UNIVERSIDADE DO VALE DO RIO DOS SINOS — UNISINOS
ESCOLA POLITÉCNICA
ENGENHARIA DA COMPUTAÇÃO
NÍVEL GRADUAÇÃO

FELIPE DE VILLA CAON

DETECÇÃO E MAPEAMENTO DE BURACOS EM VIAS ASFALTADAS UTILIZANDO VISÃO COMPUTACIONAL POR MEIO DE UM EMBARCADO INSTALADO EM VEÍCULO AUTOMOTIVO

SÃO LEOPOLDO
2019

Felipe de Villa Caon

DETECÇÃO E MAPEAMENTO DE BURACOS EM VIAS ASFALTADAS UTILIZANDO VISÃO COMPUTACIONAL POR MEIO DE UM EMBARCADO INSTALADO EM VEÍCULO AUTOMOTIVO

Dissertação apresentada como requisito parcial para obtenção do título de Bacharel em Engenharia pelo Curso de Engenharia da Computação da Universidade do Vale do Rio dos Sinos – UNISINOS

Orientador:
Prof. Jean Schmith

São Leopoldo
2019

O sucesso é uma consequência e não um objetivo.
— GUSTAVE FLAUBERT

AGRADECIMENTOS

A todos que me incentivaram durante a caminhada do TCC, fornecendo equipamentos, dados, dicas ou simplesmente dando o apoio necessário para a finalização. A estes, meu muito obrigado.

RESUMO

Buracos em vias asfaltadas são um grande problema, uma vez que estes podem estes podem realizar estragos nos carros ou até causar acidentes que podem vir a serem fatais. Dessa forma, faz-se necessário realizar manutenções com certa periodicidade nas vias para evitar tais desastres. Este trabalho tem como objetivo a proposta e implementação de um sistema inteligente para detecção de buracos utilizando uma câmera simples e técnicas de processamento de imagem. O sistema, ainda, é integrado com o propósito de enviar dados captados a órgãos supervisores, a fim de que medidas cabíveis sejam tomadas. O algoritmo proposto se mostra válido uma vez que a precisão ficou em 81%.

Palavras-chave: Processamento de imagem. Sistema inteligente. Detecção de buracos.

LISTA DE FIGURAS

Figura 1: Exemplo de imagem digital descolorida de resolução 28 x 28 12
Figura 2: Histograma em uma imagem 13
Figura 3: Processo de convolução utilizando kernel 14
Figura 4: Aplicação de desfoque gaussiano em imagem 15
Figura 5: Aplicação de *thresholding* em uma imagem 17
Figura 6: Exemplo de aplicação do método de canny 19
Figura 7: *Kernel* em forma de cruz 20
Figura 8: Exemplo de dilatação 20
Figura 9: Exemplo de erosão 21

Figura 10: Câmera montada no centro do para-brisa do veículo 23
Figura 11: Diagrama de bloco do algoritmo 24
Figura 12: Área de interesse 26
Figura 13: Imagem final gerada pelo algoritmo 27
Figura 14: Diagrama do projeto completo 27
Figura 15: Mapa com marcações de buracos 28
Figura 16: Ícone para aviso de buraco 28
Figura 17: Informações de buraco selecionado 29

Figura 18: Parte das imagens selecionadas para testes preliminares 30
Figura 19: Testes preliminares em parte do *dataset* de Figuras 31
Figura 20: *Setup* de equipamentos no carro 32
Figura 21: Cenário fácil para detecção de buracos 33
Figura 22: Cenário difícil para detecção de buracos 34
Figura 23: Mapa do trecho realizado com buracos identificados 35

LISTA DE TABELAS

Tabela 1: Exemplo de matriz de confusão 22

Tabela 2: Matriz de confusão para vídeo utilizado na validação 35

LISTA DE SIGLAS

RGB	*Red, Green, Blue*
NTSC	*National Television System Committee*
ROI	*Region of interest*
JSON	*JavaScript Object Notation*

SUMÁRIO

1 INTRODUÇÃO **10**

2 FUNDAMENTAÇÃO TEÓRICA **11**
2.1 Trabalhos relacionados 11
2.2 Imagem digital 12
2.2.1 Transformação para escala em cinza 12
2.2.2 Histograma 13
2.3 Python 14
2.4 Kernel 14
2.5 Desfoque Gaussiano 15
2.6 Segmentação de imagem 16
2.6.1 *Thresholding* 16
2.6.1.1 *Thresholding* adaptativo 17
2.6.1.1.1 Método de Otsu 17
2.6.2 Detecção de borda 18
2.6.2.1 Algoritmo de Canny 18
2.7 Operações Morfológicas 19
2.7.1 Dilatação 20
2.7.2 Erosão 21
2.8 Validação dos dados 21
2.8.1 Precisão e *recall* 21
2.8.2 Matriz de Confusão 22

3 METODOLOGIA **23**
3.1 Equipamento utilizado 23
3.2 Configuração do ambiente 24
3.3 Diagrama de blocos do algoritmo de detecção de buracos 24
3.3.1 Transformação da imagem para preto e branco 25
3.3.2 Remoção de ruído 25
3.3.3 Aplicação de *thresholding* 25
3.3.4 Erosão e dilatação 25
3.3.5 Definição da região de interesse 25
3.3.6 Aplicação de Canny 26
3.3.7 Extração de contornos 26
3.3.8 Retorno de informações de buracos 26
3.3.9 Envio de dados à base de dados 27
3.4 Diagrama de blocos do projeto completo 27
3.4.1 Checagem de funcionamento do módulo GPS e câmera 28
3.4.2 Inicialização do algoritmo de detecção de buracos 28
3.4.3 Construção do mapa com pontos onde buracos foram encontrados 28

4 RESULTADOS OBTIDOS **30**
4.1 Testes preliminares 30
4.2 Validação 32
4.3 Discussão dos resultados 32

5 CONCLUSÃO **36**
5.1 Trabalhos futuros 36

REFERÊNCIAS **38**

1 INTRODUÇÃO

Buracos são cavidades no asfalto (DNIT, 2003). São comumente gerados a partir de trincas formadas devido às altas cargas impostas a rodovias que não estão preparadas para receber tal peso. O problema de trincas aliado a chuvas moderadas fazem com que elas se expandam permitindo a infiltração da água e, posteriormente, seja gerado um buraco.

Buracos devem ser consertados da maneira mais rápida possível uma vez que sua expansão não irá parar. Comprometendo, assim, a segurança das pessoas que passam na via. Buracos podem gerar estragos no carro, especialmente em pneus que podem vir a furar. Podem causar colisões ou jogar um veículo para fora da pista, visto que pode ser necessário frear o veículo bruscamente em linha reta ou em uma curva.

Buracos são indicativos de problemas estruturais em vias asfaltadas e a detecção de forma prematura pode prolongar a vida útil da rodovia e evitar acidentes, diminuindo também índices de mortalidade. Atualmente, o sistema de gestão e checagem de buracos é feita de forma manual, uma vez que o usuário pode avisar órgãos governamentais sobre algum buraco em tal coordenada. Contudo, esta solução, além de ser custosa, é também demorada, em razão da necessidade da manutenção de servidores para tal tarefa juntamente com muito esforço braçal para a validação de dados.

Para a resolução de tais problemas é proposto um sistema que irá funcionar através de processamento de imagens no qual diferentes métodos e algoritmos serão aplicados em imagens geradas de uma câmera de vídeo para identificar o buraco desejado da imagem com maior maestria possível.

Este trabalho tem como objetivo o desenvolvimento de um protótipo de sistema automático para detecção de buracos em vias pavimentadas. Este sistema utilizará dispositivos simples que devem detectar buracos de forma rápida e eficiente através de um *setup* que contém uma câmera, um módulo GPS e um *Raspberry Pi 3*. Como resultado, foram obtidos uma gama de dados grandes em que é possível detectar buracos juntamente com a localidade destes para posteriormente criar um sistema *web* onde seja possível o aviso destes problemas a autoridades responsáveis que possam tomar atitudes a fim de consertarem pontos defeituosos. Por fim, obteve-se 81% de precisão e 89% de *recall*, mostrando que o algoritmo é eficiente e pode ser usado na detecção de buracos em vias asfaltadas.

2 FUNDAMENTAÇÃO TEÓRICA

A necessidade de manutenção de vias asfaltadas é um problema existente em qualquer lugar onde há tráfego de veículos. Logo, um sistema inteligente para realizar análise e aviso a órgãos vigentes se faz necessário.

2.1 Trabalhos relacionados

Com o objetivo de resolver o problema de detecção de buracos em vias, diversas soluções já foram testadas. Estas soluções besearam-se em vibração (Mednis et al., 2011), escaneamento 3D (Li et al., 2009) e processamento de imagem. Todavia, das três soluções apresentadas, vibração é uma solução não muito confiável, uma vez que a roda do carro deve passar dentro do buraco para o acelerômetro detectá-lo. Escaneamento e reconstrução 3D foi provado como um método extremamente preciso, porém demasiado caro devido ao preço de equipamentos para esta aplicação. Por fim, métodos baseados em processamento de imagem na qual é possível utilizar apenas uma câmera 2D para realizar a análise de buracos foram testados por Ryu, Kim e Kim (2015) e Jo e Ryu (2015).

A análise de buracos através de imagem foi introduzida, inicialmente, por Koch e Brilakis (2011), na qual buracos eram encontrados por meio de características específicas, como a coloração e forma geométrica. Buza, Omanovic e Huseinovic (2013) propuseram um algoritmo para detecção de buracos utilizando métodos não supervisionados que consistem em segmentação de imagem, identificação de formas geométricas e extração do que foi obtido.

Atualmente, em grande parte dos artigos relacionados, há processamento de imagem com finalidade de detectar buracos. São utilizados algoritmos que dependem da aplicação de filtros para, posteriormente, ser realizado um *thresholding* no qual dados úteis são extraídos, seguido de algoritmos de detecção de borda para a checagem e confirmação da detecção do buraco. Grande parte dos estudos baseiam-se no método criado por Otsu (1979) para a binarização de uma imagem, uma vez que este método se adapta conforme o ambiente.

