UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
INSTITUTO DE COMPUTAÇÃO
CURSO DE BACHARELADO EM CIÊNCIA DA COMPUTAÇÃO

FILIPE JOSÉ MACIEL RAMALHO

OBTENÇÃO DE BORDAS DE OBJETOS EM UMA IMAGEM

Implementação de uma versão do Canny Edge Detector

RIO DE JANEIRO
2023

FILIPE JOSÉ MACIEL RAMALHO

OBTENÇÃO DE BORDAS DE OBJETOS EM UMA IMAGEM

Implementação de uma versão do Canny Edge Detector

Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientadora: Profa. Luziane Ferreira de Mendonça

RIO DE JANEIRO

2023

CIP - Catalogação na Publicação

R165o

Ramalho, Filipe José Maciel
Obtenção de bordas de objetos em uma imagem: implementação de uma versão do Canny Edge Detector / Filipe José Maciel Ramalho. -- Rio de Janeiro, 2023.
71 f.

Orientadora: Luziane Ferreira de Mendonça.
Trabalho de conclusão de curso (graduação) - Universidade Federal do Rio de Janeiro, Instituto de Computação, Bacharel em Ciência da Computação, 2023.

1. processamento de imagem. 2. filtro de imagem. 3. detecção de documentos. 4. detecção de bordas. I. Mendonça, Luziane Ferreira de, orient. II. Título.

Elaborado pelo Sistema de Geração Automática da UFRJ com os dados fornecidos pelo(a) autor(a), sob a responsabilidade de Miguel Romeu Amorim Neto - CRB-7/6283.

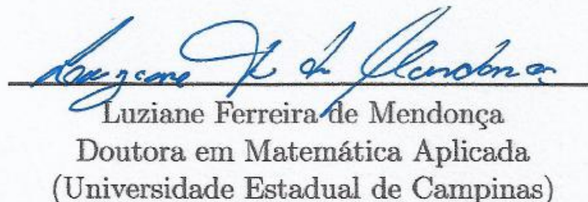
FILIPE JOSÉ MACIEL RAMALHO

OBTENÇÃO DE BORDAS DE OBJETOS EM UMA IMAGEM
Implementação de uma versão do Canny Edge Detector

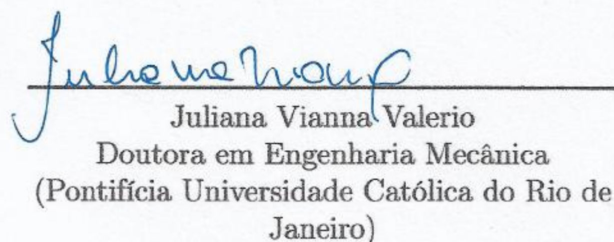
Trabalho de conclusão de curso de graduação apresentado ao Instituto de Computação da Universidade Federal do Rio de Janeiro como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em 05 de JUNHO de 2023

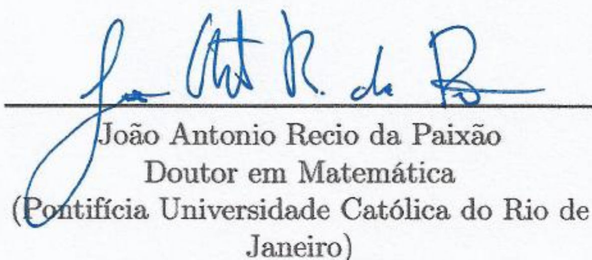
BANCA EXAMINADORA:



Luziane Ferreira de Mendonça
Doutora em Matemática Aplicada
(Universidade Estadual de Campinas)



Juliana Vianna Valerio
Doutora em Engenharia Mecânica
(Pontifícia Universidade Católica do Rio de Janeiro)



João Antonio Recio da Paixão
Doutor em Matemática
(Pontifícia Universidade Católica do Rio de Janeiro)

Dedicatória: Dedico este trabalho à minha família, que me deu o apoio necessário para superar os meus desafios.

AGRADECIMENTOS

Agradeço à UFRJ e aos professores do Departamento de Ciência da Computação por todo o conhecimento que me foi passado durante a minha graduação. À professora Luziane Ferreira de Mendonça, que me orientou durante esse trabalho, sempre me ajudando a esclarecer dúvidas, revisando textos e auxiliando com conteúdos importantes para complementar o trabalho.

Agradeço ao professor João Antonio Recio da Paixão pela atuação como orientador acadêmico me ajudando a chegar nesta etapa da minha vida.

Agradeço aos amigos Bruno Mançano Mattos, Erick Teixeira Pires e Fernando Fraga Coutinho Guimarães que me ajudaram a levantar a idéia do tema deste trabalho durante o período em que eu era estagiário na Stone Age.

Agradeço também à amiga Juliana Oliveira do Amaral que também ajudou a revisar o texto deste trabalho.

"We eat food that others have grown,
wear clothes that others have made,
live in houses that others have built.
The greater part of our knowledge and
beliefs has been communicated to us by
other people through the medium of language
which others have created"

Albert Einstein

RESUMO

Canny Edge Detector é um algoritmo multi-estágios que fornece a localização das bordas de uma imagem. Este projeto tem como objetivo uma implementação alternativa do *Canny Edge Detector* de modo que seu resultado passe a ser equiparável ao do filtro Sobel e assim possa ser utilizado em substituição à ele. Um projeto que teria um benefício com esta substituição seria o *Whiteboard It!*, que usa visão computacional para detectar quadros ou documentos. Em imagens, o *Whiteboard It!* utiliza como primeiro estágio de processamento o filtro Sobel para facilitar o processamento de bordas e conseguir extrair objetos quadrangulares das imagens. Neste trabalho será realizada a implementação e a comparação com uma aplicação de mercado do *Canny Edge Detector* proposto, o algoritmo do *OpenCV*. Por fim, conclui-se que o algoritmo implementado é um pouco mais lento que o de mercado (discutindo possíveis maneiras de diminuir esta diferença) mas com uma qualidade de resultado relativamente superior.

Keywords: processamento de imagem; filtro de imagem; detecção de documentos; detecção de bordas.

ABSTRACT

The Canny Edge Detector is a multi-stage algorithm that provides the locations of edges in an image. This project aims to provide an alternative implementation of the *Canny Edge Detector* in order that its results are comparable to the Sobel filter and can be used as a replacement. For it, an application that could benefit from this replacement is *Whiteboard It!*, which uses computer vision to detect whiteboards or documents. In images, *Whiteboard It!* uses the Sobel filter as the first stage to facilitate edge processing and to extract rectangular objects from images. The proposed *Canny Edge Detector* algorithm is implemented and compared to a market application, *OpenCV*. It is possible to conclude that the implemented algorithm is slightly slower than the market version (possible ways to reduce this gap are discussed), performing a relatively higher result quality.

Keywords: image processing; image filter; documents detection; borders detection.

LISTA DE ILUSTRAÇÕES

Figura 1 – Imagem utilizada para experiência - src.jpeg 21
Figura 2 – Imagem obtida utilizando Prewitt à direita - prewitt_r_out.jpeg . . . 24
Figura 3 – Imagem obtida utilizando Prewitt ao topo - prewitt_t_out.jpeg 25
Figura 4 – Imagem obtida utilizando Sobel à direita - sobel_r_out.jpeg 25
Figura 5 – Imagem obtida utilizando Sobel ao topo - sobel_t_out.jpeg 26
Figura 6 – Imagem obtida através de magnitudes das coordenadas polares dos vetores de intensidade 28
Figura 7 – Seleção de pixels segundo direção do gradiente 30
Figura 8 – Imagem extraida da seleção da figura 7. Linha de seleção para pixels mais intensos em verde 30
Figura 9 – Resultado do afinamento da figura 8 30
Figura 10 – Ilustração do modo de construção da máscara para o afinamento 31
Figura 11 – Resultado do afinamento na imagem de teste 39
Figura 12 – Gráfico representativo da relação entre os pixels e ambos os thresholds 42
Figura 13 – Imagem proveniente da etapa de afinamento, após multiplicação das intensidades por 50, antes do filtro *Hysteresis Thresholding* 51
Figura 14 – Imagem após filtro *Hysteresis Thresholding* e com multiplicação das intensidades por 50 realizada 51
Figura 15 – Imagem após filtro *Hysteresis Thresholding* sem multiplicação de intensidades 52
Figura 16 – Comparativo entre imagens. (a) Acima a imagem original. (b) Abaixo, à esquerda, após realizar o Canny, sem borramento. (c) Abaixo, à direita, com o filtro de borramento gaussiano 53
Figura 17 – Funções com adição de ruído gaussiano. (a) Acima, unidimensional. (b) Abaixo, bidimensional 54
Figura 18 – Visualização do funcionamento do filtro bilateral (PARIS et al., 2007). Pixel central indicado com seta em vermelho 56
Figura 19 – Resultado da aplicação do *Canny Edge Detector* utilizando filtro bilateral 57
Figura 20 – Aplicação do filtro da mediana diretamente sobre a imagem em escala de cinza 58
Figura 21 – Resultado da aplicação do *Canny Edge Detector* utilizando filtro da mediana 58
Figura 22 – Imagem resultante da matriz de ângulos produzida pelo algoritmo deste projeto. Vetor gradiente aponta da região mais clara para mais escura. 60
Figura 23 – Imagens comparativas entre os resultados de localidade dos pixels do *OpenCV* e do presente projeto (Primeira parte). 65

Figura 24 – Imagens comparativas entre os resultados de localidade dos pixels do *OpenCV* e do presente projeto (Segunda parte). 66
Figura 25 – Imagens de resultados do *OpenCV*, variando os limiares do *Histeresis Thresholding* 67

LISTA DE CÓDIGOS

Código 1 Arquivo principal com implementações simples das máscaras. 19
Código 2 Arquivo de definição das máscaras - masks.py 20
Código 3 Arquivo com definições para operações com imagens - image_operations.py. 23
Código 4 Arquivo principal com implementações para extração de vetores usando sobel_r e sobel_t 27
Código 5 Arquivo principal de início do processamento 32
Código 6 Arquivo em C++ que implementa o afinamento de bordas 33
Código 7 Arquivo que importa arquivo de afinamento compilado do C++ e faz mapeamento para funções e argumentos desse arquivo 35
Código 8 Classe Matrix que abstrai as posições das células para uso nos código em C++ 36
Código 9 Arquivo para integração do código em C++ para python - threshold/main.py. 42
Código 10 Arquivo contendo código para realização do threshold duplo - threshold.cpp. 43
Código 11 Código para execução de ambos os algoritmos, *OpenCV* e do presente projeto 61

SUMÁRIO

1 INTRODUÇÃO . . . 12
1.1 CONTEXTO MOTIVACIONAL . . . **12**
1.2 ESTRUTURAÇÃO . . . **14**

2 REALÇANDO IMAGENS COM CONVOLUÇÕES . . . 15
2.1 APLICAÇÕES . . . **16**
2.1.1 Operações de realce . . . 16
2.1.1.1 Filtro Sobel e Prewitt . . . **16**
2.2 IMPLEMENTAÇÃO SIMPLES . . . **18**
2.2.1 Obtendo coordenadas polares . . . 26

3 AFINAMENTO DAS BORDAS - NON MAXIMUM SUPPRESSION . . . 29
3.1 PROCESSANDO PIXELS . . . **29**
3.2 IMPLEMENTAÇÃO . . . **31**

4 HYSTERESIS THRESHOLDING . . . 40
4.1 IMPLEMENTAÇÃO DO FILTRO . . . **41**

5 FILTRO DE BORRAMENTO . . . 53
5.1 FILTRO BILATERAL . . . **55**
5.2 FILTRO DA MEDIANA . . . **56**
5.3 AVALIAÇÃO DOS FILTROS . . . **58**

6 ANÁLISE COMPARATIVA DOS RESULTADOS . . . 60
6.1 TEMPO DE PROCESSAMENTO . . . **61**
6.2 QUALIDADE VISUAL DOS RESULTADOS . . . **63**

7 CONCLUSÃO E CONSIDERAÇÕES FINAIS . . . 68

REFERÊNCIAS . . . 70

1 INTRODUÇÃO

A detecção de bordas é uma técnica fundamental no processamento de imagens e tem aplicações em diversas áreas, como visão computacional, robótica, automação industrial, medicina e muitas outras. Entre os vários algoritmos de detecção de bordas disponíveis, o Sobel é um dos mais conhecidos e utilizados. Ele é um filtro linear que calcula a magnitude do gradiente da imagem para identificar as bordas.

Uma aplicação que utiliza detecção de bordas é o *Whiteboard It!* (ZHANG; HE, 2002). Este projeto é resultado de uma pesquisa realizada pelo *Microsoft Reasearch* e tem por objetivo principal conseguir extrair dados em quadros e documentos, tais como aparelhos de digitalização, utilizando apenas uma câmera. O *Whiteboard It!* possui algumas etapas no seu processo e a primeira delas é o filtro Sobel.

No entanto, apesar de sua popularidade, o Sobel apresenta algumas limitações, como sensibilidade ao ruído, baixa precisão na detecção de bordas finas e espessas, e problemas com a orientação da borda. Para superar essas limitações, uma alternativa é a utilização do Canny Edge Detector, um algoritmo mais sofisticado e preciso.

O *Canny Edge Detector* é baseado em três etapas principais: suavização da imagem para evitar ruídos, detecção de gradientes e supressão não máxima. Ele também utiliza dois limiares, mínimo e máximo, para ajustar a sensibilidade do algoritmo de acordo com as características da imagem e do problema em questão. Esses recursos permitem uma detecção mais precisa e robusta de bordas em imagens.

