação da biblioteca *Jinja2* possui um guia completo da sintaxe permitida, que é similar a sintaxe de *Python*. O processo deve ser repetido caso seja necessário servir mais páginas web. O resultado pode ser visto na figura 3.7.



Figura 3.7: Página web gerada por *Flask* com o uso de um *template*

Fornecendo dados no formato JSON

Quando um endereço é solicitado ao servidor, é possível fornecer, ao invés de uma página em HTML, dados no formato JSON. Isso é possível graças a função `jsonify`. Ela é importada da seguinte maneira:

```
from flask.json import jsonify
```

Sua utilização é feita da seguinte maneira:

```

@app.route('/json_data')
def send_json_data():
    veiculos = [
        { "modelo" : "Celta",
          "fabricante" : "Chevrolet",
          "cor" : "prata"
        },
        { "modelo" : "Gol",
          "fabricante" : "VW",
          "cor" : "preto"
        },
    ]

```

```

]
motoristas = [
    "Lucas",
    "Ana"
]

return jsonify(veiculos = veiculos, motoristas = motoristas)

```

Pode-se notar que a função `render_template` foi substituída por `jsonify`. O resultado é mostrado na figura 3.8. Este método foi utilizado neste projeto para fornecer informações solicitadas pelo navegador de *internet* através de *XMLHttpRequest*, conforme discutido na seção 3.4.5.



```

{
  "motoristas": [
    "Lucas",
    "Ana"
  ],
  "veiculos": [
    {
      "cor": "prata",
      "fabricante": "Chevrolet",
      "modelo": "Celta"
    },
    {
      "cor": "preto",
      "fabricante": "VW",
      "modelo": "Gol"
    }
  ]
}

```

Figura 3.8: Exemplo da utilização da função `jsonify` em *Flask*

3.4.2 Upload de arquivos

Flask também oferece suporte para receber arquivos enviados através de *http requests* e salvá-los apropriadamente. Essa é a maneira como o sistema embarcado envia para o servidor os dados que foram coletados, conforme discutido na seção 3.3.4.

É necessário importar o objeto *request* incluído na biblioteca *Flask*:

```

from flask import request

```

O trecho a seguir mostra como criar um *view* que recebe um arquivo:

```

@app.route("upload", methods = ["GET", "POST"])
def get_file():
    if request.method == "POST":

```

```
f = request.files["file"]
f.save("path_to_file")
```

Os arquivos são enviados através de *requests* do tipo *POST*, o que deve ser verificado. Os arquivos são temporariamente salvos por *Flask* e ficam disponíveis através do método *files* do objeto *requests*. O método *save* permite armazená-los de maneira definitiva no servidor.

3.4.3 Gráficos

A biblioteca "*morris.js*" [14] foi utilizada para gerar os gráficos mostrados pela plataforma *web*. Ela é escrita em *Javascript* e oferece suporte para quatro tipos de gráficos: de linha, de área, de barras e de *pizza*. A proposta desta biblioteca é fornecer uma solução de fácil utilização, porém que produza gráficos com aparência profissional.

O exemplo a seguir mostra como proceder para gerar um gráfico de linhas. O procedimento é similar para os outros tipos de gráficos. Inicialmente, é necessário instalar e importar a biblioteca *morris.js* e as suas dependências, *jQuery* e *Raphaël*:

```
<link rel="stylesheet" href="//morris.css">
<script src="jquery.js"></script>
<script src="raphael.js"></script>
<script src="morris.js"></script>
```

Em seguida, posiciona-se o gráfico na página *HTML* através de um elemento do tipo *div*. Ele deve ser devidamente identificado através de um "*id*". As dimensões do gráfico correspondem as dimensões deste elemento.

```
<div id="meu-grafico" style="width: 600px; height:400px"></div>
```

O gráfico é então declarado no código *javascript*, onde também são informados os pontos a serem exibidos e as legendas. A biblioteca *morris.js* tem suporte para datas e horários, o que facilita a realização de gráficos baseados em séries temporais. A lista de pontos a serem exibidos é declarada em formato compatível com o *JSON*. Neste caso, o exemplo é feito com valores de velocidade. O resultado obtido é mostrado na figura 3.9. Quando o cursor do *mouse* passa por cima de um ponto, seu valor é mostrado ao usuário.

```
new Morris.Line({
  element: "meu-grafico",
  data : [
    { timestamp: "2016-11-05 10:00:00", speed: 10 },
    { timestamp: "2016-11-05 10:00:05", speed: 30 },
```

```

        { timestamp: "2016-11-05 10:00:10", speed: 37 },
        { timestamp: "2016-11-05 10:00:15", speed: 40 },
        { timestamp: "2016-11-05 10:00:20", speed: 30 },
        { timestamp: "2016-11-05 10:00:25", speed: 22 },
        { timestamp: "2016-11-05 10:00:30", speed: 28 },
    ],
    xkey: "timestamp",
    ykeys: ["speed"],
    labels: ["velocidade"],
    postUnits : " km/h"
});