O trabalho de Jo e Ryu (2015) ainda propõe um método para extração da área de interesse de uma imagem: aplica-se uma máscara triangular em frente ao veículo e, com a ajuda de utilização de algoritmos para detecção de faixa, é possível detectar a rua onde o carro trafega e, assim, analisar apenas uma parte da imagem deixando o processo mais rápido.

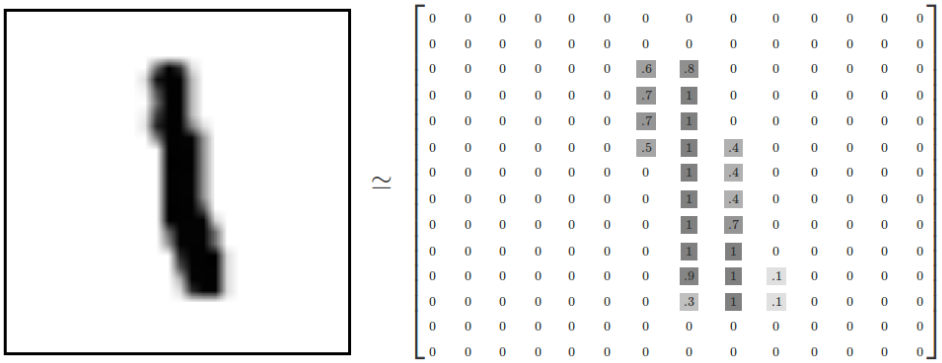
Em um trabalho posterior, Kim e Ryu (2014) apresentam uma solução de detecção de buracos em vídeo em que são discutidas maneiras de realizar a contagem de buracos uma vez que o mesmo buraco pode aparecer em vários *frames*.

Este trabalho irá se basear na sequência de algoritmos para processamento de imagem proposta por Jo e Ryu (2015). Com o objetivo de otimizar os resultados, métodos utilizados para detecção de área de interesse utilizados por Nienaber (2015) também serão implementados.

2.2 Imagem digital

Uma imagem digital é representada por um vetor de valores bidimensional que tem tamanho limitado por meio da resolução da imagem. A Figura 1 mostra um exemplo de imagem descolorida que possui resolução de 28 x 28 *pixels*. Áreas mais escuras da imagem são determinadas por valores maiores, enquanto que valores mais baixos determinam áreas mais claras. (Gonzalez e Woods, 2000).

Figura 1 - Exemplo de imagem digital descolorida de resolução 28 x 28



Fonte: https://tensorflow.rstudio.com/tensorflow/articles/tutorial_mnist_beginners.html

Ainda de acordo com Gonzalez e Woods (2000), imagens coloridas seguem o mesmo padrão em que o valor máximo de cada *pixel* pode ser 255 devido ao máximo que um valor de 8 *bits* pode atingir. Imagens coloridas também possuem canais de cores, no qual o padrão RGB é muito utilizado. A partir do momento que uma imagem possui canais de cores em sua composição, ela deixa de ser um *array* bidimensional e passa a ser um *array* tridimensional, visto que irá ter mais de uma camada de *pixels* para cada canal de cor desejado.

O conceito de imagem digital é extremamente necessário em qualquer atividade na qual processamento de imagem é utilizado, uma vez que existe a necessidade de detectar objetos em uma imagem que podem corresponder a menos *pixels* (pequena anomalia em rodovia) ou vários *pixels* (anomalia mais considerável). Este trabalho se baseia totalmente em processamento digital de imagens e os conceitos desta seção serão amplamente utilizados ao longo do projeto.

2.2.1 Transformação para escala em cinza

Uma imagem pode ser convertida em tons de cinza dividindo cada *pixel* pelo valor máximo de 255 conforme exemplificado por Chollet (2018). Com o intuito de uma conversão ainda mais fiel, é dado um pequeno peso para cada *pixel* em uma imagem conforme proposta do Comitê Nacional de Sistema de Televisão (NTSC). Dessa maneira, aplica-se a equação 1 para cada porção da imagem:

$$Y = 0.299 * R + 0.587 * G + 0.114 * B \tag{1}$$

Onde,

- R, *Red*, valor do pixel para a camada vermelha;
- G, *Green*, valor do pixel para a camada verde;
- B, *Blue*, valor do pixel para a camada azul;
- Y, imagem obtida.

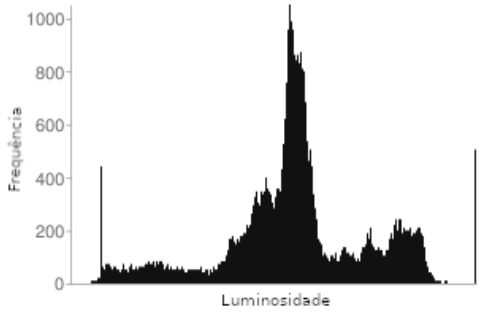
A grande necessidade de converter uma imagem para cinza é a diminuição de complexidade que a imagem terá. É possível abordar conceitos de luminosidade, contraste, textura, borda, sombra e perspectiva em imagens sem necessidade de cor, havendo assim muito menos esforço computacional. Se necessário, ainda é possível convertê-las para colorido posteriormente. A conversão de cores utilizando o sistema NTSC será utilizada neste trabalho para que as imagens obtidas pela câmera fiquem computacionalmente mais fáceis de serem analisadas devido à ausência de cor e por consequência torna-se o algoritmo mais rápido, podendo analisar dados de imagens de forma mais ágil.

2.2.2 Histograma

Já é sabido que cada *pixel* de uma imagem pode deter um valor que pode variar de 0 até 255 em cada um dos canais RGB. Um histograma é definido e gerado quando a imagem é varrida e cada pixel com seu determinado valor é registrado e contado. Ao final, é gerada uma curva que mostra uma distribuição revelando a quantidade de vezes em que cada valor aparece. Sendo assim, cada canal de cor terá um histograma próprio.

Levando em consideração a primeira imagem da Figura 5, o histograma da Figura 2 é gerado. Onde, o eixo X mostra a variação de 0 a 255 de cada pixel, e o eixo Y mostra a quantidade de pixel para cada valor apresentado no eixo X.

Figura 2 - Histograma em uma imagem



Fonte: Produzido pelo autor

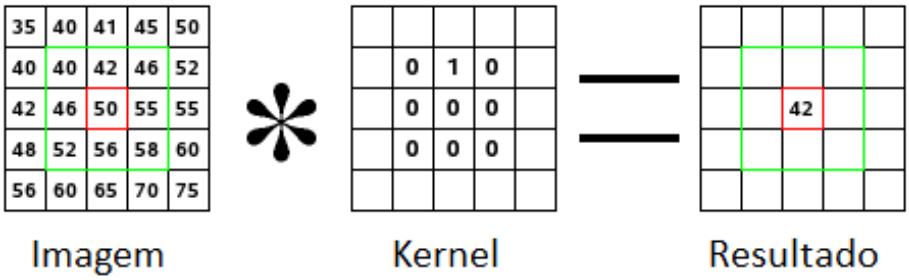
2.3 Python

Python é uma linguagem de programação de alto nível lançada a público em 1991 por Guido van Rossum (Python, 2017). Seu objetivo é ser uma linguagem que enfatiza a fácil escrita e compreensão de código (Python, 2006). De acordo com TIOBE (2019) é a terceira linguagem mais utilizada, possuindo mais de 170 mil pacotes desenvolvidos por colaboradores externos que englobam todo tipo de indústria (Python, 2019). Este projeto utilizará das vantagens da linguagem Python juntamente com a biblioteca OpenCV para realizar a análise de imagens.

2.4 Kernel

Um *kernel* é uma matriz pequena de tamanho definido pelo usuário, utilizada para a aplicação de efeitos sob a imagem necessária. Cada matriz pode gerar um resultado diferente devido ao processo de convolução gerado pelos valores anteriormente definidos. (Bradski e Kaehler, 2008). A Figura 3 exemplifica o processo utilizando um *kernel* 3x3.

Figura 3 - Processo de convolução utilizando kernel



Fonte: https://docs.gimp.org/2.8/en/plug-in-convmatrix.html

O funcionamento é simples: tendo um kernel de tamanho 5x5. Para cada *pixel* da imagem é sobreposto o *kernel* escolhido e os *pixels* da vizinhança têm seus valores multiplicados pelos valores do *kernel*. Por fim, é feita a soma dos valores obtidos e este valor será definido como o valor do pixel da imagem gerada. (Bradski e Kaehler, 2008). Este processo também é conhecido como convolução.

Kernels são amplamente utilizados no processamento de imagem quando há necessidade de realizar alteração na estrutura original de uma figura. A alteração realizada pode variar de acordo com os valores inseridos no *kernel*. Certos valores fazem com que a imagem fique mais borrada, outros transformam a imagem de modo a ficar mais aprimorada. Alguns valores específicos podem mostrar apenas as bordas da imagem.