Neste trabalho, propomos adaptar o *Canny Edge Detector* para substituir o Sobel em aplicações de detecção de bordas, como o *Whiteboard It!*. Acreditamos que essa adaptação pode trazer melhorias significativas em termos de precisão, robustez e eficiência na detecção de bordas.

Para isso, realizamos experimentos em diferentes tipos de imagens e comparamos os resultados com os obtidos pelo Sobel e também com uma aplicação de mercado do *Canny Edge Detector*. Esperamos que este estudo possa contribuir para o avanço da detecção de bordas em processamento de imagens e fornecer uma alternativa mais eficiente e precisa para o Sobel.

1.1 CONTEXTO MOTIVACIONAL

Nós estamos o tempo todo obtendo informações do mundo através dos nossos sentidos. Especialmente o sentido da visão é amplamente utilizado, uma vez que é nele que nos apoiamos para capturar impressões sobre a forma do mundo a nossa volta sem, necessariamente, precisar tocar nele.

No âmbito da computação, também é de muita importância que possamos extrair informações de imagens para que possamos automatizar tal processo e consigamos realizá-lo em larga escala. Assim, a área de pesquisa sobre processamento de imagens tem sido bastante importante para automatizarmos a obtenção destas impressões. Isto nos proporciona também outros benefícios, tais como filtros de imagens mais robustos, para melhorar a qualidade de imagem capturada por câmeras, ou a construção de robôs com a capacidade de visão semelhante a de seres vivos, como nós.

Segundo a publicação no site da FEBRABAN, aproximadamente 70% das operações bancárias realizadas no Brasil em 2021 foram realizadas por meio de *internet banking* ou celular (FEBRABAN, 2022). Este número ilustra a crescente demanda por tecnologia para realizar transações financeiras no Brasil.

Para garantir a segurança e veracidade das informações fornecidas durante qualquer operação, uma série de procedimentos são exigidos pelas instituições. Uma dessas medidas é a coleta de foto de documentos necessários para comprovar que quem está realizando determinada transação ou operação de cadastro é quem diz ser. A problemática por trás dessas operações são os algoritmos para ajustes do documento na tela para reconhecimento posterior. Muitas aplicações recorrem a redes neurais para executar esta tarefa, mas nem todos os dispositivos, principalmente móveis, estão aptos a executar tais algoritmos de maneira fluida. Como no Brasil, especialmente para a população mais pobre, o acesso a dispositivos com tais capacidades pode não ser tão fácil (PADRãO; YUGE, 2022), as empresas necessitam de construir algoritmos mais simples e que consumam menos recursos para tal tarefa, permitindo assim que um número maior de usuários adiram às suas soluções.

Pensando nesse contexto, é interessante uma alternativa aos algoritmos de inteligência artificial de reconhecimento de documentos. Mas, além da limitação social aqui descrita, existe também uma limitação de recursos da organização que quer implantar tal funcionalidade em seus produtos. É difícil, com as políticas de privacidade e proteção de dados recentemente implantadas, coletar imagens em massa para realizar o treinamento das inteligências que poderiam executar a tarefa de reconhecimento da localização de um documento em uma imagem. Tais políticas, como a LGPD[1], são muito importantes para o bem estar da sociedade e seria um erro gravíssimo violá-las. Por isso, organizações sérias podem ter mais dificuldade para construir tais algoritmos, sem desrespeitar a privacidade dos seus usuários.

Uma alternativa às soluções que envolvem Inteligência Artificial poderia ser o processamento de imagens utilizando algoritmo determinístico. Em geral, algoritmos determinísticos consomem menos recursos do dispositivo onde são executados e, apesar de precisarem de alguma massa de dados para testes, não necessitam delas para serem inicialmente

[1] Lei Geral de Proteção de Dados (TEMER et al., 2019). Importante mecanismo legislativo para garantir a privacidade das informações pessoais de todo cidadão brasileiro.

construídos, como os algoritmos de Inteligência Artificial.

Hoje existem algumas soluções de mercado para extrair imagens de documentos, como se fossem obtidas por um *scanner*, a partir de uma simples câmera de celular. O Google Lens e o Office Lens são duas dessas soluções e são amplamente utilizadas como substitutas de um *scanner*, mas poderiam ser utilizadas também para outros fins, como, por exemplo, gerar bibliotecas reutilizáveis por outros tipos de aplicações.

A problemática destas soluções de mercado, são, não só a natureza fechada de suas licenças de uso, como também a impossibilidade de reutilização como entrada para outras soluções. Infelizmente não é possível obter programaticamente os resultados do processamento dessas soluções, porque seus usos são exclusivamente pensados no consumidor final.

1.2 ESTRUTURAÇÃO

A partir da motivação explicitada, verificamos que alguns passos são importantes para a introdução e implementação de um algoritmo com múltiplos estágios. Estes passos tem o objetivo de realizar a implementação da solução aqui proposta e também apresentar e comparar seu resultado com uma aplicação de mercado.

Uma definição importante, é o que será chamado aqui de bordas de interesse. Bordas de interesse são todas aquelas que fazem parte da fronteira de um objeto com o outro, dentro da imagem. Bordas obtidas no meio da superfície de um objeto resultantes de texturas pertencentes a esta mesma superfície, serão menos importantes, uma vez que o objetivo é a construção de uma metodologia que poderá servir de entrada para o *Whiteboard It!*.

No próximo capítulo são apresentadas as definições, algoritmos e resultados da aplicação direta dos filtros de Sobel e Prewitt para a detecção de bordas. Esta é uma abordagem interessante para apresentar o funcionamento atual da primeira etapa do *Whiteboard It!* e deixar mais claro o motivo de substituí-la por uma versão alternativa.

Nos capítulos seguintes, estão presentes os detalhes sobre o *Canny Edge Detector* e a estratégia adotada para o afinamento e seleção das bordas de interesse (*Non Maximum Suppression* e *Hysteresis Thresholding*). No capítulo 5 são expostos os filtros de borramento estudados. Por fim, nos capítulos 6 e 7, temos os resultados computacionais encontrados e considerações finais.

2 REALÇANDO IMAGENS COM CONVOLUÇÕES

Convolução é uma operação matemática aplicada entre duas funções distintas. Esta operação é importante em vários campos da ciência, como estatística, engenharia e computação.

Na computação, a convolução é especialmente importante para processamento de imagens, uma vez que tem o poder de refinar ou acentuar determinadas características da imagem que se deseja estudar.

Os campos em que a convolução é aplicável são dos mais variáveis, com especial destaque em processamento de sinais para filtro de ruídos e manipulação de imagens, e por isso ela se torna muito importante. Para o contexto deste trabalho, ela será particularmente importante para suavização, com filtro de borramento, e extração de detalhes, como as bordas dos objetos presentes nas imagens.

Para entender esta operação, recorremos a literatura para buscar como ela está definida. Uma convolução (SONKA M.; HLAVAC; BOYLE, 2014) $f*g$ pode ser definida como uma operação entre duas funções f e g, denotada pela integral descrita na equação 2.1

$$(f*h)(t) \equiv \int_{-\infty}^{+\infty} f(\tau)h(t-\tau)d\tau \tag{2.1}$$

ou no caso bidimensional, como descrito na equação 2.2.

$$(f*h)(t,e) \equiv \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} f(\tau,\epsilon)h(t-\tau,e-\epsilon)d\tau d\epsilon \tag{2.2}$$

Uma imagem pode ser, então definida como uma função contínua $I(x,y)$ que indica a intensidade de cor no ponto denotado pela posição (x,y). Como na prática, uma imagem é capturada e armazenada de maneira discreta, interpreta-se que tal imagem capturada é na verdade uma função discreta, denotada também pela intensidade de cor nas coordenadas x e y, fornecidas como parâmetro. De maneira mais prática, tal função discreta representativa de uma imagem, é armazenada conforme seus pares de valores de domínio sendo a posição (x,y) e seus valores resultantes da função discreta como as intensidades $I(x,y)$, podendo também ser interpretada como a matriz abaixo.

$$I = \begin{bmatrix} I(0,0) & I(0,1) & I(0,2) & ... & I(0,j) \\ I(1,0) & I(1,1) & I(1,2) & ... & I(1,j) \\ I(2,0) & I(2,1) & I(2,2) & ... & I(2,j) \\ ... & ... & ... & ... & ... \\ I(i,0) & I(i,1) & I(i,2) & ... & I(i,j) \end{bmatrix} \tag{2.3}$$

Assim, poderemos então utilizar a convolução discreta em duas dimensões para operar nos valores desta imagem, da maneira descrita na equação 2.4, onde f_0 e f_1 são funções

que representam, respectivamente, os estados da imagem antes e depois da operação de convolução, O será definido como o espaço retangular de vizinhos do ponto $f(i, j)$ e g será uma segunda função, também discreta, a ser aplicada na operação de convolução com f, seguindo o formato $f * g$.

$$f_1(i, j) = \sum_{(m,n) \in O} f_0(i - m, j - n) g(m, n) \tag{2.4}$$

2.1 APLICAÇÕES

Quando os pontos de uma função são únicos, no contexto da convolução aplicada, ou seja, quando as funções que serão submetidas a convolução não são interdependentes, é possível dizer que não importa os valores de domínio escolhidos da primeira função, o contradomínio da segunda função será sempre o mesmo. Dessa forma, teremos valores fixados para a função g definida na equação 2.4, podendo portanto esta função ser definida, de maneira semelhante à imagem, como uma matriz de valores, tal qual a equação a seguir.

$$G = \begin{bmatrix} g(0,0) & g(0,1) & g(0,2) & ... & g(0,j) \\ g(1,0) & g(1,1) & g(1,2) & ... & g(1,j) \\ g(2,0) & g(2,1) & g(2,2) & ... & g(2,j) \\ ... & ... & ... & ... & ... \\ g(i,0) & g(i,1) & g(i,2) & ... & g(i,j) \end{bmatrix} \tag{2.5}$$

Apresentado o conceito da matriz G como função a ser superposta a uma imagem, podemos entender esta matriz como uma máscara de convolução. Algumas das principais operações de tratamento de imagem utilizam como base a técnica da convolução, aplicando uma máscara de convolução na matriz que representa a imagem a ser tratada.

2.1.1 Operações de realce

Algumas das operações de grande aplicabilidade são as operações de realce de imagem. Elas são importantes para extrair detalhes específicos da imagem, deixando-os mais evidentes do que o restante dela. Alguns exemplos de realces são os filtros Sobel e Prewitt (cujos resultados diferem em sutilezas por conta das similaridades dos pesos que compõem suas máscaras).

2.1.1.1 Filtro Sobel e Prewitt

O primeiro processamento realizado pelo projeto base deste trabalho, o *Whiteboard It!*, é justamente o filtro Sobel (BURGER; BURGE, 2009, 120–123). Uma característica deste filtro que é aproveitada pelo *Whiteboard It!* é o destaque de bordas dos objetos na imagem. Este filtro realiza isto através de uma máscara de convolução que tem como

objetivo somar os pixels vizinhos de um dos lados do pixel atual com o oposto dos pixels do lado contrário. Um exemplo de uma matriz associada a uma máscara do filtro Sobel pode ser encontrada na equação 2.6.

$$G = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.6}$$

Vale observar que os valores que estão armazenados na matriz que representam uma imagem estão associados à intensidade de cor da mesma na posição do pixel. Estes valores de intensidades de cor podem variar, dependendo da forma de representação desta imagem, como um intervalo de 0 a 255, valores binários (0 ou 1), etc. Dentro dessa imagem, é possível capturar suas bordas através da diferença de intensidade entre regiões fronteiriças. Esta diferença pode ser obtida através da derivada da função de formação destas intensidades.

De forma discreta, o valor desta derivada pode ser aproximado pela fração, cujo numerador é dado pela variação da intensidade de cor entre pixels e o denominador representa a distância entre esses pixels.

Dessa forma, é intuitiva a ideia de utilizar máscaras com valores positivos à direta (e acima) e valores negativos (e abaixo) representando os coeficientes do numerador para os pixels vizinhos ao central.

Um modo de exemplificar o ponto chave de funcionamento desta operação de forma simples pode ser vista no exemplo a seguir (COMPUTERPHILE, 2015b), que introduz o conceito do filtro Sobel utilizando um caso hipotético de uma imagem em apenas uma dimensão.

Seja uma imagem em tonalidade preto (representado por 0) e branco (representado por 1) com 1 pixel de altura e mais que 3 pixels de largura, e uma máscara com 1 pixel de altura por 3 de largura com valores [-1 0 1], se percorrermos esta máscara através da imagem, poderemos ter as seguintes combinações possíveis de trechos extraídos da imagem: [0 0 0], [0 0 1], [0 1 0], [0 1 1], [1 0 0], [1 0 1], [1 1 0], [1 1 1]. Com isto, podemos considerar os casos das vizinhanças ignorando o seu meio, uma vez que a máscara não aproveita o pixel central, ponderando-o com 0, reduzindo nossas possibilidades aos seguintes casos: [0 x 0], [1 x 1], [0 x 1], [1 x 0], com x podendo assumir os valores 0 ou 1. Assim, é possível testar cada caso como abaixo:

- Casos [0 x 0] e [1 x 1]:

  Quando aplicamos [-1 0 1] em [1 x 1], faremos o seguinte cálculo, de acordo com a definição de convolução: $resultado = 1 * (-1) + 1 * 1$. Assim, o resultado desta operação será 0. É interessante notar que o mesmo resultado aparecerá para qualquer vizinhança com valores extremos iguais, inclusive [0 x 0].