```

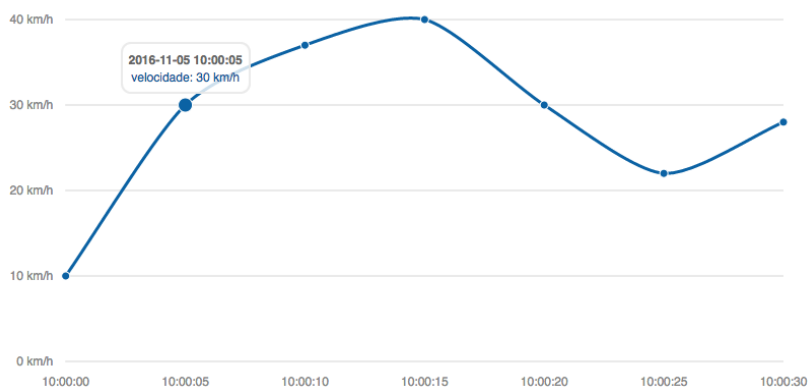


Figura 3.9: Exemplo de gráfico de velocidade produzido com a biblioteca *Morris.js* para *JavaScript*

3.4.4 Google Maps API

O *Google Maps* disponibiliza uma API que permite que desenvolvedores incluam mapas em *websites* e em aplicativos para *smartphones* [15]. Entre os muitos recursos disponíveis, é possível adicionar marcadores para indicar a localização de um ponto de interesse e incluir linhas para mostrar um trajeto.

Neste projeto, foi empregada a API para *JavaScript* para mostrar na plataforma *web* os trajetos realizados pelos veículos. O exemplo a seguir ilustra como criar um mapa, adicionar um marcador e incluir um trajeto. Para posicionar o mapa na página, deve-se criar um elemento do tipo *div* e atribuir a ele um *id*. As dimensões deste elemento serão as dimensões do mapa.

```
<div id="meu-mapa" style="width: 800px; height: 400px;"></div>
```

Todo o restante é feito em *JavaScript*. Primeiramente, cria-se um mapa especificando a sua coordenada central e o *zoom*:

```
// Cria um Mapa
var mapCanvas = document.getElementById("meu-mapa");
var mapOptions = {
    center: new google.maps.LatLng(-22.005639, -47.898364),
    zoom: 17
}
var map = new google.maps.Map(mapCanvas, mapOptions);
```

Para adicionar um marcador, deve-se indicar sua posição e em qual mapa ele deverá ser exibido. Neste caso, será no mapa chamado de *map*.

```
// Posiciona um marcador
var sel = { lat: -22.006103, lng: -47.897861};
var marker = new google.maps.Marker({
    position: sel,
    map: map,
});
```

No caso de um trajeto, seus pontos são listados em uma variável do tipo *javascript array*. O trajeto é representado pela variável tipo *Polyline* da *API*. Entre as opções da *polyline*, pode-se, por exemplo, definir a cor, a largura e a opacidade da linha (*strokeColor*, *strokeWeight* e *strokeOpacity*).

```
var pathCoordinates = [
    { lat: -22.006693, lng: -47.897725 },
    { lat: -22.006900, lng: -47.897763 },
    { lat: -22.006992, lng: -47.897911 },
    { lat: -22.007012, lng: -47.898779 },
    { lat: -22.006918, lng: -47.898894 },
    { lat: -22.006055, lng: -47.899012 },
    { lat: -22.004670, lng: -47.898899 },
    { lat: -22.004430, lng: -47.899406 },
    { lat: -22.004535, lng: -47.899717 },
    { lat: -22.004773, lng: -47.899722 },
    { lat: -22.004863, lng: -47.899534 },
    { lat: -22.005022, lng: -47.899486 },
    { lat: -22.005286, lng: -47.899792 },
    { lat: -22.005052, lng: -47.900173 },
];
```

```

var path = new google.maps.Polyline({
  path: pathCoordinates,
  strokeColor: "#0000FF",
  strokeOpacity: 0.8,
  strokeWeight: 4,
});
path.setMap(map);