2.5 Desfoque Gaussiano

O desfoque gaussiano funciona através da função gaussiana que gera uma curva de distribuição normal. Originalmente, a função gaussiana produz uma distribuição unidimensional, a equação 2 acarreta a distribuição gaussiana para duas dimensões. Esta é obtida através da multiplicação da função original por ela mesma. (Gedraite e Hadad, 2011). A distribuição produzida é utilizada em um *kernel* que, posteriormente, é convolucionado com a imagem.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2}$$

No qual,

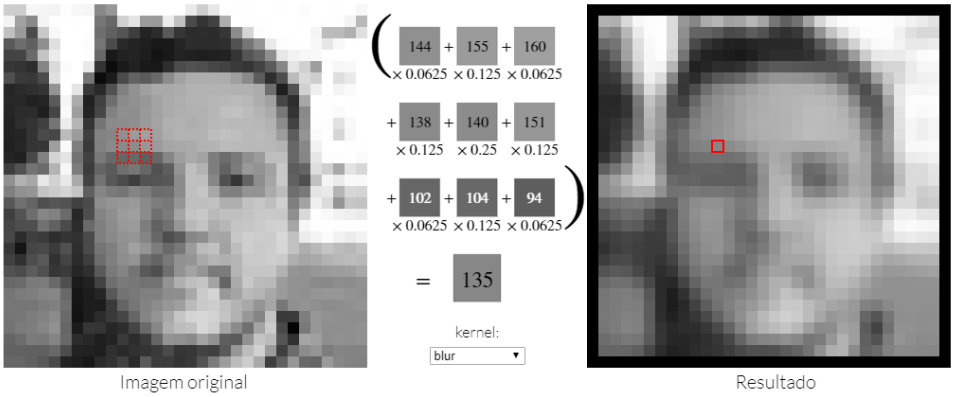
- G, valor de saída do *pixel* após aplicação da distribuição gaussiana;
- σ, controla a variação da distribuição normal;
- π, valor de pi;
- x, posição do pixel no eixo x da imagem;
- y, posição do pixel no eixo y da imagem.

O valor de sigma da equação 2 controla a variação da distribuição normal a fim de que seu valor médio (central) sempre fique destacado de acordo com o escolhido. (Shapiro, 2001).

Como os valores do *kernel* possuem distribuição normal, o resultado é bastante congruente com a entrada. O desfoque gaussiano é muito utilizado quando há necessidade e remoção dos ruídos de imagens e, posteriormente, redução dos detalhes. (Gedraite e Hadad, 2011).

A Figura 4 exibe uma imagem de entrada que passa por um *kernel* de distribuição normal gerado pela função gaussiana.

Figura 4 - Aplicação de desfoque gaussiano em imagem



Fonte: http://setosa.io/ev/image-kernels/

Nota-se que a imagem possui menos detalhes, porém ainda é possível a identificação de seu conteúdo. O desfoque gaussiano será utilizado neste trabalho para remover pequenos ruídos da

imagem que podem eventualmente causar com que o algoritmo detecte alguma anomalia onde não existe.

2.6 Segmentação de imagem

Um algoritmo de segmentação de imagem funciona de modo a se basear em certos critérios para dividir uma imagem em áreas de interesse que podem ser úteis em uma análise posterior. (Song Yuheng, 2017). De acordo com Barghout e Lee (2004), a realização da segmentação de uma imagem tem como objetivo criar uma figura simplificada que contenha as mesmas características da imagem original e que seja mais fácil analisar.

A segmentação de imagem é empregada em diversas indústrias para a localização de padrões e objetos conforme discutido em Rosin, Hervás e Barredo (2000) e Liao et al. (1996). Existem diversos tipos de algoritmos de segmentação, dentre eles *thresholding* e detecção de borda. Nas próximas subseções, cada algoritmo mencionado será melhor explanado.

2.6.1 *Thresholding*

O conceito de *thresholding* permite que, a partir de uma imagem em escala de cinza, possam ser criadas figuras binárias. (Shapiro, 2001). Em uma imagem, cada pixel tem seu valor comparado com o valor de referência de *threshold* e esta será alterada conforme o algoritmo implementado. Em implementações mais sofisticadas, nas quais deseja-se aplicar o método para imagens coloridas, é possível selecionar diversos valores para *thresholding*. (Sherlin Suresh, 2017).

A aplicação básica de *thresholding* é bastante simples. Primeiramente um valor entre 0 e 255 deve ser escolhido, neste exemplo será selecionado o valor de 127. Após, uma imagem deve ter todos seus *pixels* lidos e salvos. Ao final, *pixels* com valores abaixo de 127 têm seus valores alterados para 0 (preto), *pixels* com valores acima de 127 são alterados para branco.

A equação 3 mostra um algoritmo simples de *thresholding* que altera os *pixels* da imagem para preto se a luminosidade for menor que uma constante T definida pelo usuário. Se o pixel apresentar valor maior que a constante T, o resultado será branco.

$$dst(x,y) = \begin{cases} valor\_maximo & se \;\; src(x,y) > T \\ 0 & \forall \end{cases} \tag{3}$$

No qual,

- T, constante definida de *threshold*;
- valor_maximo, intensidade que o *pixel* terá caso tiver valor maior que T. Geralmente usado branco (1, ou 255) como valor padrão.

A Figura 5 representa a binarização de uma imagem utilizando o algoritmo da equação

3. Conforme Barghout e Lee (2004), a imagem com *thresholding* deve ser mais fácil de ser analisada. Na Figura 5 é percebido que, após a aplicação do algoritmo, a imagem ficou em preto e branco, tendo apenas duas cores possíveis. Porém, ao mesmo tempo, ainda é possível identificar que na imagem existem árvores, alguns postes e imperfeições no asfalto.

Figura 5 - Aplicação de *thresholding* em uma imagem

Imagem Original | Imagem com *thresholding*

Fonte: Produzido pelo autor

2.6.1.1 *Thresholding* adaptativo

Em muitos casos, um valor fixo de *threshold* não é apropriado para analisar imagens. Dependendo do nível de brilho, constraste ou exposição, os resultados podem não representar de forma confiável os dados do conjunto a ser testado. A fim de resolver o problema, podem ser usados algoritmos adaptativos em que a imagem é dividida em porções e, nessas porções, são realizados cálculos nos quais é possível obter um valor de *threshold* diferenciado baseado no histograma. (W. Burger, 2013).

No caso da análise de vias, um *thresholding* adaptativo deve ser utilizado devido às variações de luminosidade que podem existir na rodovia. Estas variações de luminosidade dependem diretamente do clima, que pode variar de ensolarado até nublado, alterando completamente a maneira de como a cor é percebida pela a câmera que processará as imagens. O algoritmo adaptativo empregado neste trabalho será o Método de Otsu.

2.6.1.1.1 Método de Otsu

O método de Otsu procura identificar um valor final otimizado de *threshold* a partir da análise em uma imagem em escala cinza. Este método envolve iterar sobre todos os valores de *threshold* possíveis, de 0 a 255, para uma imagem e identificar o valor que mais deixa os *pixels* da imagem dispersos, separando-os em *pixels* de primeiro e segundo plano. (Otsu, 1979).

O objetivo do algoritmo é encontrar um valor onde os *pixels* definidos como primeiro e segundo plano fiquem o mais dispersos possíveis. (W. Burger, 2013). A decisão acontece no momento que:

1. a variância calculada é a mínima possível;

2. as médias de cada grupo estão numericamente distantes uma da outra.

O algoritmo faz o uso do histograma da imagem em que, inicialmente, é escolhido um valor de *threshold* (T) presente. Com o valor T definido, são criados dois grupos: o primeiro com *pixels* de valores menores que T, e, o segundo, com valores maiores. São referenciados também como grupos de primeiro e segundo plano.

Para cada grupo, são calculados valores de variância, média e peso. Na sequência, são realizadas comparações para a obtenção do valor de T na qual as condições anteriores melhor se satisfazem. Por fim, este valor de T será utilizado como *threshold*.

A aplicação do método de Otsu neste trabalho se torna necessária uma vez que este método adaptativo responde muito bem a variações de luminosidade. Deste modo, a detecção de anomalias se torna mais fácil.

2.6.2 Detecção de borda

A aplicação de detecção de borda em uma imagem permite, como o próprio nome sugere, que sejam detectadas bordas através da mudanças repentina dos valores de *pixels* de uma figura. No geral, mudanças rápidas no valor de luminosidade em uma imagem representam eventos importantes como transição de objetos ou descontinuação de profundidade.

2.6.2.1 Algoritmo de Canny

O processo de detecção de bordas que será empregado neste trabalho foi sugerido por Canny, em sua publicação, um algoritmo multiestágio que produz uma solução otimizada é apresentado. O objetivo do algoritmo é ser preciso o suficiente em seu resultado, detectando o máximo de bordas possíveis de acordo com os parâmetros escolhidos, ignorando ruídos que podem vir a gerar falsos positivos. Canny (1987).

O algoritmo de detecção de borda de Canny possui cinco etapas:

1. redução de ruído;

2. descobrimento do gradiente da imagem;

3. encontro da orientação da borda;

4. supressão não-máxima;

5. limiarização com histerese.

Nos próximos parágrafos há uma explicação breve de cada etapa do algoritmo de Canny (1987).

Na primeira etapa - redução de ruído - um *kernel* gaussiano é aplicado na imagem. O *kernel* gerado nesta etapa sofre o mesmo processo do que foi citado na seção 2.5.

Na sequência, para a realização das etapas 2 e 3, o *kernel* de Sobel, descrito por Sobel (2014) é aplicado sob a imagem de forma horizontal e vertical. Deste modo, é possível calcular o valor de gradiente de cada pixel com sua orientação.

Após obter-se os valores de gradiente com a orientação de cada pixel, a etapa de supressão não-máxima é aplicada. Neste momento, o algoritmo irá varrer toda a imagem checando se cada pixel tem o valor máximo em sua vizinhança e se este valor tem a mesma orientação do gradiente. Caso as condições acima não forem satisfeitas, o pixel é removido da imagem.

O grande problema do *thresholding* básico para o caso de detecção de bordas é que, em muitas situações ele pode eliminar bordas reais se o valor de *thresholding* for muito alto. Para isso é utilizada uma outra abordagem chamada de *Tresholding* com histerese. Neste algoritmo, dois valores de *thresholding* devem ser escolhidos. O primeiro será a referência, então todos os *pixels* que tiverem valores acima do que foi escolhido serão automaticamente detectados como borda, conforme visto na seção 2.6.1. O segundo valor irá explorar o fato de que os *pixels* de bordas estão normalmente conectados entre si. Deste modo, se a sequência da borda estiver conectada com algum *pixel* com valor acima da referência, a borda também será aceita. Em qualquer outro caso a borda é ignorada.