- Caso [0 x 1]:

  Quando a mesma metodologia é aplicada na matriz de vizinhos [0 x 1], teremos o seguinte cálculo: $resultado = 0 * (-1) + 1 * 1$. Neste caso, evidencia-se o pixel resultante com valor 1.

- Caso [1 x 0]:

  Por último, é possível aplicar convolução entre [1 x 0] e [-1 0 1], ficando o cálculo a seguir: $resultado = 1 * (-1) + 0 * 1$. Portanto o resultado será -1. Como a imagem do exemplo está limitada aos valores 0 e 1, deve-se, por hora, entender o valor -1 obtido como 0 ao exibir a imagem final processada.

A partir do exemplo é possível entender que o filtro Sobel destaca variações de intensidade de cor dos pixels vizinhos, o que é perfeito para deixar mais evidentes bordas em imagens. A desvantagem deste filtro para a detecção de borda é o fato de que ele detecta variações apenas de uma determinada tonalidade para outra, apenas na direção para o qual sua máscara foi configurada. Esta desvantagem pode ser visualizada no caso [1 x 0] do exemplo, em que há uma borda, porém ela não é evidenciada no pixel resultante.

Para máscaras como a equação 2.6, podemos ver que, diferente do simples exemplo, existem valores acima e abaixo do pixel central. Estes valores funcionam como pesos, da mesma forma que os valores horizontais. Na equação 2.6, no entanto, eles têm valores absolutos menores que os pixels vizinhos horizontais, o que garante que a influência deles no pixel final será menor. A diferença do filtro Prewitt, no entanto, é que ele possui os mesmos valores para todas as células de interesse na máscara, o que faz com que todos os respectivos vizinhos influenciem o pixel resultante na mesma proporção. Um exemplo de matriz Prewitt pode ser vista na equação 2.7

$$G = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.7}$$

É importante registrar que os pesos de uma máscara Sobel ou Prewitt, se não normalizados, podem influenciar drasticamente os valores de intensidades da imagem resultante destas operações. Este fato está intimamente relacionado com a ideia do resultado deste operador ser uma derivada, que, por conseguinte, é representada por meio da fração mencionada anteriormente.

2.2 IMPLEMENTAÇÃO SIMPLES

Para demonstração da teoria, ambos os algoritmos de Prewitt e Sobel foram implementados. Para a implementação, foi utilizada a linguagem Python e dentro dela, as bibliotecas "numpy" e "PIL" (*Python Imaging Library*).

O código se resume a declarar as matrizes das referidas máscaras e aplicar a operação de convolução entre elas e uma imagem fornecida. No código, a imagem fornecida é a imagem "src.jpeg" (figura 1), como definido no trecho de código `im = image_operations.imread('images/sr` do código 1, que será lida a partir do diretório "images" do projeto.

As máscaras utilizadas mantém pesos 1 e -1 para as bordas e 2 e -2 para os centros. A biblioteca de manipulação de arquivos de imagem utilizada ("PIL") possui a característica de, mesmo que os valores de imagem ultrapassem o intervalo de intensidades de 8 bits (de 0 a 255 em sistema decimal) considerar estes valores acima de 255 como sendo 255, e os valores abaixo de 0 como sendo 0. Por este motivo, e também para evidenciar as bordas extraídas, não foi realizada nenhuma normalização destas operações. Entretanto, neste documento, será realizado esta operação de normalização em etapas seguintes, para o correto processamento de algoritmos que utilizam como entrada o resultado do Sobel.

Código 1 – Arquivo principal com implementações simples das máscaras.

```
import image_operations
import masks


def mount_path(dirname, filename, ext):
    return "{}/{}.{}".format(dirname, filename, ext)


def jpeg_path(image_name):
    return mount_path("./images", "{}_out".format(image_name),
    ↪  "jpeg")


# Inicia processamento com imagem monocromática
def process_monochrome(image, kernel):
    image_gray = image_operations.to_gray(image)
    arr = image_operations.to_array(image_gray)
    arr2 = image_operations.conv(kernel, arr)
    return
    ↪  image_operations.to_gray(image_operations.from_array(arr2))


if __name__ == '__main__':
    im = image_operations.imread('images/src.jpeg')
```

```
    im_sobelr = process_monochrome(
        im,
        masks.sobelr()
    )

    im_sobelt = process_monochrome(
        im,
        masks.sobelt()
    )

    im_prewittr = process_monochrome(
        im,
        masks.prewittr()
    )

    im_prewittt = process_monochrome(
        im,
        masks.prewittt()
    )

    image_operations.imsave(im_sobelr, jpeg_path('sobel_r'))
    image_operations.imsave(im_sobelt, jpeg_path('sobel_t'))
    image_operations.imsave(im_prewittr, jpeg_path('prewitt_r'))
    image_operations.imsave(im_prewittt, jpeg_path('prewitt_t'))
```

É possível observar em cada uma das imagens de saída (figura 2, figura 3, figura 4 e figura 5) o que foi apresentado anteriormente sobre a detecção de bordas dos algoritmos de convolução utilizando os kernels Sobel e Prewitt. É possível visualizar também a diferença nos processamentos vertical e horizontal de cada um deles.

Código 2 – Arquivo de definição das máscaras - masks.py

```
import numpy as np


# Máscara Prewitt Direita.
def prewittr(factor=1):
```

Figura 1 – Imagem utilizada para experiência - src.jpeg

```
    return np.array([
        [-factor, 0, factor],
        [-factor, 0, factor],
        [-factor, 0, factor],
    ], dtype=np.float32)


# Máscara Prewitt Topo.
def prewittt(factor=1):
    return np.array([
        [factor, factor, factor],
        [0, 0, 0],
        [-factor, -factor, -factor],
    ], dtype=np.float32)


# Máscara Prewitt Fundo.
def prewittb(factor=1):
    return np.array([
        [-factor, -factor, -factor],
```

```
        [0, 0, 0],
        [factor, factor, factor],
    ], dtype=np.float32)


# Máscara Prewitt Esquerda.
def prewittl(factor=1):
    return np.array([
        [factor, 0, -factor],
        [factor, 0, -factor],
        [factor, 0, -factor],
    ], dtype=np.float32)


# Máscara Sobel Esquerda.
def sobell(center_factor=2, edge_factor=1):
    return np.array([
        [edge_factor, 0, -edge_factor],
        [center_factor, 0, -center_factor],
        [edge_factor, 0, -edge_factor]
    ], dtype=np.float32)


# Máscara Sobel Topo.
def sobelt(center_factor=2, edge_factor=1):
    return np.array([
        [edge_factor, center_factor, edge_factor],
        [0, 0, 0],
        [-edge_factor, -center_factor, -edge_factor],
    ], dtype=np.float32)


# Máscara Sobel Fundo.
def sobelb(center_factor=2, edge_factor=1):
    return np.array([
        [-edge_factor, -center_factor, -edge_factor],
        [0, 0, 0],
        [edge_factor, center_factor, edge_factor],
```

```
    ], dtype=np.float32)


# Máscara Sobel Direita.
def sobelr(center_factor=2, edge_factor=1):
    return np.array([
        [-edge_factor, 0, edge_factor],
        [-center_factor, 0, center_factor],
        [-edge_factor, 0, edge_factor]
    ], dtype=np.float32)
```

Código 3 – Arquivo com definições para operações com imagens - image_operations.py.

```
import numpy as np
from PIL import Image
from scipy import signal


# Executa a operação de convolução entre a matriz que representa a
↪ imagem
# e a matriz ou kernel de convolução.
def conv(kernel_matrix, convimg):
    return signal.convolve2d(
        convimg,
        kernel_matrix,
        mode='same',
        boundary='fill', fillvalue=1
    )


# Lê a imagem a partir de um dado caminho no computador
def imread(path):
    return Image.open(path)


# Exibe a imagem no exibidor de imagens padrão do sistema.
def imsave(image, image_path):
```

```
    return image.save(image_path)


# Converte um objeto de imagem em uma matriz
def to_array(image):
    return np.array(image)


# Converte uma matriz que representa uma imagem em um objeto de
↪   imagem
def from_array(array):
    return Image.fromarray(array)


# Converte uma imagem em escala de cinza, extraindo sua luminância
def to_gray(image):
    return image.convert('L')
```

Figura 2 – Imagem obtida utilizando Prewitt à direita - prewitt_r_out.jpeg

Figura 3 – Imagem obtida utilizando Prewitt ao topo - prewitt_t_out.jpeg

Figura 4 – Imagem obtida utilizando Sobel à direita - sobel_r_out.jpeg

Desta forma, com processamentos vertical ou horizontal e valores negativos desconsiderados, o processamento com Sobel ou Prewitt é, infelizmente, bastante subutilizado, limitando a quantidade de informações que é possível obter da imagem. De maneira ideal,

Figura 5 – Imagem obtida utilizando Sobel ao topo - sobel_t_out.jpeg

as informações obtidas sobre as bordas da imagem deveriam ser completas, e não apenas parciais, com informações sobre bordas na vertical ou bordas na horizontal. Para resolver este problema, podemos entender os resultados do filtro Sobel como uma matriz de gradientes de intensidade com coordenadas vertical (G_y) e horizontal (G_x), quando escolhe-se respectivamente as máscaras Sobel ao topo e à direita (as outras duas não foram escolhidas porque são apenas orientações inversas destas). Assim, podemos definir G_x e G_y nas equações 2.8 e 2.9 e portanto, podemos entender um valor qualquer definido em $G_x(i,j)$ e $G_y(i,j)$ como um vetor num plano cartesiano ($\vec{v} = (G_x(i,j), G_y(i,j))$)[1].

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} * Img \tag{2.8}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * Img \tag{2.9}$$

2.2.1 Obtendo coordenadas polares

Após entender os resultados de filtros Sobel vertical e horizontal como partes integrantes do mesmo conjunto de vetores, é possível converter estes vetores em coordenadas

[1] Esta definição pode ser facilmente entendida no vídeo “Finding the edges (sobel operator)”(COMPUTERPHILE, 2015b).

polares e assim obter as magnitudes absolutas de intensidade em cada ponto de todos os pixels de borda, bem como as orientações dos mesmos. Executamos, então, o código 4 para realizar esta operação. Utilizando a saída de magnitudes podemos reconstruir uma imagem com o resultado da aplicação do filtro em todas as direções da imagem e não apenas vertical e horizontal, o que nos permite, finalmente, observar todas as bordas da imagem de uma única vez na imagem na figura 6.

Código 4 – Arquivo principal com implementações para extração de vetores usando sobel_r e sobel_t

```
import numpy

import image_operations
import masks



def mount_path(dirname, filename, ext):
    return "{}/{}.{}".format(dirname, filename, ext)



def jpeg_path(image_name):
    return mount_path("./images", "{}_out".format(image_name),
    ↪  "jpeg")



# Inicia processamento para extração de vetores X e Y de intensidade
def process_vectors(image):
    image_gray = image_operations.to_gray(image)
    arr = image_operations.to_array(image_gray)
    sobelr = masks.sobelr()
    sobelt = masks.sobelt()
    gx_mat = image_operations.conv(sobelr, arr)
    gy_mat = image_operations.conv(sobelt, arr)
    return gx_mat, gy_mat



# Converte cada vetor em coordenadas polares
def getpolar(gx_mat, gy_mat):
    powergx = gx_mat ** 2
```

```
    powergy = gy_mat ** 2
    mag_mat = numpy.sqrt(powergx + powergy)
    angle_mat = numpy.arctan2(gy_mat, gx_mat)
    return mag_mat, angle_mat


if __name__ == '__main__':
    im = image_operations.imread('images/src.jpeg')
    gx, gy = process_vectors(im)
    mag, ang = getpolar(gx, gy)
    im2 = image_operations.from_array(mag)
    image_operations.imsave(
            image_operations.to_gray(im2),
            jpeg_path('mag_sobel')
        )
```

Figura 6 – Imagem obtida através de magnitudes das coordenadas polares dos vetores de intensidade

3 AFINAMENTO DAS BORDAS - NON MAXIMUM SUPPRESSION

Foi notado durante as experiências mencionadas no capítulo 2 que a aplicação dos filtros de Sobel e Prewitt gera como resultado imagens com bordas grossas e em geral um pouco desfocadas.

Seguindo a proposta de verificar a possibilidade de melhorar um ou mais aspectos das imagens processadas pelo *Whiteboard It!*, a observação mencionada no parágrafo anterior suscitou a busca por outras estratégias que gerassem resultados mais refinados. A abordagem proposta neste trabalho visa utilizar uma estratégia diferente para a detecção das bordas: o *Canny Edge Detector* (CANNY, 1986).

Para tanto, uma boa opção seria a de apenas utilizar bibliotecas que possuem essa rotina já previamente implementada, como por exemplo o *OpenCV* (TEAM, 2022) para tal propósito. Entretanto, é necessário, para substituição correta do Sobel, que o resultado do algoritmo a ser utilizado seja dado na forma de duas matrizes: Uma matriz com magnitudes das intensidades das bordas e uma segunda matriz com os ângulos direcionais de cada um dos pixels que as compõem, pois ambas informações serão necessárias para as etapas seguintes de detecção de documento.