```

O resultado do mapa com um marcador e um trajeto pode ser visto na figura 3.10

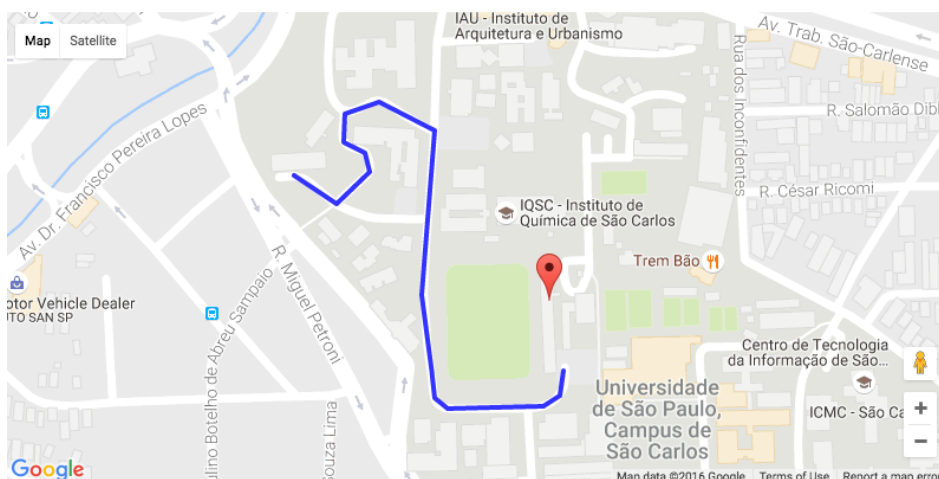


Figura 3.10: Exemplo do uso da API do *Google Maps* para *JavaScript*

3.4.5 Troca de dados entre o servidor e *JavaScript*

O objeto *XMLHttpRequest* é uma biblioteca para *JavaScript* que permite a troca de informações entre um navegador de *internet* e um servidor [16]. Ele é particularmente útil para atualizar os dados mostrados em uma página sem a necessidade de recarregá-la. Também pode ser utilizado para que o navegador envie dados ao servidor. Apesar do nome, é possível trocar dados nos formatos XML, JSON, HTML ou em texto.

Neste projeto, o *XMLHttpRequest* foi utilizado para enviar ao navegador de *internet* dados sobre um veículo que devem ser exibidos através de mapas ou gráficos como, por exemplo, o trajeto percorrido e a velocidade. Optou-se por essa solução pelo fato de não ser necessário incluir os pontos no código HTML de cada veículo e pela possibilidade de modificar o que é exibido sem a necessidade de recarregar a página.

Para possibilitar o envio desses dados, foi necessário intervir no código *JavaScript* e no servidor em *Flask*. No código do servidor, foi disponibilizado um endereço *url* exclusivamente para esse propósito. Ao receber uma requisição, o servidor *Flask* retorna os dados em

formato *JSON* utilizando o método *jsonify*.

O código *JavaScript* é responsável por fazer a requisição ao servidor. O trecho a seguir mostra como isso pode ser feito:

```
var xmlhttp = new XMLHttpRequest();
var url = "http://0.0.0.0:5000/data_report/gps";

xmlhttp.onreadystatechange = function () {
    if (this.readyState == 4 && this.status == 200) {
        var myData = JSON.parse(this.responseText);
        someFunction(myData);
    }
};

xmlhttp.open("GET", url, true);
xmlhttp.send();
```

Primeiramente, é criada a variável *xmlhttp*, que requisitará os dados ao endereço contido na variável *url* por meio dos métodos *open* e *send*. O método *onreadystatechange* contém o código que deve ser executado quando o servidor responder com os dados solicitados. Caso os dados sejam recebidos corretamente, uma função é chamada. Aqui, a título de exemplo, chama-se a função *someFunction*.

Capítulo 4

Resultados

4.1 Sistema embarcado

O sistema embarcado desenvolvido cumpriu suas funções de coletar dados sobre a posição do veículo e dados fornecidos pela interface OBD. As figuras 4.1 e 4.2 mostram os dados obtidos pelo sistema embarcado, e que estão armazenados em bancos de dados *SQLite*.

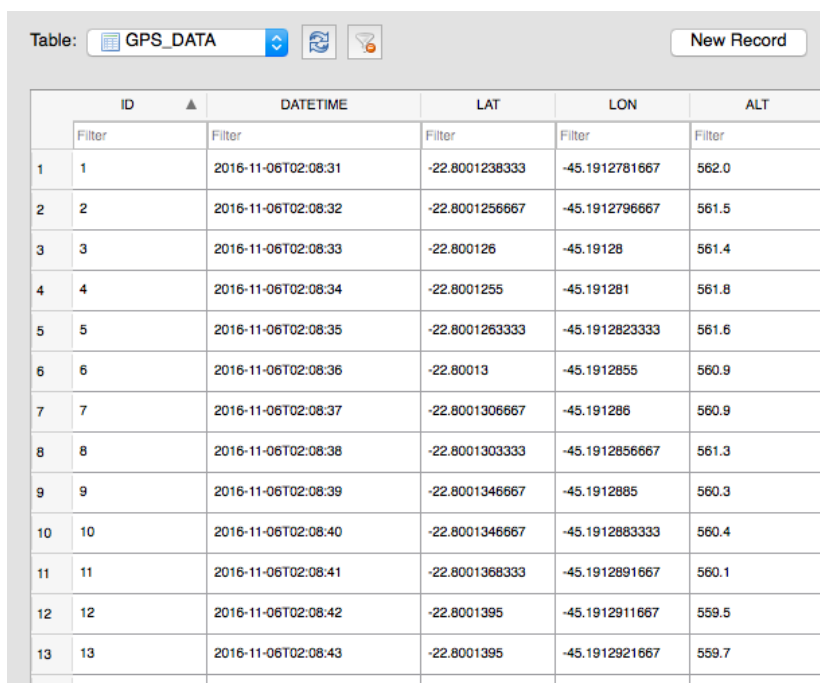


Table:

	ID ▲	DATETIME	LAT	LON	ALT
	Filter	Filter	Filter	Filter	Filter
1	1	2016-11-06T02:08:31	-22.8001238333	-45.1912781667	562.0
2	2	2016-11-06T02:08:32	-22.8001256667	-45.1912796667	561.5
3	3	2016-11-06T02:08:33	-22.800126	-45.19128	561.4
4	4	2016-11-06T02:08:34	-22.8001255	-45.191281	561.8
5	5	2016-11-06T02:08:35	-22.8001263333	-45.1912823333	561.6
6	6	2016-11-06T02:08:36	-22.80013	-45.1912855	560.9
7	7	2016-11-06T02:08:37	-22.8001306667	-45.191286	560.9
8	8	2016-11-06T02:08:38	-22.8001303333	-45.1912856667	561.3
9	9	2016-11-06T02:08:39	-22.8001346667	-45.1912885	560.3
10	10	2016-11-06T02:08:40	-22.8001346667	-45.1912883333	560.4
11	11	2016-11-06T02:08:41	-22.8001368333	-45.1912891667	560.1
12	12	2016-11-06T02:08:42	-22.8001395	-45.1912911667	559.5
13	13	2016-11-06T02:08:43	-22.8001395	-45.1912921667	559.7

Figura 4.1: Exemplo de dados de geolocalização coletados pelo sistema embarcado e armazenados em um banco de dados *SQLite*

Table:

ID	DATETIME	SENSOR	VALUE	UNIT
<input type="text" value="Filter"/>	<input type="text" value="Filter"/>	<input type="text" value="Filter"/>	<input type="text" value="Filter"/>	<input type="text" value="Filter"/>
469	2016-11-06T00:11:08.775987	Distance traveled codes cleared	10668	NULL
470	2016-11-06T00:11:09.127255	Ethanol fuel %	20.7843137255	NULL
471	2016-11-06T00:11:09.490969	Calculated engine load	16.4705882353	NULL
472	2016-11-06T00:11:09.844851	Speed	20	NULL
473	2016-11-06T00:11:10.199769	Engine RPM	1432	NULL
474	2016-11-06T00:11:10.501478	MAF	NULL	NULL
475	2016-11-06T00:11:10.835997	Fuel-air eq ratio	0.9208984375	NULL
476	2016-11-06T00:11:11.198629	Calculated engine load	20.3921568627	NULL
477	2016-11-06T00:11:11.553533	Speed	21	NULL
478	2016-11-06T00:11:11.905974	Engine RPM	1534	NULL
479	2016-11-06T00:11:12.208056	MAF	NULL	NULL
480	2016-11-06T00:11:12.527210	Fuel-air eq ratio	0.920166015625	NULL
481	2016-11-06T00:11:12.880146	Calculated engine load	21.9607843137	NULL
482	2016-11-06T00:11:13.197687	Speed	24	NULL

Figura 4.2: Exemplo de dados coletados pelo sistema embarcado através da interface OBD de um veículo. Dados estão armazenados em um banco de dados *SQLite*

4.2 Plataforma Web

Mostra-se nesta seção detalhes sobre a plataforma *web* que foi desenvolvida neste projeto. A solução desenvolvida permite visualizar informações sobre toda a frota, assim como detalhes sobre os veículos e as informações coletadas pelo sistema embarcado.

4.2.1 Resumo

Por se tratar de uma ferramenta destinada à gestão de uma frota, é importante que a plataforma *web* possua uma página que apresente um resumo sobre a situação de todos os veículos. Para isso foi desenvolvida a página mostrada na figura 4.3.

É exibida uma foto de cada veículo, assim como informações, como por exemplo, modelo, placa, cor, estimativa do odômetro e situação da manutenção. A coluna da direita mostra uma lista das manutenções que precisam ser realizadas para cada veículo. A situação de cada manutenção é indicada por meio de uma barra de progresso.