A aplicação de todas as etapas em uma imagem pode ser visualizada na Figura 6, em que a foto de uma via foi utilizada.

Figura 6 - Exemplo de aplicação do método de canny

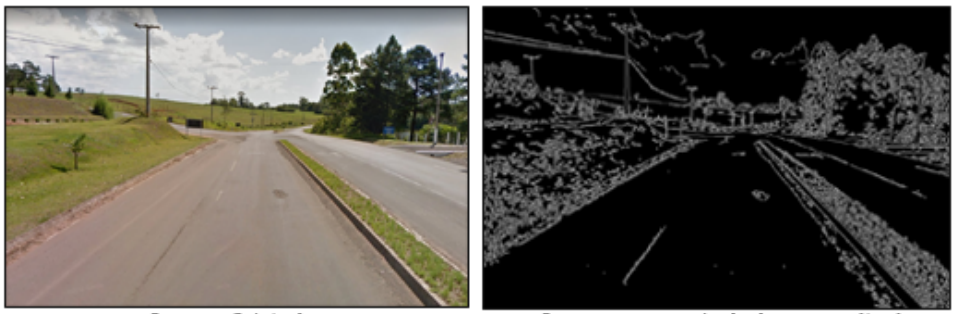


Imagem Original | Imagem com o método de canny aplicado

Fonte: O autor, 2019

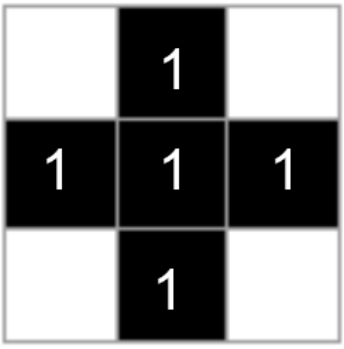
2.7 Operações Morfológicas

Para a realização de uma operação morfológica, um *kernel* específico é aplicado em uma imagem binária. Supondo um *kernel* em forma de cruz, conforme apresentado na Figura 7.

Deseja-se que o *kernel* varra a imagem e realize operações de *hit* e *miss* nela, na qual para cada pixel da imagem é checado se os valores 1 da estrutura de cruz se chocam (*hit*) com algum valor 1 da imagem. Caso o resultado seja positivo, a condição desejada é aplicada, o oposto

acontece em casos de *miss*.

Figura 7 - *Kernel* em forma de cruz



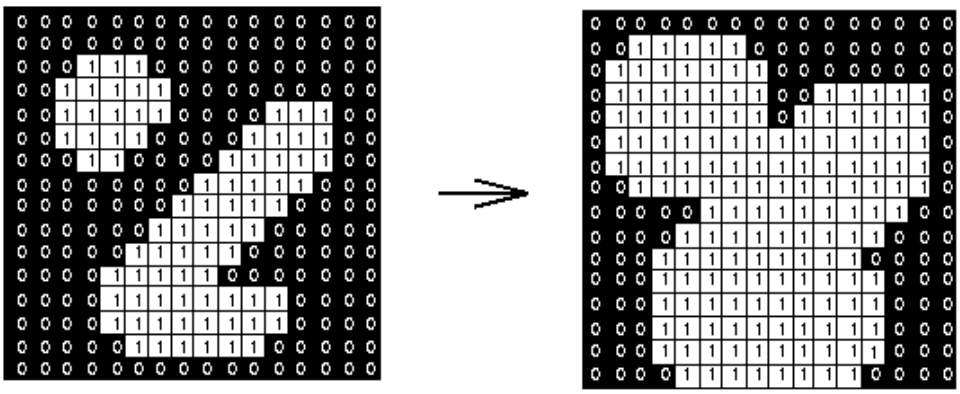
Fonte: O autor, 2019

Operações morfológicas são comumente usadas para isolamento de objetos, remoção de ruído e detecção de buracos ou pontos de intensidade de uma imagem como descrito por Bradski e Kaehler (2008). Para este projeto, operações morfológicas serão úteis para erodir ruídos que o *thresholding* pode vir a gerar. Ainda, dilatar casos que devem ser destacados para a posterior checagem e validação de buracos.

2.7.1 Dilatação

De acordo com Bradski e Kaehler (2008), a aplicação de dilatação tem como objetivo expandir a imagem de modo a deixá-la mais visível a cada iteração executada a ela. Levando em consideração o exemplo mencionado em 2.7 e a Figura 8, o algoritmo com a condição de dilatação aplicado a uma imagem deve expandir de modo a adicionar um pixel ao redor da imagem toda vez que o resultado da operação for *hit*. Ou seja, toda vez que o algoritmo detectar 1, a imagem deve ser expandida.

Figura 8 - Exemplo de dilatação

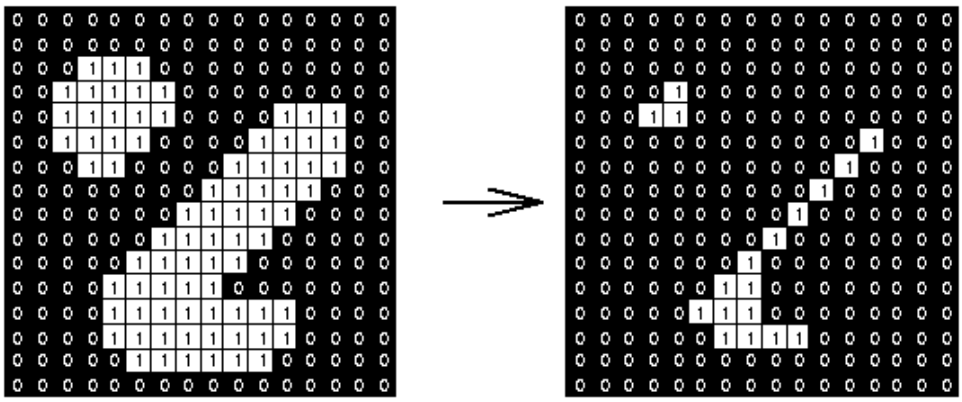


Fonte: Wolfram Alpha

2.7.2 Erosão

Para a erosão, um processo bastante similar ao anteriormente mencionado acontece. Tendo o mesmo elemento estruturador em forma de cruz, a Figura 9 será a saída. A diferença para este caso é que, na eventualidade da operação de *hit*, o pixel detectado será removido da imagem, deixando-a mais encolhida em relação a imagem original. (Bradski e Kaehler, 2008). Em outras palavras, quando o algoritmo detectar 1, a imagem encolhe.

Figura 9 - Exemplo de erosão



Fonte: Wolfram Alpha

2.8 Validação dos dados

Esta seção tem como objetivo clarificar e exemplificar a maneira de como serão validados os dados pós teste. Informações adicionais de como os testes foram feitos na prática podem ser visualizados na seção 4.2.

2.8.1 Precisão e *recall*

Os conceitos propostos por Powers (2011) de precisão e *recall* serão usados para validação dos dados uma vez que é possível determinar o melhor de conjunto de parâmetros que o trabalho poderá utilizar para detectar o maior número de buracos. Precisão indica quantos buracos foram identificados pelo método e se estão realmente corretos, o *recall* indica quantos buracos do total foram identificados.

As equações 4 e 5 mostram como a precisão e o *recall* foram calculados ao longo do projeto. Os valores utilizados nas equações são retirados de uma tabela de confusão que faz com que os dados fiquem mais visuais.

$$\text{Precisão } (\%) = \frac{VP}{VP + FP} * 100 \tag{4}$$

$$\textit{Recall}\ (\%) = \frac{VP}{VP + FN} * 100 \tag{5}$$

Onde

- VP, quantidade de verdadeiros positivos;
- FP, quantidade de falsos positivos;
- FN, quantidade de falsos negativos.

2.8.2 Matriz de Confusão

Uma matriz de confusão é uma matriz que relaciona dados reais com valores preditos de forma que fique fácil a visualização do conjunto que está sendo analisado. Um exemplo de matriz de confusão pode ser visualizado na Tabela 1.

Tabela 1 - Exemplo de matriz de confusão

		Esperado	
		Buraco	**Não é buraco**
Previsto	**Buraco**	14	10
	Não é buraco	4	12

Fonte: O autor, 2019

Os dados apresentados na Tabela 1 podem ser interpretados como:

- O algoritmo detectou 14 casos de buracos e que realmente eram buracos. Também chamado de verdadeiro positivo.
- O algoritmo detectou 4 casos de "não buracos"que na verdade eram de fato buracos. Chamados de falsos negativos.
- O algoritmo detectou 10 casos de buracos onde na verdade não eram buracos. Chamados de falsos positivos.
- O algoritmo não detectou buracos em 12 casos, onde realmente não existiam buracos. Chamados de verdadeiros negativos.

3 METODOLOGIA

Nesta etapa serão detalhados os procedimentos realizados ao longo do trabalho, bem como a configuração utilizada, os testes realizados e o método de validação aplicado para a detecção correta de buracos, objetivando o menor índice de erros.

3.1 Equipamento utilizado

Para a coleta das imagens e posteriormente dos vídeos, utilizou-se uma câmera do tipo *dashcam* de modelo: GD-80, da fabricante GiiNii. A fim de obter o máximo de detalhes na gravação, a resolução da câmera foi alterada para 1920x1080, com filmagens em HD. Com o intuito de diminuir o tamanho do arquivo gerado ao final da gravação, os áudio das filmagens foram desativados pois não seriam de uso no trabalho.

A câmera também foi configurada para funcionar como uma *webcam*, de modo a não escrever o que foi gravado diretamente em um *micro sd*. Esta alteração se faz necessária uma vez que deseja-se detectar os buracos em tempo real para posterior consulta na base de dados com as coordenadas de onde tal anomalia foi detectada.

Além dos passos mencionados, o processamento do código para a detecção de buracos é realizado por um *Raspberry Pi 3*, (Raspberry Pi, 2016), que recebe os dados diretamente da câmera através de uma conexão USB.