Com isto em vista, e também com o propósito de entender o funcionamento do próprio *Canny Edge Detector*, será implementado esse algoritmo como parte do trabalho aqui descrito.[1]

O primeiro passo da execução do *Canny Edge Detector* é a execução do filtro Gaussiano (BURGER; BURGE, 2009, 97-98) para borramento da imagem.[2] O segundo passo é a obtenção das intensidades e gradientes da imagem que justamente é o produto da execução do filtro Sobel, e como esta etapa já foi discutida, apenas utilizaremos o que já foi realizado como base para a construção do *Canny Edge Detector*. O terceiro passo é conhecido como *Non-maximum Supression*, que consiste em remover pixels não desejados para o processamento, causando um afinamento das bordas. Por último, há um algoritmo que precisa ser executado para remover bordas remanescentes por um processo de corte de intensidades, conhecido como *Hysteresis Thresholding.*

3.1 PROCESSANDO PIXELS

O método de afinamento de bordas aqui desenvolvido consiste em decidir quais pixels da matriz de intensidade irão ou não ser removidos. Para esta decisão, a matriz de gradientes é necessária. O processo de afinamento consiste em localizar todos os pixels que são máximos locais de uma determinada borda em termos de intensidade, ou seja,

[1] A implementação realizada foi baseada em explicações de implementação do *OpenCV* e também a explicação do canal do Youtube Computerphile (COMPUTERPHILE, 2015a)

[2] No capítulo 5 serão abordados o funcionamento e a importância desta etapa do processamento

obter os pixels que tem maior intensidade que todos os seus vizinhos imediatos. Isto nos leva também a entender que precisamos de um meio para fornecer ao algoritmo qual é a borda com a qual ele precisa trabalhar para avaliar isto.

O algoritmo utilizará a direção do gradiente do pixel selecionado e, então, irá comparar com os vizinhos deste pixel a intensidade deles. Se o pixel selecionado for o mais intenso, ele será aceito na nova imagem e, se não, ele será recusado na nova imagem. As imagens 7, 8 e 9 ilustram este processo.

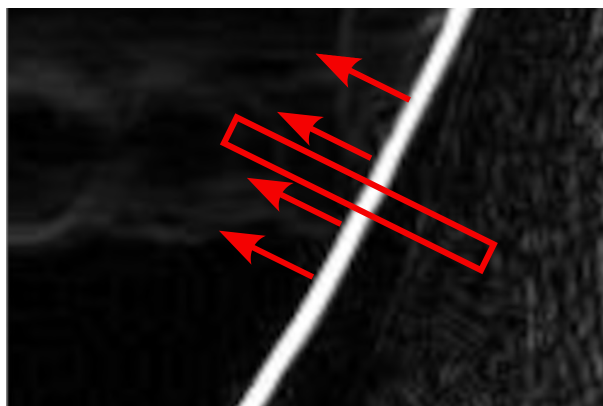


Figura 7 – Seleção de pixels segundo direção do gradiente

Figura 8 – Imagem extraida da seleção da figura 7. Linha de seleção para pixels mais intensos em verde

Figura 9 – Resultado do afinamento da figura 8

A metodologia aqui, novamente, será baseada na utilização da convolução para conseguir filtrar estes pixels máximos na imagem seguindo o gradiente das bordas. Como explicado, o filtro precisa decidir se o pixel será incluso ou não na nova imagem. Diferente das operações de filtro Sobel e Prewitt, este filtro precisa utilizar uma máscara que é dinâmica para cada pixel analisado, ou seja, para cada pixel da imagem, é necessário gerar uma nova máscara baseada no vetor gradiente da matriz de ângulos, na mesma posição do pixel observado.

Neste algoritmo de afinamento quanto maior o tamanho da máscara, mais precisa é a identificação do máximo local entre os pixels da borda, o que significa que, para bordas pequenas, a probabilidade de ocorrer dois ou mais máximos locais é maior, ou seja, uma borda tem mais chances de se dividir em duas outras menores. No entanto, para não ficar complexo, foi escolhida uma máscara pequena de tamanho 3 por 3. Foi avaliado que

é possível posteriormente remover estes erros na etapa seguinte do algoritmo do *Canny Edge Detector*.

A figura 10 ajuda a entender como esta construção da máscara se dá. O propósito é que a partir da submatriz 3×3 selecionada da imagem em uma determinada iteração da convolução, seja obtido o valor do ângulo, que corresponde à posição central desta submatriz, na matriz de ângulos. Com este valor do ângulo em mãos, então, descobre-se, seguindo a direção deste ângulo, quais serão os vizinhos do pixel central a serem analisados.

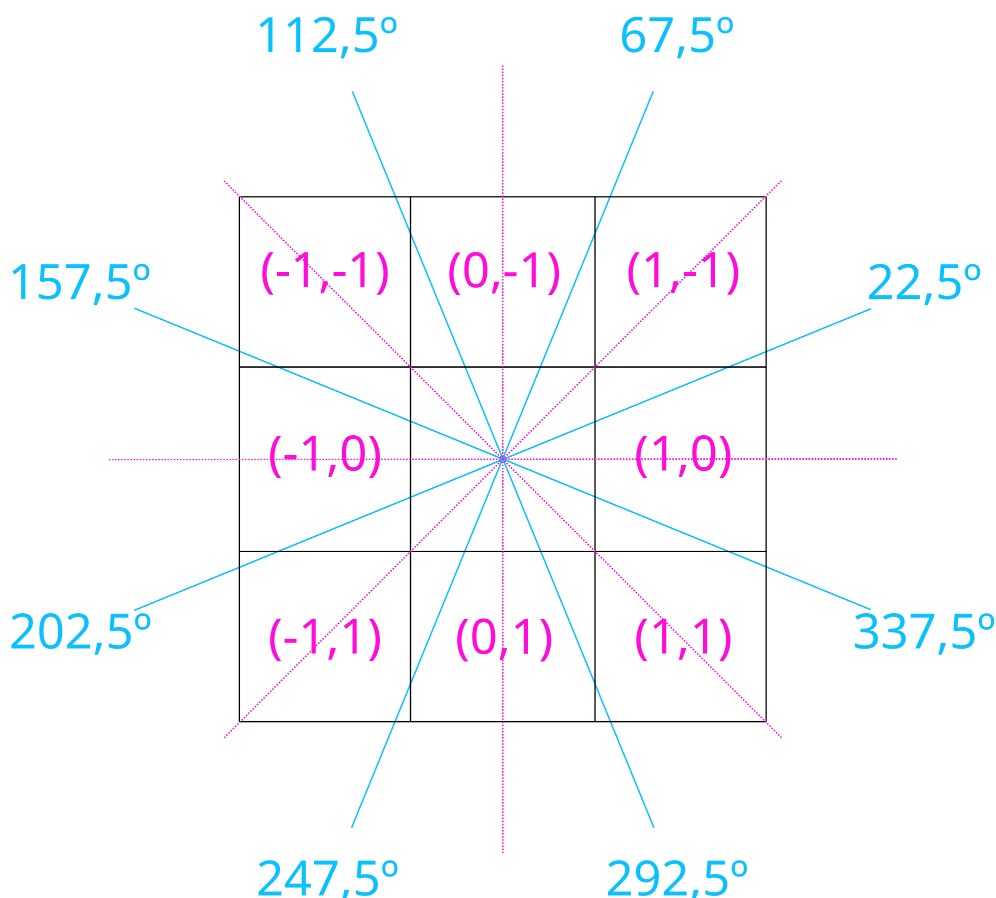


Figura 10 – Ilustração do modo de construção da máscara para o afinamento

Por exemplo, se o ângulo medido for de 22,4 graus (que está entre 22,5 graus e 337,5 graus), as células selecionadas para comparação do máximo com a central serão as de coordenadas (-1,0) e (1,0), conforme descrito na imagem, e se alguma dessas células selecionadas tiverem magnitudes de intensidade maior que a célula central, a célula central não será inclusa na imagem final (caso contrário, ela será).

3.2 IMPLEMENTAÇÃO

Para implementação deste algoritmo foram estudadas maneiras de realizá-lo utilizando apenas código Python, porém sem sucesso. O problema foi o fato de que era preciso,

para a construção dá máscara dinâmica, enviar submatrizes dos dois objetos: ângulos e magnitudes. A solução tomada, então, foi a de implementar uma convolução utilizando código em C++, compilando este código para uma biblioteca compartilhada, e finalmente importando esta biblioteca compartilhada para o projeto em Python.

O código principal desta implementação (código 5) realiza todas as chamadas necessárias antes de começar o processo de afinamento de bordas. Todas as etapas, incluindo os processos que passam pela primeira etapa (filtro Gaussiano) e segunda etapa (filtro Sobel) são realizados para preparar os dados que se esperam ser repassados ao afinamento, ou seja, as matrizes de magnitudes de intensidades dos ângulos direcionais das bordas.

Código 5 – Arquivo principal de início do processamento

```
from scipy.ndimage import gaussian_filter

import image_operations
from sobel.sobel import sobel
import thinner.main


def mount_path(dirname, filename, ext):
    return "{}/{}.{}".format(dirname, filename, ext)


def jpeg_path(image_name):
    return mount_path("./images", "{}_out".format(image_name),
    ↪  "jpeg")


def canny(image_matrix):
    mag, ang = sobel(image_matrix)
    return thinner.main.thinner(mag, ang)


def gaussian(img):
    return gaussian_filter(img, 1.4, mode='constant', cval=0)


def main():
    im = image_operations.imread('images/src.jpeg')
```

```
    image_gray = image_operations.to_gray(im)
    arr = image_operations.to_array(image_gray)
    garr = gaussian(arr)
    mag, ang = canny(garr)
    im2 = image_operations.from_array(mag)
    image_operations.imsave(
        image_operations.to_gray(im2),
        jpeg_path('mag_sobel')
    )


if __name__ == '__main__':
    main()
```

Para o filtro de borramento é usado o filtro Gaussiano. Como falado, este é o tipo de filtro de borramento utilizado nas implementações convencionais do *Canny Edge Detector*. Mais adiante será discorrido sobre ele, e também sobre outros filtros de borramento.

O código em C++ (código 6) é importado para o Python utilizando a biblioteca ctypes (código 7). Assim há uma maior facilidade de escrever o algoritmo usando Python, mas ao mesmo tempo é possível aproveitar o desempenho do C++. A biblioteca ctypes impõe que a importação do recurso escrito em C ou C++ deve ser realizada compilando-o para uma “biblioteca compartilhada”; esta biblioteca precisa ser importada.

Código 6 – Arquivo em C++ que implementa o afinamento de bordas

```
#include <cmath>
#include <iostream>
#include "../cpp_components/matrix.cpp"

using namespace std;

// Number of sections to be divided
// Note: the divisions are relative to the half of the trigonometric
↪  circle, from 0 to PI, or from 0 to negative PI.
const int divisions = 8;

// Angle range in the section
```

```
const float ang_divide = (1.0/divisions) * M_PIf32;

// Mapping arrays that given an angle section. Gives the
↪  correspondent i, j neighbor cell, given the section position.
const int mapIndexJ[] = {1, 1, 1, 0, 0, -1, -1, -1, -1};
const int mapIndexI[] = {0, 1, 1, 1, 1, 1, 1, 0, 0};

/** Compare the neighbors in the angle direction.
 * @param i: the pixel i coord in the image matrix.
 * @param j: the pixel j coord in the image matrix.
 * @param m: the magnitude input matrix
 * @param a: the angle input matrix
 * @return if the analyzed pixel (the center pixel in the mask) has
 ↪  the stronger magnitude over the analysed direction in the mask
 ↪  scope.
 */
bool isGreaterPixel(int i, int j, const Matrix& m, Matrix& a) {
    float ang = a.get(i, j);

    float place = ang/ang_divide;
    int absPlace = abs(place);
    // The sign variable indicates which half of the trigonometric
    ↪  circle we are on
    int signal = place > 0 ? -1 : 1;
    int maskI = mapIndexI[absPlace] * signal;
    int maskJ = mapIndexJ[absPlace];

    return m.get(i + maskI, j + maskJ) < m.get(i, j)
        && m.get(i - maskI, j - maskJ) < m.get(i, j);
}

// Expose the thinner function as shared c object.
extern "C"
/** Execute the thinner process to all pixels in the image.
 *
 * @param mag: the input magnitude matrix as flat array reference
 * @param ang: the input angle matrix as flat array reference
 * @param out_mag: the output magnitude matrix as flat array
 ↪  reference
```

```
 * @param out_ang: the output angle matrix as flat array reference
 * @param l: number of lines of the image matrices
 * @param c: number of columns of the image matrices
 *
 * @note the size of all flat array matrices must be equal to l*c
 */
void thinner(
    float* mag,
    float* ang,
    float* out_mag,
    float* out_ang,
    int l,
    int c
)
{
    Matrix m(mag, l, c);
    Matrix a(ang, l, c);
    Matrix o_mag(out_mag, l, c);
    Matrix o_ang(out_ang, l, c);

    for(int i = 0; i < l; i++) {
        for (int j = 0; j < c; j++) {
            auto isGreater = isGreaterPixel(i, j, m, a);
            o_mag.set(i , j,  isGreater ? m.get(i, j) : 0);
            o_ang.set(i , j,  isGreater ? a.get(i, j) : 0);
        }
    }
}

int main()
{}
```

Código 7 – Arquivo que importa arquivo de afinamento compilado do C++ e faz mapeamento para funções e argumentos desse arquivo

```
import os
from ctypes import CDLL
```

```python
from ctypes import c_size_t, c_int
import numpy as np
import pathlib

# load the library

_NDPOINTER = np.ctypeslib.ndpointer(dtype=np.float32,
                                    ndim=1,
                                    flags="C")
currdir = pathlib.Path(__file__).parent.resolve()
thin_lib = CDLL("{}/thinner.so".format(currdir))
thin_lib.thinner.argtypes = [
    _NDPOINTER, _NDPOINTER, _NDPOINTER, _NDPOINTER,
    c_int, c_int
]
thin_lib.thinner.restype = None


def thinner(arr_mag, arr_ang):
    assert arr_mag.shape == arr_ang.shape
    a = arr_mag.flatten()
    b = arr_ang.flatten()
    c = np.zeros(a.shape, dtype=np.float32).flatten()
    d = np.zeros(b.shape, dtype=np.float32).flatten()
    thin_lib.thinner(a, b, c, d, arr_mag.shape[0], arr_mag.shape[1])
    return c.reshape(arr_mag.shape), d.reshape(arr_ang.shape)
```

Além disso, é implementada a classe C++ “Matrix” (código 8), com o objetivo de facilitar os cálculos com os vetores provenientes do Python, representando-os como um objeto matricial e mapeando as posições corretamente para qualquer cálculo realizado. Este código será reutilizado em experiências posteriores neste trabalho.