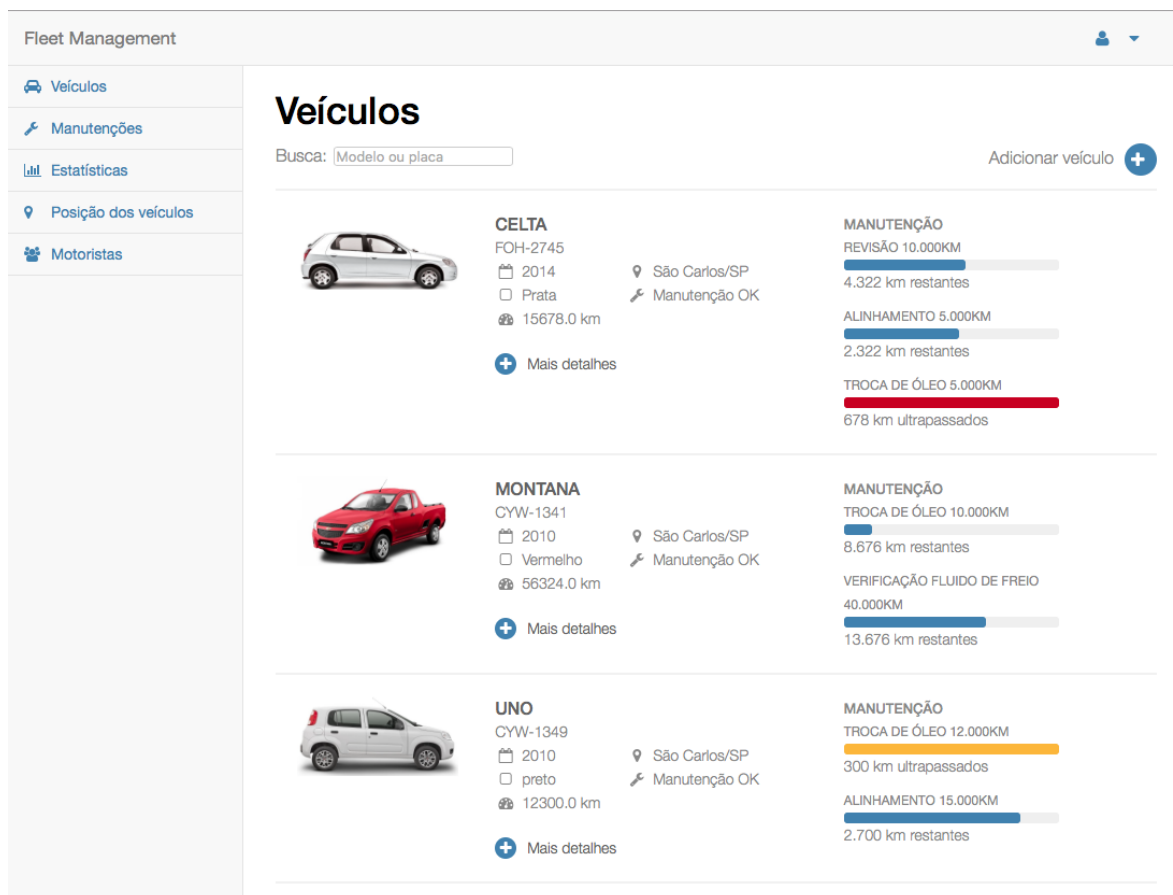


Figura 4.3: Página que contém informações sobre todos os veículos da frota

4.2.2 Adicionando um veículo à frota

A página mostrada na figura 4.4 permite adicionar um novo veículo a frota. Devem ser fornecidos dados sobre o veículo e sobre as manutenções que precisam ser realizadas. É possível adicionar uma ou mais manutenções que devem ser realizadas.

4.2.3 Dados sobre um veículo

Por fim, foi desenvolvida uma página para mostrar detalhes sobre um veículo, assim como as informações coletadas pelo sistema embarcado. Ela é mostrada nas figuras 4.5, 4.6 e 4.7. Esse tipo de página é gerada para cada veículo da frota com base nas informações enviadas pelo sistema embarcado.

Na figura 4.5 estão presentes detalhes sobre o veículo, assim como informações sobre manutenções que também são exibidas na página que contém um resumo da frota. Além disso, há um gráfico de barras que exibe a distância que foi percorrida nos últimos dias. Esse tipo de informação é obtido através do sistema embarcado. O sistema também é capaz de

gerar alertas, como por exemplo, excesso de velocidade e nível baixo de combustível.

Nas figuras 4.6 e 4.7 são mostradas mais informações coletadas pelo sistema embarcado. É possível visualizar através de um mapa os trajetos realizados. Gráficos mostram a velocidade do veículo e a carga do motor em função do tempo. A carga do motor corresponde, em termos percentuais, à razão entre o torque instantâneo e o torque máximo disponível. Essas informações permitem, por exemplo, determinar se o condutor excedeu os limites de velocidade ou se ocorreu algum desvio nos trajetos. Registrar os trajetos realizados também pode ser útil para determinar se é possível torná-los mais curtos.

A combinação de dados de geolocalização com informações lidas do computador de bordo é justamente o que diferencia este projeto em relação a outras soluções para gestão de frotas presentes no mercado brasileiro atualmente.

0.0.0.0:5000/add-vehicle

Fleet Management

- Veículos
- Manutenções
- Estatísticas
- Posição dos veículos
- Motoristas

Adicionar novo veículo

Dados do Veículo

Modelo

Placa

Ano

Cor

Odômetro

Foto do veículo

Choose File No file chosen

Dados de manutenção

Tipo de manutenção

Exemplo: Troca de Óleo

Unidade:

km dias horas

Período total

Exemplo: 10.000km

Início do período (odômetro)