A fim de que o algoritmo proposto funcione com o máximo de aproveitamento, é necessário seguir uma premissa: a câmera deve estar sempre posicionada na parte superior central do para-brisa do veículo, conforme a Figura 10 mostra. Deste modo, a imagem da rodovia pode ser captada como um todo, funcionando de acordo com a área de interesse definida pelo algoritmo construído.

Figura 10 - Câmera montada no centro do para-brisa do veículo

Fonte: O autor, 2019

3.2 Configuração do ambiente

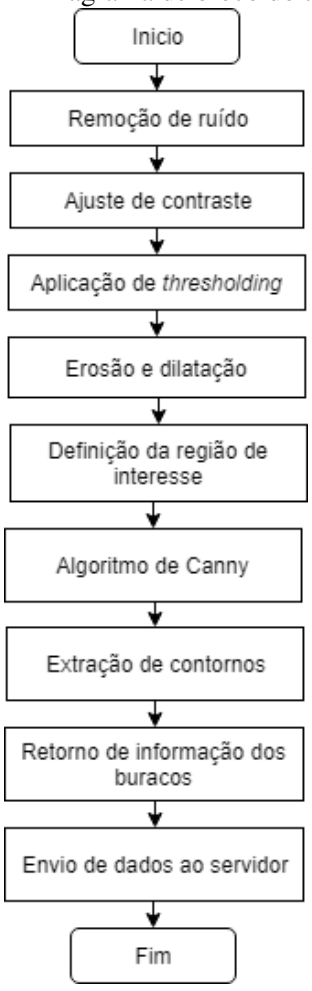
Além da utilização da câmera montada ao painel do carro, é necessário configurar o ambiente de modo a processar e enviar os dados capturados de forma mais rápida possível para uma base de dados local. Que posteriormente será utilizada para a montagem de um mapa.

O *Raspberry* inicialmente deve ser configurado para funcionar com o algoritmo, deste modo, as bibliotecas utilizadas: *matplotlib*, *numpy* e *opencv* devem ser instaladas. Os dados serão salvos em um arquivo do tipo *JSON*, que permite leitura e escrita de dados de forma fácil e organizada. Além, o módulo GPS também deve ser configurado para funcionar com o *Raspberry*, os dados de latitude e longitude irão ser enviados a base de dados posteriormente.

3.3 Diagrama de blocos do algoritmo de detecção de buracos

A Figura 11 representa o passo a passo do algoritmo que foi projetado e construído ao longo do projeto. As seções subsequentes explicam cada passo de forma mais elaborada.

Figura 11 - Diagrama de bloco do algoritmo



Fonte: O autor, 2019

3.3.1 Transformação da imagem para preto e branco

A primeira etapa consiste em converter a imagem em preto e branco pois a utilização de cor não será necessária para os processos futuros que o algoritmo propõe. Além, remover cor da imagem original permite que o processamento fique mais veloz, diminuindo assim o tempo necessário para detecção de buracos.

A conversão da imagem é feita de acordo com as propostas do NTSC, conforme citado na seção 2.2.1.

3.3.2 Remoção de ruído

Esta etapa consiste na aplicação do filtro gaussiano para a diminuição de ruído da imagem. O *kernel* aplicado neste passo tem tamanho 5x5 e retorna uma imagem com menos detalhes conforme mencionado na seção 2.5.

3.3.3 Aplicação de *thresholding*

Neste momento, o método de *thresholding* adaptativo proposto por Otsu é utilizado. Este tem o objetivo de fazer a binarização da imagem de acordo com seu histograma, permitindo que os resultados sejam os mais otimizados possíveis mesmo se a imagem utilizada detiver muitas variações.

3.3.4 Erosão e dilatação

O algoritmo de erosão é utilizado para diminuir os buracos que foram detectados no passo anterior. A dilatação por sua vez é aplicada para se livrar de falsos positivos, ou seja, áreas da imagem muito pequenas que não são consideradas buracos e não foram descartadas no processo anterior.

Com os conhecimentos citados em 2.7.1 e 2.7.2 é possível chegar a conclusão que deve se dilatar mais do que erodir uma vez que a erosão pode criar falsos positivos e ter valores iguais para ambas operações não resultaria em nenhum processamento produtivo.

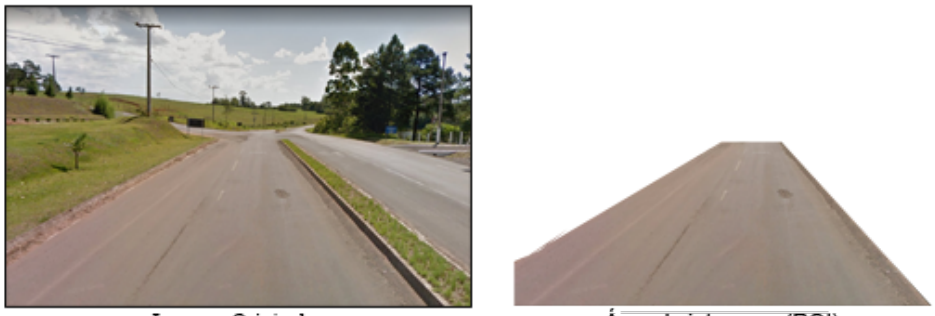
O algoritmo aplica primeiramente a erosão, onde o *kernel* para este é de 5x5. Após é aplicada a dilatação com um *kernel* relativamente maior, de 7x7.

3.3.5 Definição da região de interesse

A região de interesse (ROI) é determinada por uma área triangular sobre o capô do veículo. O única premissa para uma correta detecção de buracos é posicionar a câmera no centro do para-brisa conforme exemplificado na Figura 12. Além disso, a imagem gerada pela câmera

deve sempre manter a mesma resolução, uma variação pode causar resultados inconsistentes.

Figura 12 - Área de interesse



Fonte: O autor, 2019

3.3.6 Aplicação de Canny

O algoritmo de Canny é aplicado sobre o ROI para obter-se áreas da imagem que são candidatas a serem buracos conforme a Figura 6 representa. O algoritmo de Canny cria contornos sobre possíveis objetos fazendo com que seja possível a extração destes contornos futuramente.

3.3.7 Extração de contornos

Este passo está ligado à aplicação do algoritmo de Canny. Os contornos encontrados no passo anterior são extraídos nesta etapa. A extração de contornos é feita de forma inteligente, ignorando pontos repetitivos salvando, assim, memória e fazendo com que esta etapa fique mais ágil.

Este passo também conta com uma determinação de hierarquia de buracos, onde cada buraco possui um valor de 1-5. Este nível de hierarquia depende de sua posição. Então, se um contorno está dentro de outro contorno este terá uma hierarquia de valor 2, fazendo com que o algoritmo considere menos seu contorno.

3.3.8 Retorno de informações de buracos

Após a verificação de buracos é necessário receber os dados verdadeiros e marcá-los na imagem. Para fins visuais, o algoritmo desenha um quadrado ao redor do buraco detectado conforme a Figura 13 mostra.

Figura 13 - Imagem final gerada pelo algoritmo

Fonte: O autor, 2019

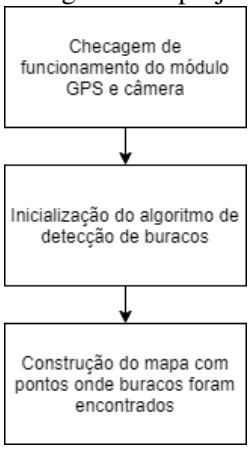
3.3.9 Envio de dados à base de dados

Dados da área aproximada do buraco e localização de onde o buraco foi detectado são enviadas a uma base de dados que é utilizada para posterior construção de uma aplicação *web* na qual é possível visualizar o lugar em que estão localizados os buracos em um mapa interativo e de fácil entendimento.

3.4 Diagrama de blocos do projeto completo

A Figura 14 mostra o diagrama de blocos do projeto completo, em que foram integradas as partes de hardware e software. O diagrama completo do projeto é muito simples, possuindo três passos apenas. Cada um deses passos é crítico para o correto e total funcionamento da aplicação. Cada passo é descrito nas subseções seguintes.

Figura 14 - Diagrama do projeto completo



Fonte: O autor, 2019

3.4.1 Checagem de funcionamento do módulo GPS e câmera

Nesta etapa é realizada uma checagem prévia de funcionamento da câmera com o GPS. Se o algoritmo receber uma resposta inicial de ambos, é detectado que está tudo certo com o sistema e o algoritmo de detecção de buracos pode ser iniciado.

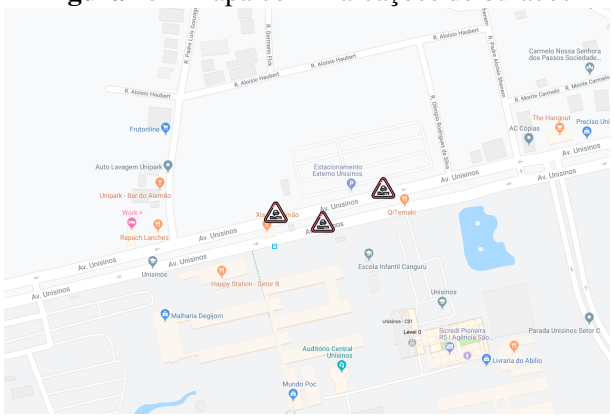
3.4.2 Inicialização do algoritmo de detecção de buracos

O algoritmo exemplificado na seção 3.3 começa a ser executado.

3.4.3 Construção do mapa com pontos onde buracos foram encontrados

Os dados de geolocalização são obtidos pela módulos GPS conectado ao *Raspberry Pi*. Estes dados são salvos em um arquivo *JSON*, na memória interna do embarcado. O arquivo com as coordenadas deve então ser utilizado para que seja possível realizar a construção de um mapa, conforme a Figura 15 demonstra.