Código 8 – Classe Matrix que abstrai as posições das células para uso nos código em C++

```cpp
#include <tuple>
#include <cmath>
```

```
struct Position {
    int x;
    int y;
    int operator[](int coord) {
        return coord ? y : x;
    }
};

typedef Position Size;

class Matrix {
private:
    float * _matrix_arr;
    int l, c;
public:
    Matrix(float* arr, int l, int c): _matrix_arr(arr), l(l), c(c) {}

    int index(int i, int j) const {
        return i * c + j;
    }

    long index(Position position) const {
        return index(position[0], position[1]);
    }

    Position position(int arr_index) {
        return Position{arr_index / c, arr_index % c};
    }

    Size shape(){
        return Size{l, c};
    }

    float get(int i, int j) const {
        long n = index(i, j);
        if (n >= 0 && n < l*c) return _matrix_arr[n];
        return 0;
```

```
    }

    void set(int i, int j, float value) {
        long n = index(i, j);
        if (n >= 0 && n < l*c) {
            _matrix_arr[n] = value;
        }
    }

    float get(Position pos) {
        return get(pos[0], pos[1]);
    }

    void set(Position pos, float value) {
         set(pos[0], pos[1], value);
    }

    ~Matrix(){
        _matrix_arr = nullptr;
    }

    float * get_arr() {
        return _matrix_arr;
    }
};
```

O resultado deste trabalho pode ser traduzido, então, na imagem 11. É facilmente observável, se compararmos com o resultado do capítulo 2, que a imagem aqui apresentada tem bordas mais nítidas e precisas quanto à localização. Entretanto, ainda é possível notar bordas originárias de ruídos da imagem original, que precisam ser removidas para melhor analisar os dados nas etapas futuras.

Figura 11 – Resultado do afinamento na imagem de teste

4 HYSTERESIS THRESHOLDING

No contexto mencionado sobre o afinamento de bordas, em que é interessante conseguir remover os ruídos remanescentes dentro da imagem, é que surge a ideia do *Hysteresis Thresholding*. O objetivo desta técnica, ainda no âmbito do processamento do *Canny Edge Detector*, é o de eliminar registros espúrios que não puderam ser removidos nas etapas anteriores.

Existe uma questão a ser considerada: como limpar a imagem de modo que apenas as bordas mais intensas, ou seja, as bordas de interesse, permaneçam, enquanto todo o resto é removido. A resposta a esse questionamento é o filtro *Hysteresis Thresholding*.

Uma maneira muito simples de implementar um filtro que possa remover dados espúrios, seria a adição de apenas um limiar (threshold). Todos os pixels que tivessem intensidades maiores ou iguais a ele seriam considerados válidos para a formação da nova imagem. Todos os pixels abaixo, seriam descartados.

John Canny, em seu artigo sobre detecção de bordas (CANNY, 1986), definiu preceitos importantes para garantir a performance de um detector desta natureza. Resumidamente, tal artigo define três critérios de performance para um detector de bordas explicados a seguir:

1. Boa detecção. Um detector deverá ter baixa probabilidade de falhar em marcar pontos que realmente pertencem a uma borda, ao mesmo tempo que deve ter também uma baixa probabilidade de marcar incorretamente um ponto que não pertence.

2. Boa localização. Os pontos marcados como pontos da borda deverão estar o mais próximos possível do centro dela.

3. Somente uma resposta para uma única borda. Se houverem duas respostas para a mesma borda, uma delas deverá ser incorreta. Este critério está implícito no primeiro, porém, Canny explica que, matematicamente, o primeiro critério não consegue capturar essa característica quando o operador é executado.

É possível perceber que os critérios definidos por John não estabelecem valores numéricos para a definição do que seria uma boa detecção, e que, portanto, o filtro simples definido acima poderá ser aceito como válido segundo eles. Contudo, tal filtro pode ser melhorado, pois, como Canny também afirma no mesmo artigo, os critérios podem ser avaliados de forma matemática utilizando métricas que possam ser otimizadas. O método estabelecido para remover bordas espúrias, se não for bom o suficiente para a aplicação ao qual está destinado, poderá causar uma baixa performance do detector em relação ao primeiro e também ao terceiro critérios, pois com ele, é possível que pontos que não pertençam a borda sejam aceitos e pontos que deveriam pertencer não sejam.

Um trabalho bastante semelhante ao que está sendo proposto aqui, no que diz respeito à implementação do *Canny Edge Detector*, é o artigo publicado na plataforma Medium por Sofiane Sahir (SAHIR, 2019). Durante todo o seu desenvolvimento, o artigo realiza escolhas bastante parecidas com o que está sendo proposto aqui, tanto com o operador Sobel quanto o operador de afinamento de bordas (*Non-Maximum Suppression*, ou, em português, Supressão Não-Máxima).

A abordagem de Sahir para o caso do threshold da imagem, entretanto, não parece funcionar completamente. Analisando seu trabalho, e entendendo sua definição de pixel forte (pixel que certamente irá ser selecionado para a borda), pixel fraco (pixel que deverá passar por um processo decisório se vai ou não fazer parte da borda) e pixel irrelevante (pixel descartado da borda), é possível observar que apenas os pixels fracos imediatamente próximos de um pixel forte são selecionados para compor a borda, deixando a possibilidade de um pixel fraco mais adiante, que deveria pertencer a borda, não ser selecionado. A idéia deste presente trabalho, diferente da idéia de Sahir, é de selecionar para a composição da borda todos os pixels fracos que são alcançáveis por um forte. Ou seja, um pixel fraco deve pertencer a borda se existe um caminho entre um pixel forte e ele, independente da distância, que não contenha pixels irrelevantes.

4.1 IMPLEMENTAÇÃO DO FILTRO

Observando a idéia descrita aqui, é fácil notar de antemão a abordagem a ser implementada. O método escolhido para alcançar o objetivo de processar todos os pixels fortes e fracos dentro da imagem será a busca em largura.

A busca em largura é bem interessante para buscar todos os pixels alcançáveis a partir de um pixel inicial e um critério, mas neste contexto, poderá haver pixels que deveriam ser selecionados e que não serão por estarem em uma região isolada. Por este motivo é importante considerar na imagem, não o conceito de árvore de busca, mas sim floresta.

Nesta floresta, teremos raízes onde o percurso se iniciará, ramos que serão percorridos pelo código e finalmente folhas, onde será finalizado o caminho. As raízes das árvores serão sempre um pixel forte, já os ramos e as folhas poderão ser pixels fortes ou fracos, e os pixels irrelevantes não farão parte da árvore. Para definir, neste código, o que seria pixel forte, pixel fraco ou pixel irrelevante, será utilizado, um filtro *Hysteresis Thresholding* implementando esta busca. Acima ou no mesmo nível do primeiro threshold (upper_threshold), estarão todos os pixels fortes, abaixo do segundo threshold (bottom_threshold), estarão todos os pixels irrelevantes. Os pixels fracos estarão no mesmo nível ou acima do segundo threshold e abaixo do primeiro.

A imagem 12 ilustra em um gráfico estes thresholds em roxo. Ainda nesta imagem, em verde estão coloridos os pixels fortes, em azul os pixels fracos alcançáveis pelo pixel forte. Estas duas classes serão consideradas para a imagem final. Em cinza, pixels irrelevantes

e, finalmente, em laranja pixels fracos não alcançáveis pelo pixel forte. As duas classes finais (cinza e laranja) não serão consideradas.

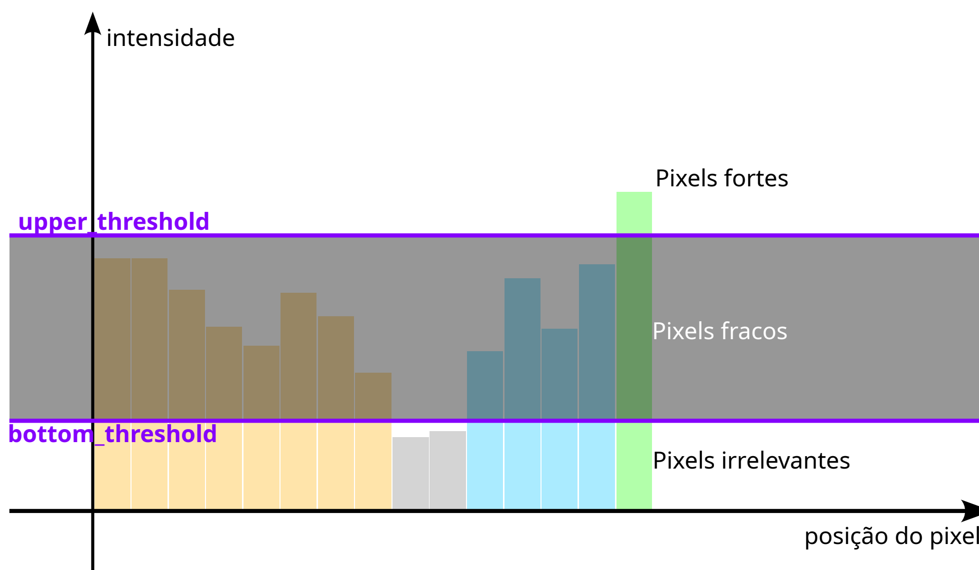


Figura 12 – Gráfico representativo da relação entre os pixels e ambos os thresholds

Tal como o afinamento de bordas, o código aqui mencionado também precisou ser realizado em C++ para otimizar o desempenho. Assim, se faz necessário a construção de um novo integrador para que este código consiga ser chamado pelo código principal em Python.

Código 9 – Arquivo para integração do código em C++ para python - threshold/-main.py.

```
import os
from ctypes import CDLL
from ctypes import c_float, c_int
import numpy as np
import pathlib

# load the library

_NDPOINTER = np.ctypeslib.ndpointer(dtype=np.float32,
                                    ndim=1,
                                    flags="C")
currdir = pathlib.Path(__file__).parent.resolve()
threshold_lib = CDLL("{}/threshold.so".format(currdir))
threshold_lib.threshold.argtypes = [
    _NDPOINTER,
```

```
    _NDPOINTER,
    _NDPOINTER,
    _NDPOINTER,
    c_int,
    c_int,
    c_float,
    c_float
]
threshold_lib.threshold.restype = None


def threshold(
        arr_mag,
        arr_ang,
        upper_threshold,
        bottom_threshold
):
    assert arr_mag.shape == arr_ang.shape
    a = arr_mag.flatten()
    b = arr_ang.flatten()
    c = np.zeros(b.shape, dtype=np.float32).flatten()
    d = np.zeros(b.shape, dtype=np.float32).flatten()
    threshold_lib.threshold(
        a, b, c, d, arr_mag.shape[0],
        arr_mag.shape[1], upper_threshold, bottom_threshold
    )
    return c.reshape(arr_mag.shape), d.reshape(arr_ang.shape)
```

O algoritmo completo do filtro *Hysteresis Thresholding* encontra-se no código 10. Neste código, como mencionado, é executada a busca em largura nos pixels da imagem a fim de remover registros espúrios. Aqui é importante explicar que um pixel pode ser candidato a pertencer a duas ou mais árvores da floresta, entretanto, ao visitá-lo uma vez, ele passará a pertencer à árvore que estiver sendo construída no momento, pois ele será marcado como pixel visitado, e, nesse algoritmo, pixels só podem ser visitados uma única vez.