Exemplo: 10.000km

Janela da manutenção

Exemplo: 500km

+ Adicionar manutenção

Adicionar veículo

Figura 4.4: Página desenvolvida para a inclusão de um novo veículo à frota

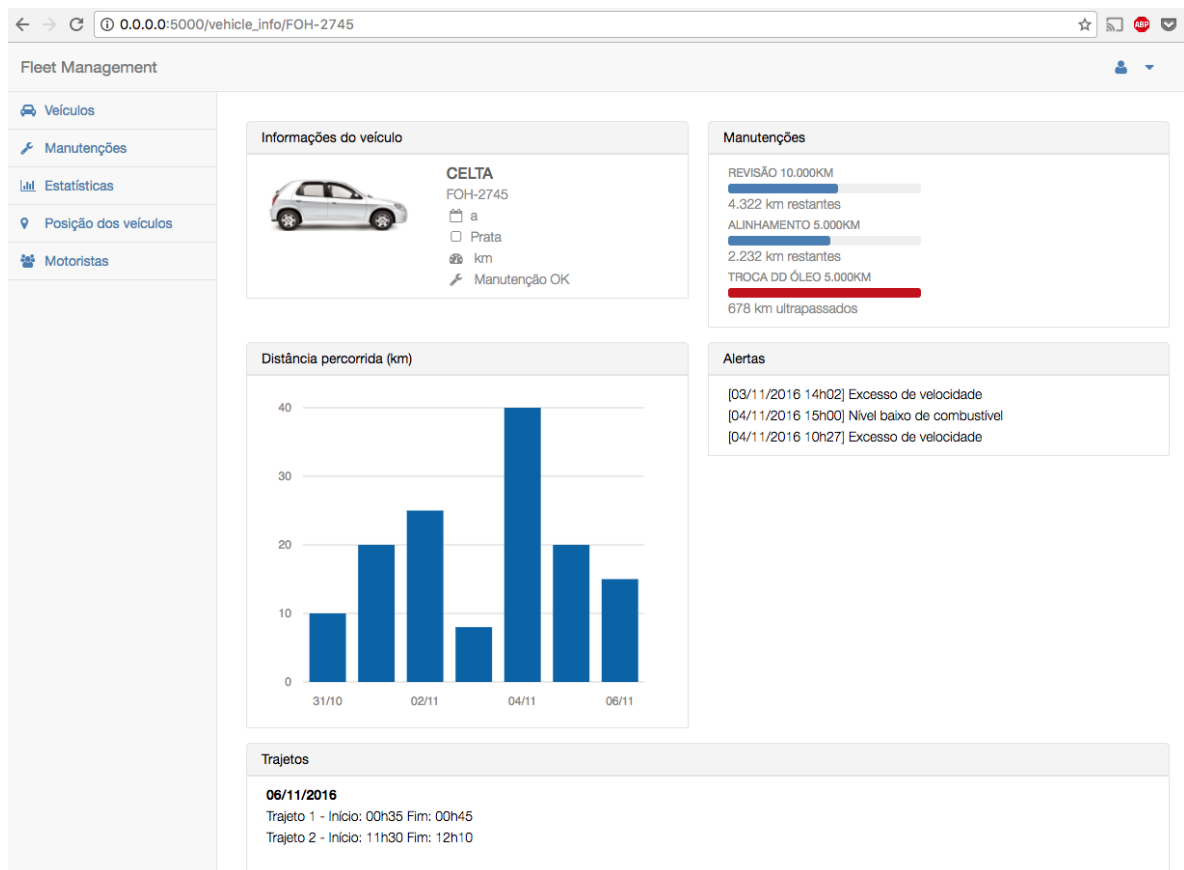


Figura 4.5: Topo da página que contém detalhes sobre um veículo. São mostradas informações sobre o veículo, manutenções, distância percorrida nos últimos dias e alertas gerados pelo sistema.