Figura 15 - Mapa com marcações de buracos



Fonte: O autor, 2019

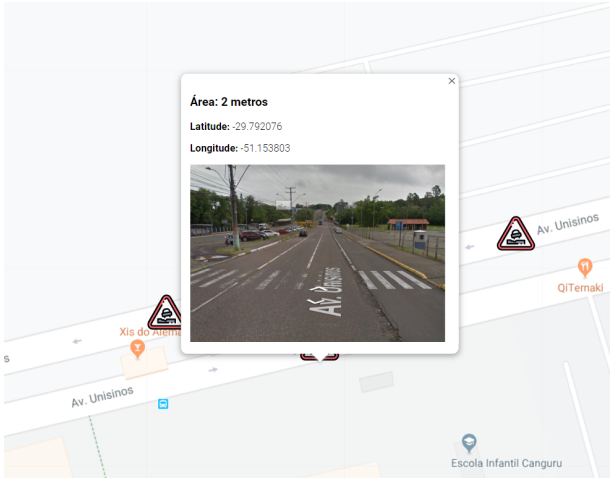
O mapa é baseado no *Google Maps* e possui marcações dos buracos encontrados. As marcações podem ser detectadas pelo ícone mostrado na Figura 16.

Figura 16 - Ícone para aviso de buraco

Fonte: Flat Icon

Caso o usuário necessite, ele pode clicar em uma das demarcações para mostrar mais dados sobre o buraco. A área estimada, a latitude, a longitude do buraco e uma imagem do momento em que foi detectado o buraco são exemplos de informações que podem ser visualizadas ao clicar no ícone anteriormente mostrado. A Figura 17 mostra três buracos detectados em São Leopoldo, em frente a Universidade do Vale do Rio dos Sinos. Em um deles é possível visualizar informações adicionais.

Figura 17 - Informações de buraco selecionado



Fonte: O autor, 2019

O desenvolvimento do mapa foi feito utilizando a tecnologia *React*, proporcionada pelo *Facebook*. A integração com o *Google Maps* foi realizada utilizando uma chave privada fornecida pelo próprio sistema do *Google*.

4 RESULTADOS OBTIDOS

Nas seções seguintes serão apresentados os resultados obtidos, bem como os testes realizados para verificar a validade do algoritmo desenvolvido.

4.1 Testes preliminares

Com o objetivo de testar o algoritmo foram utilizadas diversas imagens. As fontes de dados foram as mais diversas: filmagens realizadas com a câmera mencionada em diferentes trajetos, capturas de tela feitas através do *Google Maps*, fotos tiradas dos mais diversos ângulos pelo autor e pesquisas básicas por palavras-chaves em motores de busca.

No total, obteve-se um conjunto de dados de 35 fotos. Neste conjunto existem imagens dos mais diversos locais, todas com asfalto, em diferentes horários do dia e em diferentes condições climáticas. A Figura 18 mostra algumas das imagens obtidas. O objetivo de testar com um *dataset* variado é validar em quais momentos do dia o algoritmo se comporta de maneira esperada.

Figura 18 - Parte das imagens selecionadas para testes preliminares

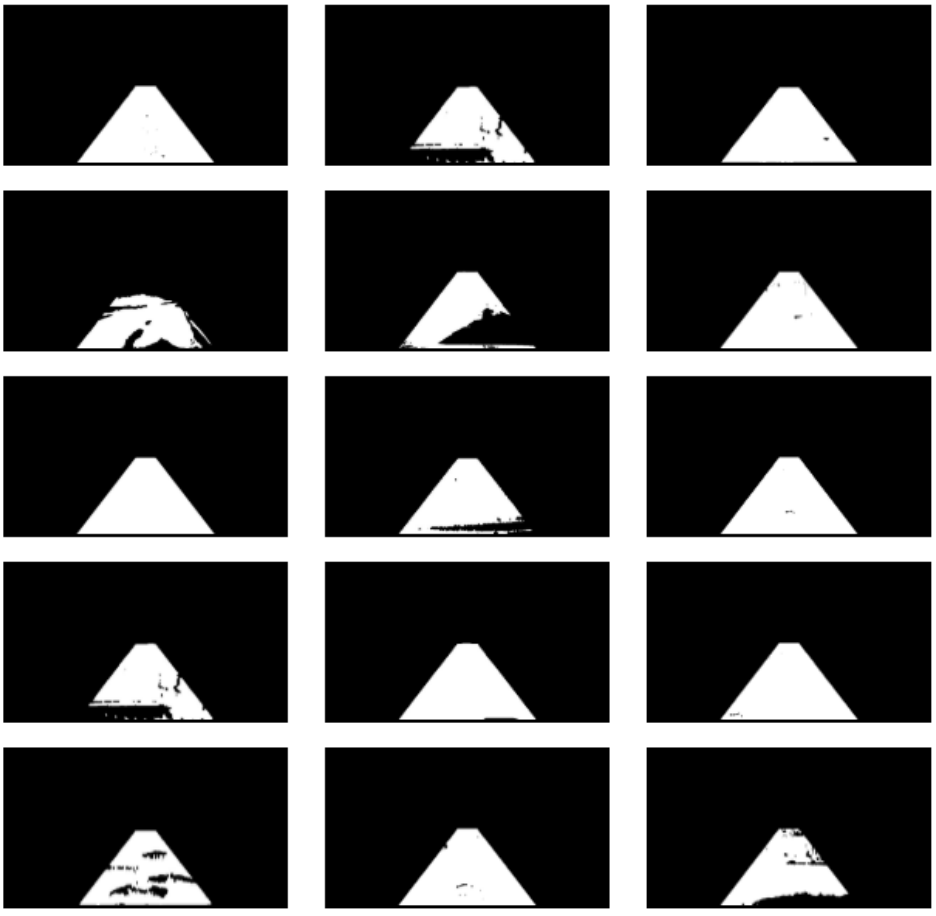
Fonte: O autor, 2019

Para cada uma das imagens anteriores foi aplicado um algoritmo inicial de teste para verificar a eficácia do processo de manipulação de imagem mencionado. O algoritmo aplicado envolve todos os passos citados anteriormente, ele utiliza uma área de interesse da região desejada, sem qualquer tipo de análise para se obter a melhor *ROI*.

Fazendo uma análise mais completa do conjunto de imagens apresentado, é possível visualizar que 11 imagens são em horários ensolarados do dia, sem sombras. Duas imagens foram tiradas em momentos noturnos, uma imagem foi retirada em um dia de neblina e, uma terceira, foi retirada em um dia de chuva.

Para cada imagem foi aplicado o algoritmo descrito anteriormente. Todos os passos foram executados a fim de verificar e validar a presença de buracos nas imagens. A Figura 19 mostra o *dataset* de imagem com o algoritmo aplicado.

Figura 19 - Testes preliminares em parte do *dataset* de Figuras



Fonte: O autor, 2019

Os resultados obtidos com a aplicação do algoritmo evidenciam os buracos em determinados momentos. Para ambientes com boa luminosidade, nota-se que os resultados foram satisfatórios uma vez que grande parte dos buracos foram detectados. Ambientes com baixa luminosidade mostram um deficit na detecção de buracos, uma vez que a imagem irá ter seu histograma de forma menos uniforme.

4.2 Validação

A validação do algoritmo será feita de forma manual, onde o autor visualizará a imagem pós processada e verificará se o algoritmo detectou o buraco com sucesso ou não. Ainda existirão situações onde o algoritmo pode detectar buracos e este não existir, ou o algoritmo não detectar o buraco, e este existir. Tendo estes possíveis caso em mente serão realizados cálculos de precisão e *recall*.

Com os dados de validação obtidos, uma matriz de confusão será montada de modo que sejam visualizados valores reais de positivos verdadeiros, falsos positivos, falsos negativos e negativos verdadeiros. É importante salientar que, na avaliação final da matriz de confusão, os casos negativos verdadeiros serão ignorados uma vez que estes casos avaliam momentos em que o algorítimo não detectou buracos, e que de fato não existiam buracos na pista. Este dado será descartado uma vez que deseja-se obter informações sobre a detecção de possíveis buracos, e não o contrário.

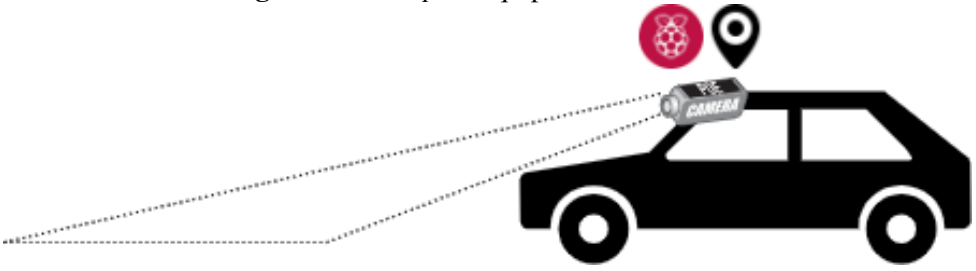
Um vídeo de um trajeto precário deverá ser utilizado para validação do algoritmo criado, o trajeto a ser escolhido deve possuir falhas na via com diferentes características. A velocidade do carro no momento da gravação deverá ser de até 80km/h.

4.3 Discussão dos resultados

Com o objetivo de validar o trabalho, o trajeto com sentido Carlos Barbosa a São Vendelino através da RS-446 foi realizado. Este trecho foi selecionado devido ao seu histórico de buracos, sendo que, neste trecho, eles possuem diversos tamanhos, fazendo com que o algoritmo implementado trabalhe com potencial máximo. O veículo utilizado neste trajeto foi um *Hyundai HB20*.

O *setup* montado no carro foi colocado conforme a Figura 20. A câmera foi instalada na parte superior do para-brisa, junto a ela, foram conectados o GPS e o *Raspberry Pi*. Por meio dos resultados obtidos, realizando o trecho foi possível realizar análises extensivas com o intuito de validar o algoritmo criado para detecção de buracos.