Código 10 – Arquivo contendo código para realização do threshold duplo - threshold.cpp.

```cpp
#include <iostream>
#include <queue>
#include <unordered_set>
#include "./../cpp_components/matrix.cpp"

using namespace std;

/* Set of all visited pixels.
 * @note Useful to not repeat operations on the same pixel.
 */
typedef  unordered_set<string> VisitedList ;

/** Create a Position object.
 * @param i: the i coord
 * @param j: the j coord
 * @return the created Position Object
 */
Position makePos(int i, int j) {
    return {i, j};
}

/** Test if a given position is already visited, given i, j coords.
 * @param i: the i coord
 * @param j: the j coord
 * @param visited: the reference to the visited list
 * @return the test result (true or false)
 */
bool isVisited(const int i, const int j, VisitedList& visited ) {
    string visitedValue = to_string(i) + "-" + to_string(j);
    return visited.find(visitedValue) != visited.end();
}

/** Test if a given position is already visited, given a Position
↪  object
 * @param position: the Position object
 * @param visited: the reference to the visited list
 * @return the test result (true or false)
 */
```

```
bool isVisited( Position position, VisitedList& visited ) {
    return isVisited(position.x, position.y, visited);
}

/** Mark a position as visited, given i, j coords.
 * @param i: the i coord
 * @param j: the j coord
 * @param visited: the reference to the visited list
 */
void setVisited(int i, int j, VisitedList& visited) {
    string visitedValue = to_string(i) + "-" + to_string(j);
    visited.insert(visitedValue);
}

/** Mark a position as visited, given a Position object.
 * @param position: the Position object
 * @param visited: the reference to the visited list
 */
void setVisited(Position position, VisitedList& visited) {
    setVisited(position.x, position.y, visited);
}

/** Test if a position is invalid.
 * A position is invalid if it does not belong to the image matrix.
 */
bool isInvalidPosition(
    const Position &position, const Size & matrix_size
)
{
    int rows = matrix_size.x;
    int cols = matrix_size.y;
    return  position.x < 0 || position.y < 0 || position.x >= rows ||
     ↪  position.y >= cols;
}

/** Find the next tree root.
 * Aka: the next non-visited strong pixel position.
 * If none was found, return an invalid position.
```

```
 * The upper_threshold is used to do so.
 */
Position findNextRoot(
        Position current, Matrix &mag,
        float upper_threshold, VisitedList& visited
    )
{
    const Size shape = mag.shape();
    const int rows = shape.x;
    const int cols = shape.y;
    const auto posI = current.x;
    const auto posJ = current.y;
    const long initialIndex =
            posI == -1 && posJ == -1 ? -1 : mag.index(current);

    for (long i = initialIndex + 1; i < rows*cols; i++)
    {
        Position candidatePos = mag.position(i);
        auto candidateValue = mag.get(candidatePos);

        if (
            candidateValue >= upper_threshold
            && !isVisited(candidatePos, visited)
        )
        {
            return candidatePos;
        }
    }

    return makePos(-1, -1);
}

/** Add node to the queue and mark it as visited.
 * @param position: the Position object
 * @param queue: the queue of accepted position nodes
 * @param visited: the reference to the visited list
 */
void visitNode(
```

```
    Position & position, deque<Position> & queue,
    VisitedList& visited
) {
    queue.push_back(position);
    setVisited(position, visited);
}

/** Process all adjacent nodes from a given node position.
 * Once the node is processed, it will not be processed again.
 * All the nodes are classified to enter the queue
 * (as children in the tree) or not.
 * The bottom_threshold is used to do so.
 */
void visitNodes(
    Position & currentPosition,
    Matrix &mag,
    float bottom_threshold,
    VisitedList& visited,
    deque<Position> & queue
) {
    int current_i = currentPosition[0];
    int current_j = currentPosition[1];

    for (int i = -1; i <= 1; i++)
    {
        for (int j = -1; j <= 1; j++)
        {
            Position neighborPos = {current_i + i, current_j + j};
            float neighborValue = mag.get(neighborPos);
            if (
                !isInvalidPosition(neighborPos, mag.shape())
                && !isVisited(neighborPos, visited)
            ) {
                if (neighborValue >= bottom_threshold  )
                {
                    visitNode(neighborPos, queue, visited);
                } else {
```

```
                    // Even if the node doesn't belong to the tree,
                    ↪  it will be marked as visited to avoid
                    ↪  unnecessary processing in the future.
                    setVisited(neighborPos, visited);
                }
            }
        }
    }
}


/**
 * Perform the Hysteresis Threshold operation
 *
 * @param mag: the input magnitude matrix, as a reference to a flat
 ↪  array
 * @param ang: the input angle matrix, as a reference to a flat array
 * @param out_mag: the output magnitude matrix, as a reference to a
 ↪  flat array
 * @param out_ang: the output angle matrix, as a reference to a flat
 ↪  array
 * @param l: number of matrices lines.
 * @param c: number of matrices columns.
 * @param upper_threshold: Hysteresis upper threshold
 * @param bottom_threshold: Hysteresis bottom threshold
 *
 * @note: All flat array matrices must have l*c size.
 * @note: The possible range for the thresholds must be the magnitude
 ↪  range values of the input magnitude matrix
 */
extern "C" void threshold(
        float *mag, float *ang, float *out_mag, float *out_ang,
        int l, int c, float upper_threshold, float bottom_threshold
    )
{
    Matrix m(mag, l, c);
    Matrix a(ang, l, c);
    Matrix o_mag(out_mag, l, c);
```

```
    Matrix o_ang(out_ang, l, c);

    unordered_set<string> visited;
    deque<Position> queue;

    Position currentRoot = findNextRoot(
            makePos(-1, -1), m, upper_threshold, visited
    );

    // Perform the forest building
    while (!isInvalidPosition(currentRoot, m.shape())) {

            visitNode(currentRoot, queue, visited);

            // Perform the tree building.
            while (!queue.empty())
            {
                auto currentPosition = queue.front();
                queue.pop_front();
                auto currMag = m.get(currentPosition);
                auto currAng = a.get(currentPosition);

                // Set the current pixel got from the queue to the
                ↪  output magnitude and angle matrices.
                o_mag.set(currentPosition, currMag);
                o_ang.set(currentPosition, currAng);

                visitNodes(
                 currentPosition, m, bottom_threshold, visited, queue
                );
            }
            currentRoot = findNextRoot(
                currentRoot, m, upper_threshold, visited
            );
    }

}
```

```
int main()
{}
```

O objetivo aqui é encontrar uma raiz inicial. Isto é feito percorrendo a matriz linha a linha até encontrar o primeiro pixel forte, e este será então a raiz corrente. Depois de encontrado a raiz corrente, o algoritmo analisa todos os vizinhos imediatos dele. Aqueles vizinhos não visitados ainda e que são pixels fortes ou fracos serão adicionados na fila para serem os próximos analisados. O processo se repete até que não haja nenhum pixel na fila, mas isso não significa que o algoritmo finalizou, mas apenas que a primeira árvore terminou de ser processada.

Em seguida, partindo da posição da raiz escolhida anteriormente, uma nova raiz não visitada é escolhida e o processo todo se repete. Assim, pode-se garantir que todos os pixels alcançáveis por um pixel forte serão visitados e que, portanto, todas as bordas de interesse serão adicionadas.

Os valores dos limiares foram conceitualizados de forma a, se necessário, ser possível que eles assumam qualquer valor no intervalo válido de intensidade de pixels. Desta forma, foi necessário intervir no resultado do filtro Sobel para normalizar as intensidades das imagens em um intervalo predefinido, já que os valores resultantes não normalizados deste filtro podem ser bem altos. Este intervalo foi escolhido com base nos valores máximos de intensidade de um pixel de 8 bits, que é padrão em uma imagem.

A preocupação, neste contexto da busca em largura sendo executada como filtro, seria a da performance de execução não ser satisfatória. A busca em largura (no modo como foi construída aqui) executaria no pior caso uma velocidade igual a da convolução; em média, ela deverá executar ligeiramente mais rápido.

A justificativa para que a busca em largura execute ligeiramente mais rápido que a convolução na média vêm da natureza dos dados analisados. Como a busca só visita, e portanto só analisa, pixels fortes e fracos, ignorando pixels irrelevantes; dessa forma, muitos pixels em uma imagem obtida das etapas anteriores não seriam visitados. Isto, é claro, se os thresholds escolhidos fossem tais que permitissem uma gama maior de pixels classificados como irrelevantes. E esse tipo de otimização não ocorre na convolução, porque esta precisa analisar todos os pixels independentemente de sua natureza.

Para demonstrar com mais clareza os resultados obtidos, foi realizado uma pequena modificação na imagem obtida do passo do afinamento, para que fosse possível visualizar os pixels que, apesar de quase imperceptíveis, tem uma intensidade diferente de nula. Tal modificação consistiu apenas de multiplicar as intensidades dos pixels por um valor alto o suficiente para que sejam vistos. A figura 13 mostra esta imagem após modificação.

A figura 14, por sua vez, mostra a ação do *Hysteresis Thresholding* na imagem. A

Figura 13 – Imagem proveniente da etapa de afinamento, após multiplicação das intensidades por 50, antes do filtro *Hysteresis Thresholding*

mesma multiplicação foi realizada para melhor visualização desta ação. É fácil notar que quase todos os registros espúrios foram removidos, deixando apenas os contornos que realmente pertencem às bordas de interesse.

Figura 14 – Imagem após filtro *Hysteresis Thresholding* e com multiplicação das intensidades por 50 realizada

Por fim, pode-se ver na figura 15 a imagem após a execução do filtro, porém, neste caso, sem a multiplicação para visualização e com a normalização mencionada anterior-

mente, preservando as proporções originais de intensidades da imagem após a execução do afinamento de bordas.

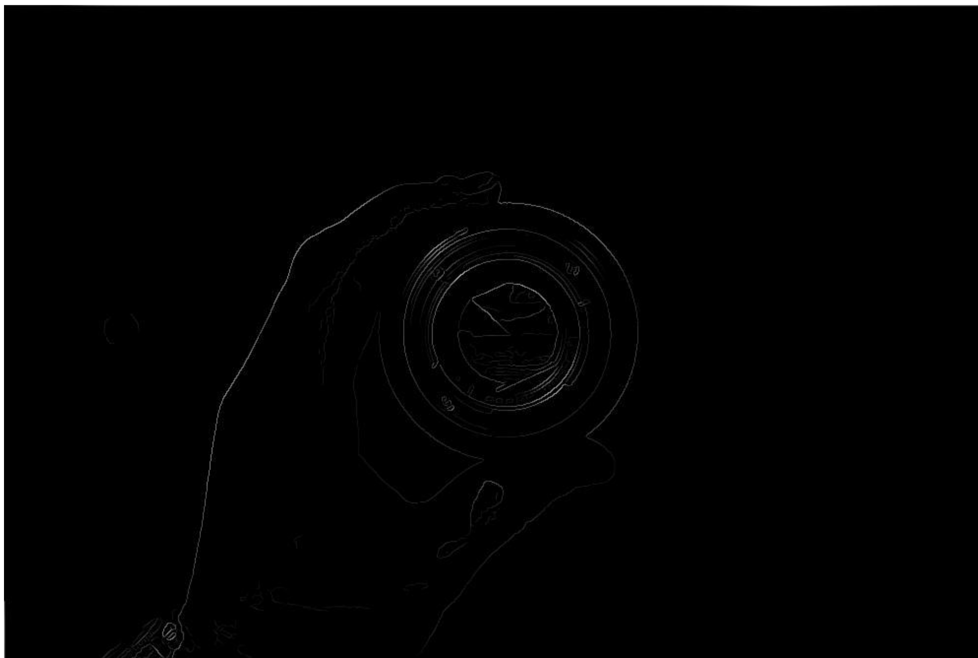


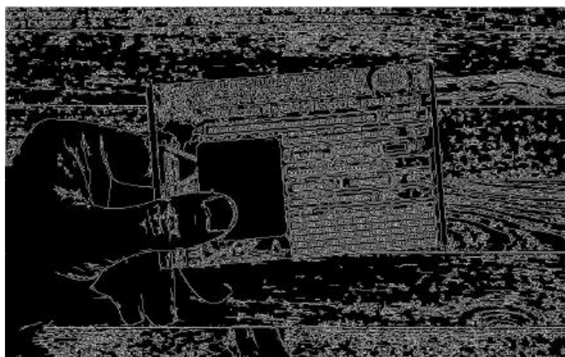
Figura 15 – Imagem após filtro *Hysteresis Thresholding* sem multiplicação de intensidades

5 FILTRO DE BORRAMENTO

Grande parte das experiências anteriores realizadas neste trabalho utilizaram um filtro de borramento. Um filtro que borre a imagem pode parecer contra-intuitivo no contexto em que é necessário obter detalhes presentes nela. Entretanto a vantagem deste tipo de filtro aqui é a de remover detalhes excessivos, os ruídos.

(a) Imagem original

(b) Imagem após Canny Edge Detector (sem borramento)



(c) Imagem após Canny Edge Detector (com borramento)

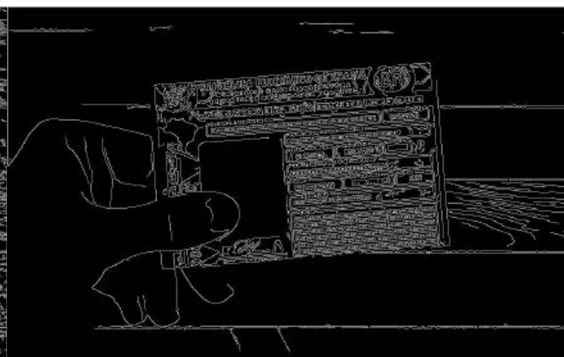


Figura 16 – Comparativo entre imagens. (a) Acima a imagem original. (b) Abaixo, à esquerda, após realizar o Canny, sem borramento. (c) Abaixo, à direita, com o filtro de borramento gaussiano

Uma imagem ruidosa é muito prejudicial à execução do *Canny Edge Detector* apresentado aqui, pois este utiliza o filtro Sobel para obter as bordas. Este filtro é bastante sensível a ruídos, e por este motivo, é importante removê-los para que se tenha uma precisão maior nos resultados. Isso significa que sem o filtro de borramento, a imagem processada final teria muito mais bordas do que as de interesse. A figura 16 mostra

um comparativo demonstrando o quão essencial é o filtro de borramento. Nela é possível observar o que ocorreria com a imagem final após o *Canny Edge Detector* se não aplicássemos um filtro de borramento como pré-processamento.[1]

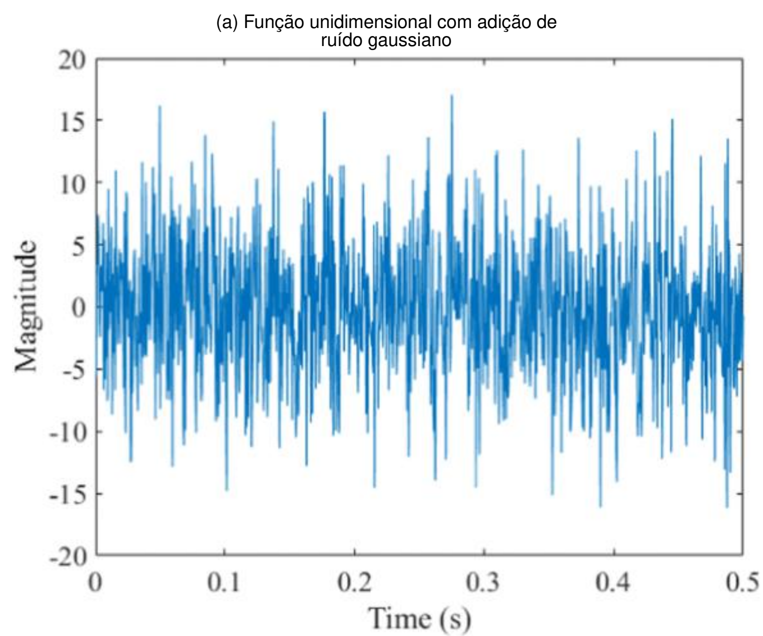


Figura 17 – Funções com adição de ruído gaussiano. (a) Acima, unidimensional. (b) Abaixo, bidimensional

[1] Parâmetros utilizados para execução do filtro de borramento gaussiano: kernelSize=5, standardDeviation=1.4

Um tipo de ruído bem comum é o gaussiano. Este tipo de ruído, aparece em ondas de rádio, distorcendo a forma de onda original. Ele também é comum em imagens de vídeo e fotografias, principalmente aquelas capturadas em ambientes com pouca luz. Um ruído é dito gaussiano se ele, ao gerar perturbações na função de interesse, obedece a uma distribuição de probabilidade gaussiana. A figura 17 mostra o ruído gaussiano ocorrendo em duas funções. Uma função contínua em uma dimensão, que poderia ser, por exemplo, um sinal de áudio sendo transmitido. E uma função discreta em duas dimensões, que no exemplo, é uma imagem.