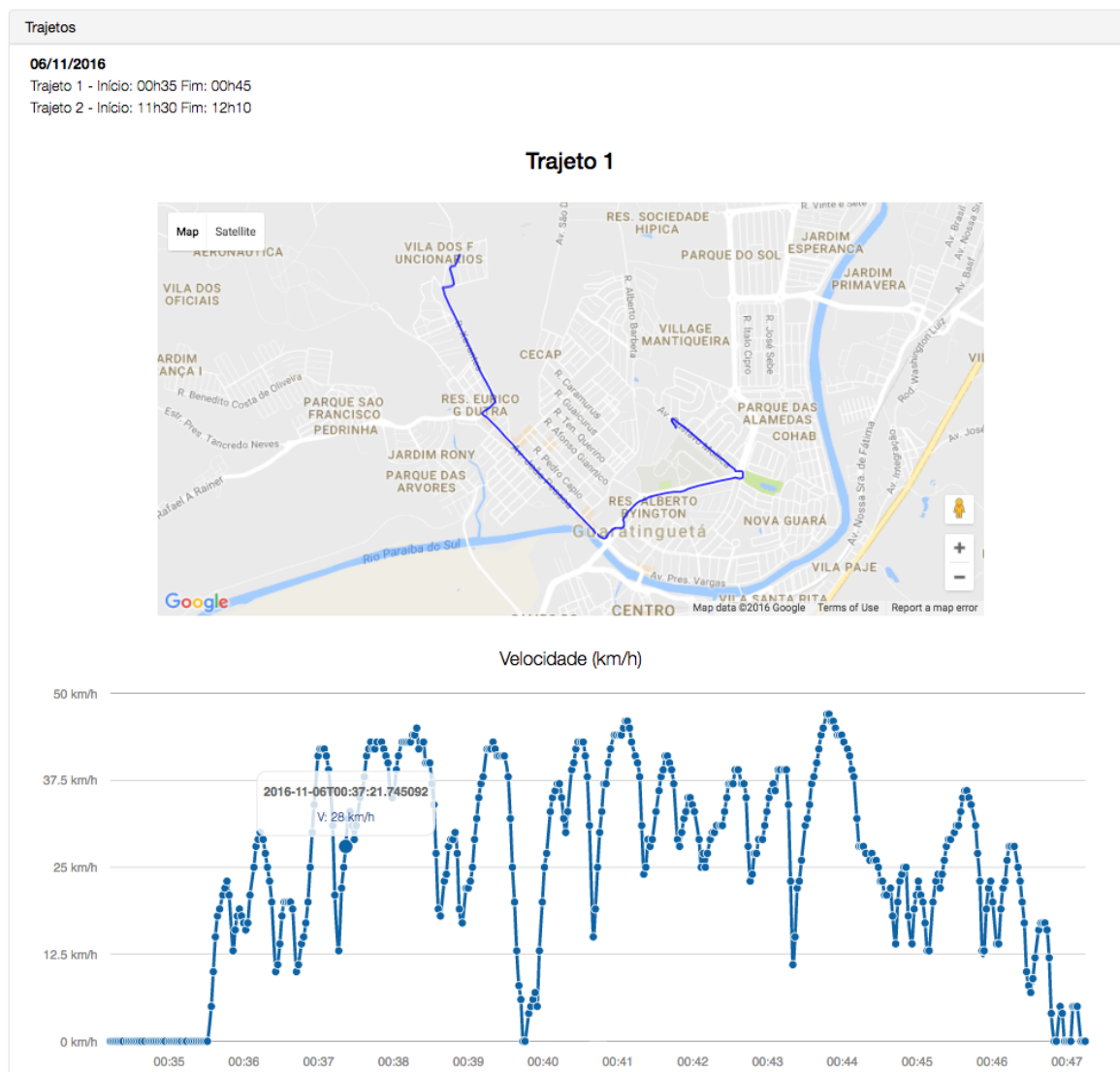


Figura 4.6: Trajeto realizado por um veículo e sua velocidade em função do tempo. Os dados foram coletados pelo sistema embarcado e são exibidos pela plataforma *web*.

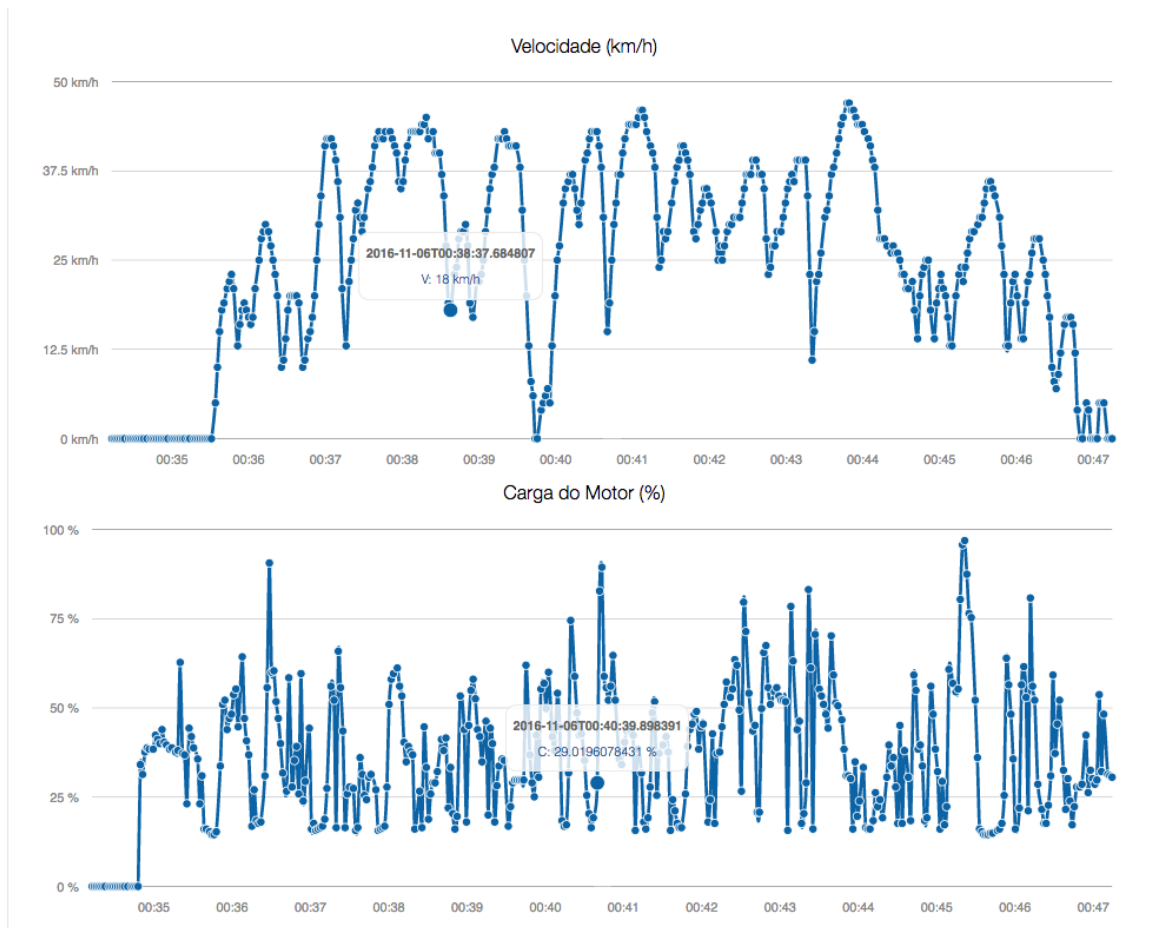


Figura 4.7: Velocidade de um veículo e a carga do motor durante um determinado trajeto. Os dados foram coletados pelo sistema embarcado e são exibidos pela plataforma *web*. A carga do motor indica, em termos percentuais, a razão entre o torque instantâneo e o torque máximo disponível.