Figura 20 - *Setup* de equipamentos no carro



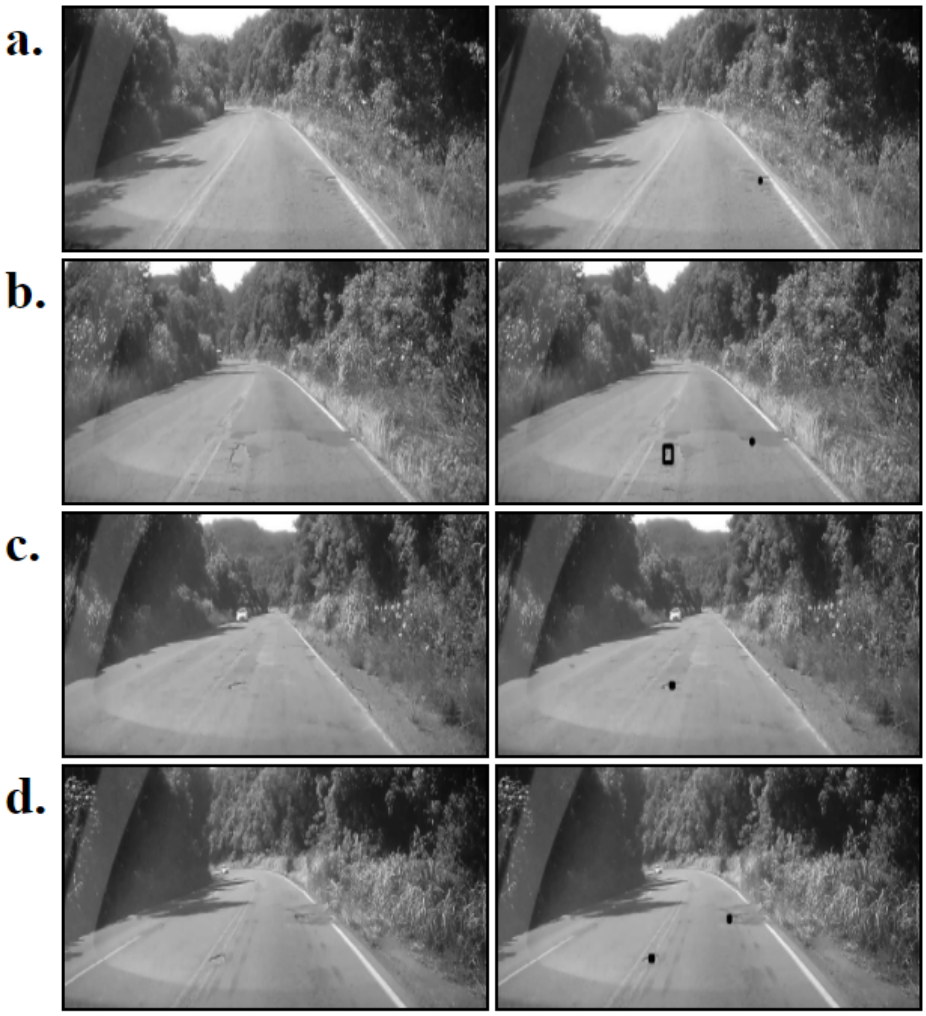
Fonte: O autor, 2019

Do trajeto mencionado anteriormente foi retirado um trecho considerado crítico, onde vá-

rios buracos estão presentes. Neste trecho foi aplicado o algoritmo e posteriormente feita a análise necessária. Alguns *frames* da gravação foram retirados e classificados de acordo com sua dificuldade de deteçcão. Os *frames* podem ser visualizados nas Figuras 21 e 22.

As Figuras 21 e 22 mostram momentos do trecho proposto onde foram detectados buracos. A Figura 21 apresenta um conjunto de imagens em que os buracos podem ser facilmente detectados. A detecção dos buracos pode ser visualizada como um ponto na rodovia. Cada ponto represente um buraco.

Figura 21 - Cenário fácil para detecção de buracos



Fonte: O autor, 2019

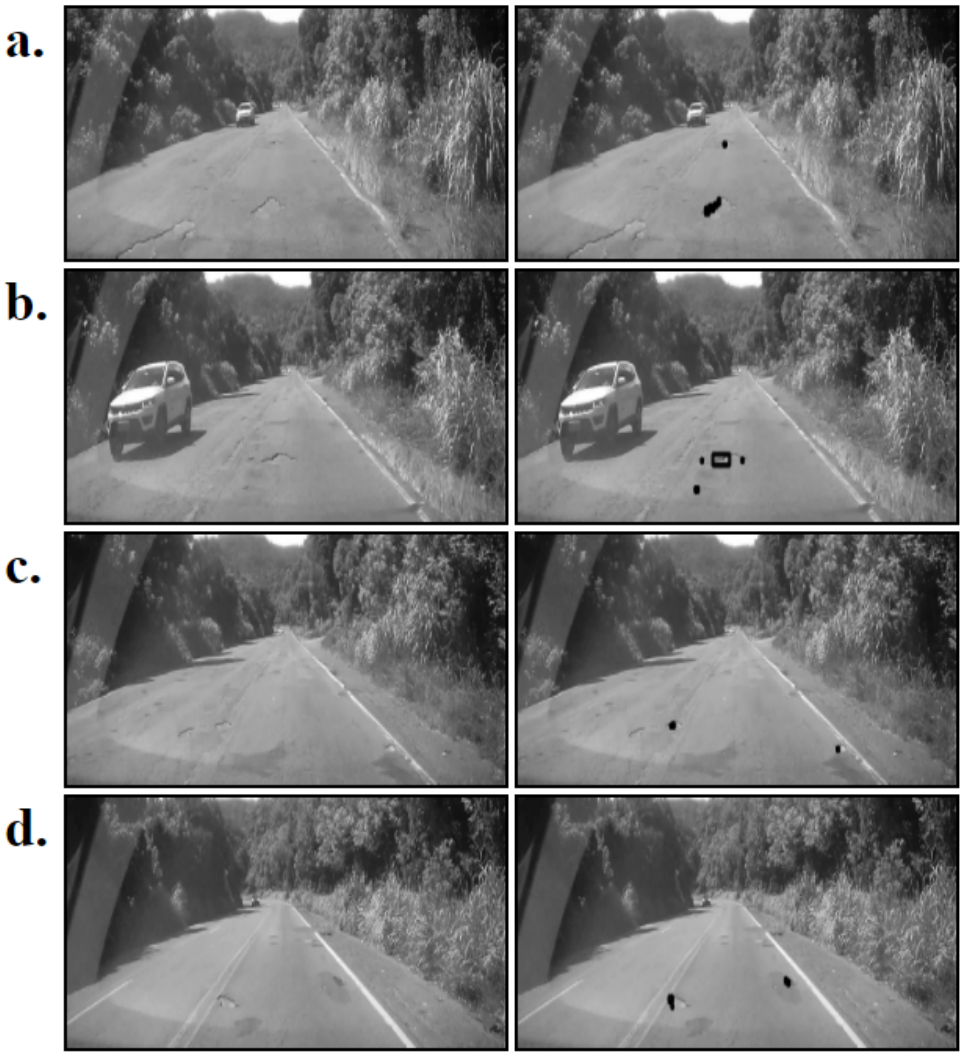
Um buraco facilmente identificável pode ser definido como uma mancha no asfalto que destoa do resto da pista. As situações a) e c) da Figura 21 apresentam casos em que existem apenas um buraco na via e este era facilmente identificável.

Os itens b) e d) retratam casos em que haviam dois buracos na via e estes também eram facilmente identificáveis uma vez que seu tamanho e cor tinham as características necessárias.

A Figura 22 apresenta um conjunto mais complicado, no qual existem múltiplos buracos e algumas rachaduras visíveis. Aqui, o algoritmo procura apenas casos onde existam de fato

buracos.

Figura 22 - Cenário difícil para detecção de buracos



Fonte: O autor, 2019

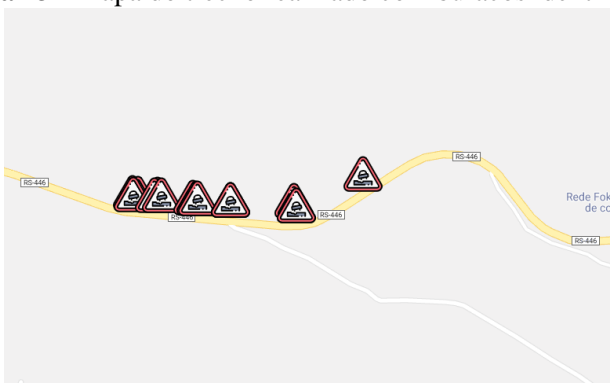
Nas situações a) e b) da Figura 22, os buracos não são tão visíveis e não possuem características tão fortes para serem identificados como buracos. Como pode ser visto, o algorítimo não tem problemas para classificá-los, a detecção neste caso se dá pelas áreas mais escuras que destoam do resto da pista. Buracos fora do *ROI* definido são ignorados.

As situações c) e d) mostram casos onde os buracos estavam no limite do *ROI* definido. Revelando que o algoritmo conseguiu separar o delimitador de área de interesse com o buraco. Em situações que não existem buracos o algoritmo se porta de forma esperada, não marcando alarmes falsos.

Para os buracos detectados ainda foi possível a criação de um mapa que mostra a localização as saliências encontradas. A Figura 23 mostra os buracos encontrados no referido trecho. O mapa se comporta conforme citado anteriormente, onde o usuário pode clicar no ícone para receber mais informações conforme o requisitado, estas informações incluem latitude, longitude

e uma foto do momento em que o buraco foi detectado.

Figura 23 - Mapa do trecho realizado com buracos identificados



Fonte: O autor, 2019

Sombras de árvores sob a rodovia são, muitas vezes, detectadas como buracos devido a sua sombra destoar da coloração da via. Esta má interpretação ocorre em virtude do algoritmo de *OTSU*, que acaba aceitando ruídos na área de interesse que estejam abaixo do *thresholding* escolhido. Com o objetivo de evitar erros desta origem o algoritmo aplica um verificador em cascata, que checa a área e a forma geométrica do que foi detectado, para finalmente identificar o buraco.