O artigo original do *Canny Edge Detector*, bem como algumas implementações posteriores, fazem uso do filtro gaussiano para o borramento, justamente por considerar, de um modo geral, que os ruídos presentes nas imagens podem ser mais facilmente modelados através dos gaussianos. Isto garante a remoção de uma boa parte dos ruídos, permitindo uma imagem mais limpa. Além disso, o filtro de borramento gaussiano é uma operação linear, e por isso é possível otimizá-la através de transformações matemáticas, como a transformada de Fourier (BRACEWELL, 2000).

Entretanto, ao aplicar o filtro gaussiano, tem-se uma perda na qualidade da imagem como um todo, com um borramento total dela, inclusive as bordas, diminuindo assim a assertividade de um algoritmo cujo propósito é obtê-las. Desta forma, para o contexto deste trabalho foram realizados testes com outros filtros, que preservem melhor as bordas presentes na imagem.

5.1 FILTRO BILATERAL

Um dos filtros que preserva bordas da imagem, sem borrá-las, o bilateral é um filtro não linear que utiliza o resultado de duas funções gaussianas aplicadas na vizinhança do pixel corrente. Ele é definido na equação 5.1.

$$I_1(x) = \frac{1}{W_p} \sum_{x_i \in \Omega} I_0(x_i) G_r(||I_0(x_i) - I_0(x)||) G_s(||x_i - x||) \tag{5.1}$$

onde, I_0 e I_1 representam, respectivamente, os estados da imagem antes e após a execução do filtro; G_s e G_r são funções gaussianas (ou kernels gaussianos) distintas, que ponderam a importância dos vizinhos considerando, respectivamente, a distância espacial e a distância entre os valores das intensidades em relação ao pixel avaliado (x); Ω é o espaço de vizinhos de x e W_p pode ser escrito como a equação 5.2.

$$W_p = \sum_{x_i \in \Omega} G_r(||I_0(x_i) - I_0(x)||) G_s(||x_i - x||) \tag{5.2}$$

O ponto principal deste filtro é a premissa de que dentro da vizinhança do pixel que está sendo analisado, há uma probabilidade alta de que pixels vizinhos que tenham valores de intensidades (ou cores, se considerar imagens coloridas) parecidas com ele de fato

pertençam ao mesmo "objeto" dentro da imagem e que, portanto, podem ser ponderados como importantes para imagem final.

O que acontece com a imagem submetida a este processo é que em regiões de borda há uma diminuição acentuada dos efeitos do filtro além dos limites da borda, permitindo que esta seja preservada. A figura 18 evidencia bastante este efeito.

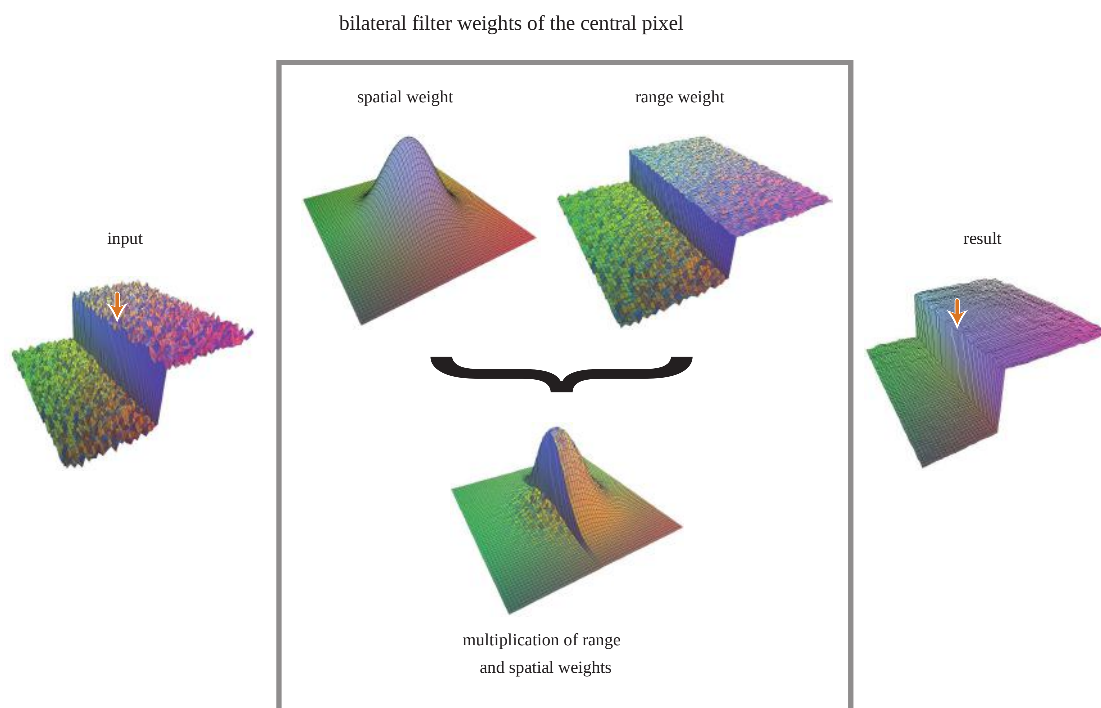


Figura 18 – Visualização do funcionamento do filtro bilateral (PARIS et al., 2007). Pixel central indicado com seta em vermelho

Para realizar o filtro bilateral, foi escolhido a implementação da biblioteca do *OpenCV*, permitindo uma maneira rápida de executar testes de comparação entre os filtros mencionados aqui. O resultado desta introdução do filtro bilateral pode ser visualizado na figura 19. Parâmetros utilizados para a experiência: $d = 9$, $sigmaColor = 80$, $sigmaSpace = 100$ (TEAM, 2023a).

5.2 FILTRO DA MEDIANA

Outro filtro experimentado neste trabalho para remoção de ruídos foi o filtro da Mediana. Este filtro consiste em obter a mediana entre as intensidades de pixels da imagem, e tem um resultado parecido em certo nível com o filtro bilateral.

Diferente da média, a mediana não obtém um novo valor com base nos valores existentes. Ela de fato utiliza um dos valores existentes na lista de valores analisados. No

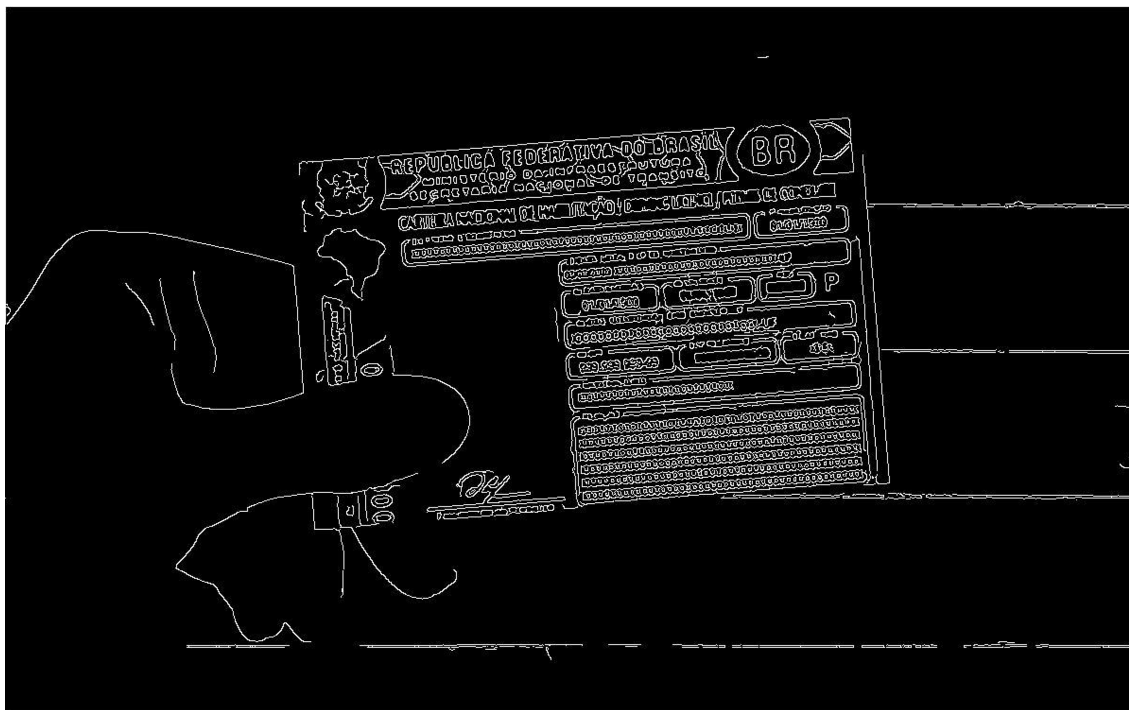


Figura 19 – Resultado da aplicação do *Canny Edge Detector* utilizando filtro bilateral

caso do filtro da mediana, uma espécie de "convolução" é aplicada na imagem, porém de forma não linear, tal como no filtro bilateral.

Aqui o objetivo é separar uma janela de vizinhos do pixel atual, ordenar estes pixels por valor de intensidade em uma lista e finalmente escolher aquele pixel que se encontrar no meio da lista. Como esta operação envolve apenas ordenar e escolher aquele (ou aqueles) que se encontrar no meio da lista, o único parâmetro necessário é a definição do tamanho da lista, ou seja, do número de vizinhos que farão parte do cálculo. Esta lista é construída obtendo os vizinhos presentes dentro da janela analisada e, portanto, o único parâmetro necessário para realizar este filtro, além da imagem, é o tamanho desta janela de vizinhos.

Da mesma forma que na operação comum da mediana, há o caso em que ela contiver um número par de elementos. Quando isto ocorrer, é necessário obter a média dos dois elementos centrais da lista.

Para esta experimentação, novamente, foi utilizada a implementação do filtro fornecida pela biblioteca do *OpenCV* (TEAM, 2023b). É possível observar a ação deste filtro na imagem 20 e o resultado final do *Canny Edge Detector* ao aplicá-lo no processo, na figura 21.[2]

Aqui neste experimento é possível observar o efeito relativamente parecido com o filtro bilateral, entretanto, ao atentar-se para as bordas da imagem, é possível visualizar que de certa forma nem todas elas são completamente preservadas.