Capítulo 5

Conclusões

5.1 Conclusões gerais

A solução desenvolvida permite que empresas possuam maior controle sobre suas frotas de veículos. A coleta de dados por meio de um sistema embarcado garante que o sistema de gestão irá trabalhar com informações atualizadas, além de eliminar a possibilidade de erro humano nas leituras. A combinação de dados de geolocalização com dados fornecidos pelo computador de bordo dos veículos resulta em uma quantidade de informações muito superior às utilizadas por outras soluções para a gestão de frotas disponíveis atualmente.

A solução desenvolvida elimina a necessidade de ler de maneira constante o odômetro de um veículo. As manutenções podem ser agendadas apenas com base nos dados recolhidos pelo sistema embarcado. Em empresas que possuem um número elevado de veículos, reduz-se drasticamente a dificuldade e o esforço envolvidos na gestão de uma frota, além de melhorar a capacidade de planejamento. A realização das manutenções em seus períodos adequados reduz a probabilidade de falhas mecânicas indesejadas nos veículos, o que representa economia de recursos para o frotista.

A capacidade de leitura dos sensores previstos no sistema OBDII é um diferencial deste projeto. É possível registrar informações, como por exemplo, velocidade, rotação do motor e carga do motor durante todos os trajetos realizados pelos veículos. Desta forma, as empresas podem verificar se os condutores respeitam os limites de velocidade e a maneira como os veículos são conduzidos. Esses dados podem, inclusive, ser utilizados em programas para a conscientização dos condutores sobre como reduzir o consumo de combustível em seus deslocamentos.

5.2 Trabalhos futuros

A solução apresentada funciona apenas em veículos leves. Veículos pesados, como por exemplo, caminhões e ônibus, utilizam outros tipos de interface diferentes do sistema OBD. Destaca-se, em particular, o padrão FMS (*Fleet Management System*) que, ao contrário do sistema OBD, foi criado com o objetivo de gerenciamento de frotas de veículos. Este padrão é baseado no protocolo SAE J1939. Sugere-se, portanto, a expansão do projeto desenvolvido de modo a oferecer suporte ao padrão FMS. Desta forma, a mesma solução poderia ser empregada para o gerenciamento de veículos leves, de caminhões e de ônibus.

Outra maneira de melhorar o sistema desenvolvido seria o desenvolvimento de um aplicativo para *smartphones* que permitisse acesso a todos os dados que atualmente são disponibilizados apenas pela plataforma *web*.

Para a redução dos custos do sistema embarcado, sugere-se a utilização do computador *Raspberry Pi Zero*. Diferentemente da *Raspberry Pi 3* que custa 35 dólares americanos, este dispositivo é comercializado por apenas 5 dólares americanos. Além do preço mais baixo, existe também a vantagem de consumir menos energia.

Referências Bibliográficas

- [1] BASTOS, E., *Estudo das Diferenças dos Requerimentos das Principais Legislações de On Board Diagnostics para Padronização de Testes de Desenvolvimento e Validação de Transmissão Automática de Automóveis*. 2012. Instituto Mauá de Tecnologia.
- [2] *NMEA Revealed*. Disponível em <http://catb.org/gpsd/NMEA.html>. Acesso em 10 de novembro de 2016.
- [3] *SQLite Home Page*. Disponível em <https://sqlite.org/>. Acesso em 5 de novembro de 2016.
- [4] *Python Website*. Disponível em <https://www.python.org/>. Acesso em 2 de novembro de 2016.
- [5] *sqlite3 - DB-API 2.0 Interface for SQLite databases*. Disponível em <https://docs.python.org/2/library/sqlite3.html>. Acesso em 2 de novembro de 2016.
- [6] ELM Electronics, "ELM327 datasheet", 2015.
- [7] *threading - Higher-level threading interface*. Disponível em <https://docs.python.org/2/library/threading.html>. Acesso em 9 de novembro de 2016.
- [8] *pySerial's documentation*. Disponível em <http://pyserial.readthedocs.io/en/latest/index.html>. Acesso em 10 de novembro de 2016.
- [9] *PyBluez: Bluetooth Python extension module*. Disponível em <https://github.com/karulis/pybluez>. Acesso em 15 de novembro de 2016.
- [10] *Requests: HTTP for Humans*. Disponível em <http://docs.python-requests.org/en/master/>. Acesso em 12 de novembro de 2016.
- [11] *Flask*. Disponível em <http://flask.pocoo.org/>. Acesso em 10 de novembro de 2016.
- [12] *Jinja*. Disponível em <http://jinja.pocoo.org/>. Acesso em 10 de novembro de 2016.

- [13] *Werkzeug*. Disponível em <http://werkzeug.pocoo.org/>. Acesso em 10 de novembro de 2016.
- [14] *Morris.js*. Disponível em <http://morrisjs.github.io/morris.js/>. Acesso em 14 de novembro de 2016.
- [15] *Google Maps API*. Disponível em <https://developers.google.com/maps/documentation/javascript/>. Acesso em 13 de novembro de 2016.
- [16] *The XMLHttpRequest Object*. Disponível em http://www.w3schools.com/xml/xml_http.asp. Acesso em 15 de novembro de 2016.