Uma análise foi feita para montar a matriz de confusão. O vídeo original, no qual os buracos foram mapeados manualmente, foi utilizado como base. Como comparativo, o mesmo vídeo com o algoritmo aplicado foi utilizado, onde os buracos ficaram marcados, conforme as imagens anteriores demonstram.

A tabela 2 mostra os resultados obtidos no formato de matriz de confusão.

Tabela 2 - Matriz de confusão para vídeo utilizado na validação

		Esperado	
		Buraco	**Não é buraco**
Previsto	**Buraco**	35	8
	Não é buraco	4	-

Fonte: O autor, 2019

O algoritmo detectou buracos corretamente 35 vezes. Foram 8 vezes que ele detectou como buraco algo que não era, e, 4 vezes o algoritmo falhou em detectar buracos totalmente. Tendo os dados mencionados, pode-se calcular a precisão do algoritmo, que foi de 81%. O *recall* calculado foi de 89%, mostrando que grande parte dos buracos do vídeo original foram de fato mapeados.

Um grande adendo para a análise final é a posição da câmera no momento da gravação. Por meio dos testes foi possível verificar melhores resultados em vídeos gravados com carros altos. Isto acontece pois buracos são geralmente mais escuros, uma foto tirada de cima possibilita o fácil reconhecimento deste pois a área destoa do resto da pista.

5 CONCLUSÃO

A manutenção das rodoviais é um problema que deve ser considerado e pensado constantemente. Enquanto buracos podem surgir por diversos motivos, é necessário pensar em maneiras de reparo destes para que eventuais acidentes não ocorram. A manutenção constante de buracos se faz necessária uma vez que a tendência da qualidade da via é piorar caso nada seja feito. Além de evitar acidentes, a detecção e mapeamento prévio de buracos pode dar mais visibilidade a administração local, deixando claro que trabalhos de manutenção estão sendo realizados.

Neste projeto um sistema para detecção e mapeamento de buracos foi projetado utilizando um *Raspberry Pi 3*, que roda um algoritmo customizado. Este algoritmo utilizou recursos de processamento de imagem para detectar buracos. As imagens para alimentar o algoritmo foram obtidas através de uma câmera instalada dentro do carro que foi conectada ao *Raspberry Pi 3*. Para realizar o mapeamento dos buracos um módulo GPS foi utilizado, o módulo envia a localização do buraco uma vez que o algoritmo detecta o mesmo. Com as localizações de buracos obtidas foi possível plotar um mapa com marcações dos buracos encontrados.

Descobriu-se que, para detecção de buracos utilizando processamento de imagens é interessante a utilização de filtros adaptativos, como exemplo, o *thresholding* de *OTSU*. Este filtro possibilita uma melhor análise da imagem, independente de variáveis externas. Outro grande ponto é a escolha do *ROI*, a área de interesse deve levar em consideração apenas a área da rodovia, ignorando o painel do carro. Ainda, é interessante levar em consideração a posição da câmera, uma vez que imagens tiradas mais do alto possibilitam a detecção de buracos de forma mais clara.

O algoritmo proposto funciona de forma otimizada em rodovias asfaltadas, em dias ensolarados. Testes mostraram uma detecção de buracos melhor neste ambiente. A validação ocorreu através de imagens retiradas do *Google Street View* e de *frames* de vídeos do *Youtube*.

O sistema proposto pode ser utilizado em veículos governamentais. Desta forma as autoridades vigentes podem ter informações constantes do estado das rodovias e, assim, podendo realizar as devidas manutenções em tempo hábil, prevenindo problemas. Veículos que realizam transporte público também poderiam utilizar o sistema.

Este trabalho apresentou um sistema para detecção de buracos baseado em processamento de imagem. Os resultados obtidos se mostram promissores uma vez que em 89% dos casos buracos foram detectados, e em 81% dos casos, buracos foram detectados de maneira correta.

5.1 Trabalhos futuros

Nesta seção serão apresentadas possíveis melhorias que podem ser feitas a partir deste trabalho, estas melhorias são ideias que ainda precisam ser avaliadas para posterior implementação.

Há a possibilidade de integração com o aplicativo *Waze*. O objetivo é enviar buracos mais rigorosos encontrados pelo algoritmo para a base de dados do *Waze*. Deste modo, motoristas

poderiam ser notificados da existência de anomalias nas ruas e evitar possíveis avarias nos veículos ou até acidentes.

Outra possibilidade seria enviar a localização de buracos detectados para servidor na nuvem em tempo real. Desta forma é possível manter um domínio na qual a comunidade que possui o dispositivo pode compartilhar locais onde existem buracos. A partir desta massa de dados ainda é possível realizar análises e descobrir cidades, bairros ou ruas onde a concentração de buracos é maior. Além disto, tendo esta massa de dados é possível construir algoritmos de predição que podem identificar épocas do ano em que buracos são mais comuns em rodovias, fazendo com que órgãos responsáveis possam criar planos de contingência antes que problemas ocorram.

REFERÊNCIAS

BARGHOUT, L.; LEE, L. **Perceptual information processing system**. US Patent App. 10/618,543.

BRADSKI, G.; KAEHLER, A. **Learning OpenCV**: computer vision with the opencv library. [S.l.]: "O'Reilly Media, Inc.", 2008.

BUZA, E.; OMANOVIC, S.; HUSEINOVIC, A. **Pothole detection with image processing and spectral clustering**. [S.l.: s.n.], 2013. 48–53 p.

CANNY, J. A computational approach to edge detection. **Readings in computer vision**, [S.l.], p. 184–203, 1987.

CHOLLET, F. **Deep Learning with Python**. [S.l.]: Manning, 2018.

DNIT. **Defeitos nos pavimentos flexíveis e semi-rígidos Terminologia**. 2003.

GEDRAITE, E.; HADAD, M. **Investigation on the effect of a Gaussian Blur in image filtering and segmentation**. [S.l.: s.n.], 2011.

GONZALEZ, R. C.; WOODS, R. E. **Processamento de imagens digitais**. [S.l.]: Edgard Blucher, 2000.

JO, Y.; RYU, S. Pothole detection system using a black-box camera. **Sensors**, [S.l.], v. 15, n. 11, p. 29316–29331, 2015.

KIM, T.; RYU, S.-K. System and method for detecting potholes based on video data. **Journal of Emerging Trends in Computing and Information Sciences**, [S.l.], v. 5, n. 9, p. 703–709, 2014.

KOCH, C.; BRILAKIS, I. Pothole detection in asphalt pavement images. **Advanced Engineering Informatics**, [S.l.], v. 25, n. 3, p. 507–515, 2011.

LI, Q.; YAO, M.; YAO, X.; XU, B. A real-time 3D scanning system for pavement distortion inspection. **Measurement Science and Technology**, [S.l.], v. 21, n. 1, p. 015702, 2009.

LIAO, P.-S.; HSU, B.-C.; LO, C.-S.; CHUNG, P.-C.; CHEN, T.-S.; LEE, S. K.; CHENG, L.; CHANG, C.-I. **Automatic detection of microcalcifications in digital mammograms by entropy thresholding**. [S.l.: s.n.], 1996. 1075–1076 p. v. 3.

MEDNIS, A.; STRAZDINS, G.; ZVIEDRIS, R.; KANONIRS, G.; SELAVO, L. **Real time pothole detection using android smartphones with accelerometers**. [S.l.: s.n.], 2011. 1–6 p.

NIENABER, B. Detecting potholes using simple image processing techniques and real world footage. **African Transport Conference**, [S.l.], 2015.

OTSU, N. A threshold selection method from gray-level histograms. **IEEE transactions on systems, man, and cybernetics**, [S.l.], v. 9, n. 1, p. 62–66, 1979.

POWERS, D. M. Evaluation: from precision, recall and f-measure to roc, informedness, markedness and correlation. **Bioinfo Publications**, [S.l.], 2011.

PYTHON, S. F. **What is Python?** Disponível em: <https://www.python.org/doc/essays/blurb/>. Acesso em: 23 março 2019.

PYTHON, S. F. **History of the software**. Disponível em: <https://docs.python.org/2.0/ref/node92.html>. Acesso em: 23 março 2019.

PYTHON, S. F. **Python Package Index**. Disponível em: <https://pypi.org/>. Acesso em: 23 março 2019.

RASPBERRY PI, F. **Raspberry Pi 3 Model B+**. Disponível em: <https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf>. Acesso em: 26 maio 2019.

ROSIN, P. L.; HERVÁS, J.; BARREDO, J. I. **Remote sensing image thresholding for landslide motion detection**. [S.l.: s.n.], 2000. 10–17 p.

RYU, S.-K.; KIM, T.; KIM, Y.-R. Feature-based pothole detection in two-dimensional images. **Transportation Research Record: Journal of the Transportation Research Board**, [S.l.], n. 2528, p. 9–17, 2015.

SHAPIRO, L. G. . S. **Computer Vision**. [S.l.]: Prentice Hall, 2001. 137-150 p.

SHERLIN SURESH, D. A. J. Multilevel thresholding for color image segmentation using optimizaion alghoritm. **International Journal of Scientific and Engineering Research**, [S.l.], 2017.

SOBEL, I. History and definition of the sobel operator. **Retrieved from the World Wide Web.**, [S.l.], v. 1505, 2014.

SONG YUHENG, Y. H. Image Segmentation Algorithms Overview. **ArXiv**, [S.l.], 2017.

TIOBE. **Popularity of programming languages**. Disponível em: <https://www.tiobe.com/tiobe-index/>. Acesso em: 23 março 2019.

W. BURGER, M. B. **Principles of Digital Image Processing**: advanced method. [S.l.: s.n.], 2013. 30 p.