[2] Parâmetro utilizado na experiência: $ksize = 5$

Figura 20 – Aplicação do filtro da mediana diretamente sobre a imagem em escala de cinza

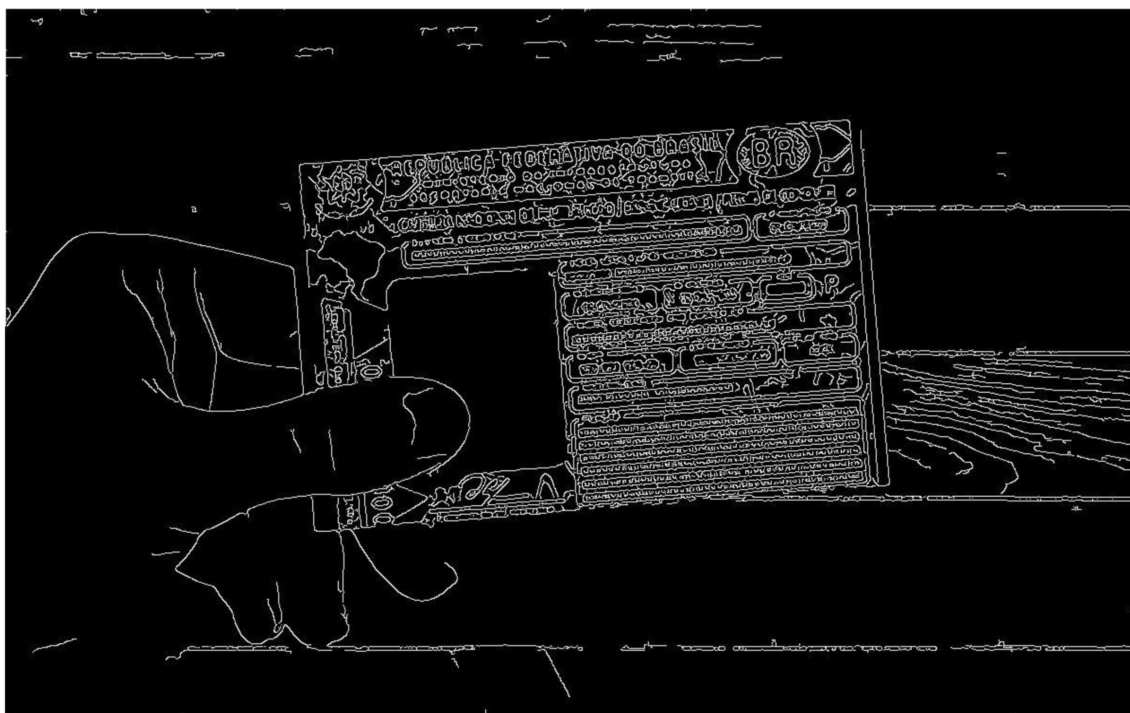


Figura 21 – Resultado da aplicação do *Canny Edge Detector* utilizando filtro da mediana

5.3 AVALIAÇÃO DOS FILTROS

Dentre os filtros implementados, não obtivemos resultados significativamente piores do que o que estava sendo feito no início deste projeto.

Foi identificado, contudo, que um resultado satisfatório na implementação de ambos

os filtros, bilateral e da mediana, necessitaria de um custo computacional muito mais elevado, pois tais filtros não se comportam de maneira linear.

Um teste de performance também foi executado entre os filtros gaussiano, da mediana e bilateral, utilizando os mesmos parâmetros utilizados para as execução do Canny Edge Detector mencionadas anteriormente. Para este teste foram executados tais filtros 100 vezes. O tempo de execução do filtro bilateral foi de cerca de $191ms$. O tempo de execução do filtro da mediana foi de cerca de $172ms$. Enquanto o do gaussiano, foi em torno de $7ms$.

Além disso, o ganho de qualidade de imagem existe, mas é muito pequeno quando comparado com o custo computacional mencionado para alcançá-lo, levando em consideração os resultados visuais das imagens aqui testadas.

Além dos filtros mencionados, foi cogitado também como alternativa inicial o filtro LS-Images (WANG et al., 2015) que utiliza uma outra forma de suavizar imagens preservando bordas, mas que não foi utilizado porque foi identificado que seria proibitivo para o projeto, uma vez que uma parte importante desta implementação é a realização de inversas de matrizes, para cada pixel e vizinhança analisados, tornando seu custo computacional muito alto.

O filtro de borramento escolhido para a tarefa de suavização da imagem e remoção de ruídos, dados os testes realizados, foi o filtro de borramento gaussiano. Além das vantagens de performance, por ele ser um filtro linear, manter o filtro gaussiano permite a realização de um efeito de borramento bem alto na imagem, mesmo com uma matriz de vizinhos bem menor (3 por 3 pixels). Esta decisão tem a desvantagem de causar borramento nas bordas de interesse, como dito, mas com os testes, percebeu-se que o impacto não difere muito das outras alternativas.

6 ANÁLISE COMPARATIVA DOS RESULTADOS

Até agora, foi possível comparar os resultados obtidos aqui com os resultados do filtro Sobel. Foi possível verificar que o *Canny Edge Detector* melhora muito o resultado do Sobel removendo bordas grossas e removendo ruídos desnecessários. Porém faz-se necessário uma verificação com outras implementações do *Canny Edge Detector*.

Uma comparação de resultados do projeto com uma implementação de mercado do *Canny Edge Detector* é importante para ter uma referência de qualidade e performance para qualquer trabalho científico. Para realizar esta avaliação aqui, a ferramenta de mercado escolhida foi o *OpenCV*. Sua escolha foi motivada pela ampla utilização em diversas ferramentas, o que reflete em um bom parâmetro comparativo para o presente projeto.

É verdade que os resultados de ambos os algoritmos são diferentes. Enquanto o *OpenCV* fornece uma imagem com apenas a localização das bordas, o presente projeto fornece as bordas acompanhadas dos seus respectivos valores de intensidade e direções de gradientes (a figura 22 nos mostra o resultado das direções de gradientes deste projeto). Entretanto, é possível fazer esta comparação, se desconsiderarmos os valores de intensidade e direção de gradientes e considerarmos apenas as localidades das bordas.

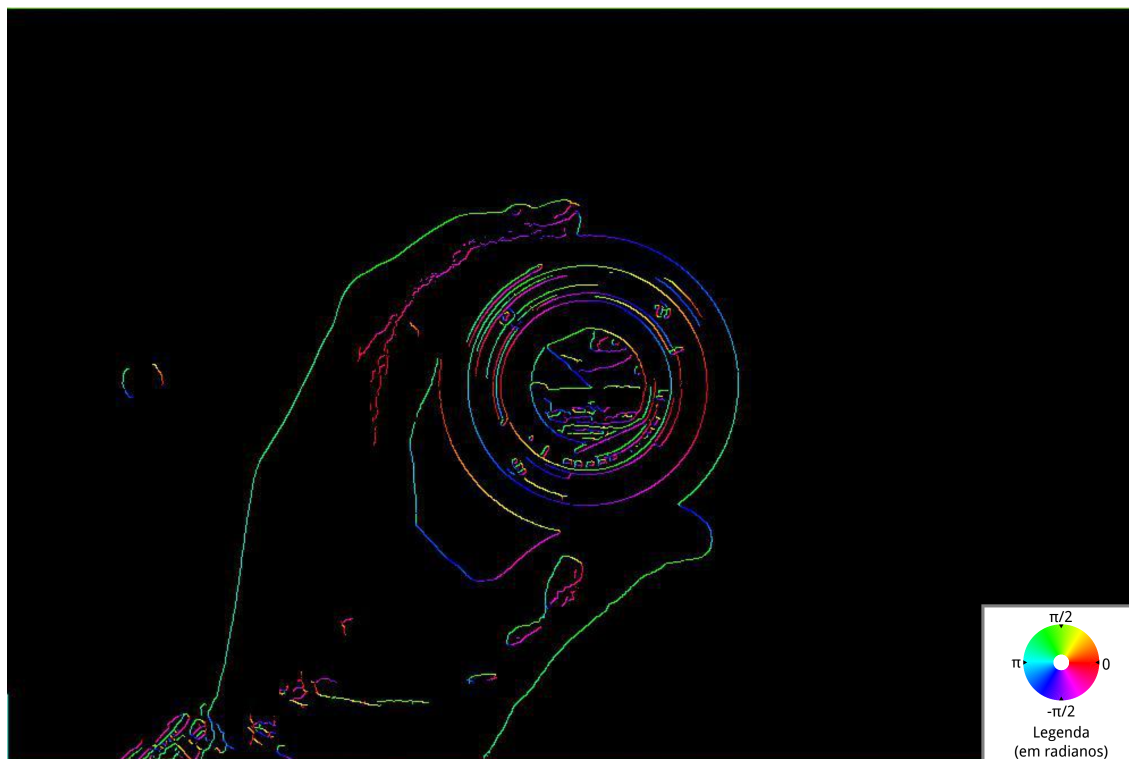


Figura 22 – Imagem resultante da matriz de ângulos produzida pelo algoritmo deste projeto. Vetor gradiente aponta da região mais clara para mais escura.

A metodologia principal dos testes realizados aqui foi a de coletar uma certa quan-

tidade de imagens a serem submetidas à ambos os algoritmos, o do *OpenCV* e também o do projeto aqui desenvolvido e avaliar visualmente a semelhança dos resultados. Além disso, uma comparação de tempo de processamento entre eles foi realizada para mensurar a aplicabilidade do presente algoritmo e possíveis maneiras de melhorá-lo. Para fins de reprodutibilidade é importante salientar também que os testes aqui descritos foram executados em uma máquina com as seguintes configurações:

- Processador: AMD Ryzen 7 7700X (ADVANCED MICRO DEVICES, INC, 2022).
- Placa de Vídeo: MSI GeForce RTX™ 3070 GAMING X TRIO (MICRO STAR INTERNATIONAL, 2023).
- Sistema Operacional: Linux Manjaro 22.1.0 (MüLLER et al., 2023).
- Ambiente Gráfico: Gnome 43 (FOUNDATION, 2023)

6.1 TEMPO DE PROCESSAMENTO

A análise de tempo foi concebida realizando chamadas a ambos os algoritmos várias vezes com um determinado número de imagens. Antes de realizar estas chamadas, foi coletado o horário em *Unix Timestamp*[1] com segundos fracionários. Logo após, este dado foi coletado novamente e subtraído do primeiro, fornecendo assim os segundos decorridos. Este procedimento foi feito em cada um dos algoritmos em testes (*OpenCV* e o projeto aqui apresentado). O código em Python para esta execução pode ser visualizado no código 11.

Código 11 – Código para execução de ambos os algoritmos, *OpenCV* e do presente projeto

```
from math import floor

from canny.main import post_process_image, prepare_image,
↪  process_image
import cv2 as cv
from os import walk
import time


def getimage_test_files_path(dir_path):
    return next(walk(dir_path), (None, None, []))[2]
```

[1] Número que corresponde ao tempo decorrido, em segundos, desde a meia noite do dia primeiro de janeiro de 1970 (UTC)

```
def compare():
    test_image_paths = getimage_test_files_path('./test_image_srcs')
    output1 = []
    output2 = []
    input_arr = []
    for img in test_image_paths:
        img_arr, size =
        ↪ prepare_image('./test_image_srcs/{}'.format(img))
        input_arr.append((img_arr, img, size))


# ------------------------------------
# Project Canny Edge Detector
# ------------------------------------
    start_time = time.time()
    for (img_arr, img, size) in input_arr:
        mag, ang = process_image(img_arr, 20, 30)
        output1.append((mag * 100, img))

    print("--- %s seconds project ---" % (time.time() - start_time))

    for (mag, img) in output1:
        post_process_image(mag,
        ↪ './test_image_dsts_canny_project/{}'.format(img))


# ------------------------------------
# OpenCv Canny Edge Detector
# ------------------------------------
    start_time = time.time()
    for (img_arr, img, _) in input_arr:
        opencv_mag = cv.Canny(img_arr, 100, 200)
        output2.append((opencv_mag, img))

    print("--- %s seconds opencv ---" % (time.time() - start_time))
```

```
    for (mag, img) in output2:
        post_process_image(mag,
        ↪  './test_image_dsts_opencv_canny/{}'.format(img))


if __name__ == '__main__':
    compare()
```

Ao todo, 100 imagens foram utilizadas para o teste de tempo de execução. Os tamanhos das imagens variam entre 576 e 2560 pixels de largura, e entre 384 e 1920 pixels de altura. O tempo total de execução do algoritmo da biblioteca do *OpenCV*, nestas condições, foi de aproximadamente $0,04$ segundos, enquanto o tempo de processamento da execução do projeto, nas mesmas condições, foi de aproximadamente $5,66$ segundos.

Existe, como demonstrado, uma discrepância bastante significativa entre as duas implementações. Contudo, é possível realizar otimizações no código do projeto para que ele se aproxime mais da performance do *OpenCV*. A biblioteca referência, para alcançar tal velocidade, aproveita múltiplas tecnologias. Uma destas tecnologias é o *OpenCl*, o que significa que a execução do *OpenCV* é realizada primariamente na GPU do dispositivo. Para otimizar o projeto em teste, é possível realizar uma ou mais das seguintes otimizações:

- Escrever todo o código em C++ e não apenas parte dele, evitando assim idas e voltas da linguagem interpretada para o programa compilado.
- Implementar múltiplas threads, aproveitando todo o potencial do processador.
- Utilizar tecnologias que melhorem performance, como *OpenCl* ou SIMD, tal como são utilizadas no *OpenCV*.

6.2 QUALIDADE VISUAL DOS RESULTADOS

No que se refere qualidade de resultados, o teste realizado consistiu em obter imagens de ambos os algoritmos e compará-las visualmente. O código utilizado para isso também foi o código 11. Tal código permite salvar os resultados de processamento de ambos os algoritmos para posterior análise. É importante notar também que o resultado da implementação deste projeto foi multiplicado pixel a pixel por 100, para que se tenha um resultado visual mais parecido com o do *OpenCV*, com valores em preto e branco apenas, e não em escala de cinza. Entretanto, para o propósito de adaptar o *Canny Edge Detector* ao *Whiteboard It!* é importante relembrar que os valores de intensidade (escala de cinza) bem como os valores angulares são importantes.

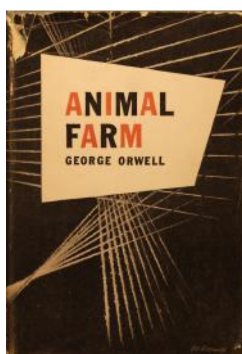
As imagens provenientes do *OpenCV* tem um propósito mais generalizado, ou seja, como esta biblioteca é utilizada para os mais diversos fins, o *Canny Edge Detector* implementado nela também deve seguir este princípio. Isto implica que ela não pode ter detalhes demais e nem detalhes de menos nos resultados dos processamentos, bem como não pode ser lenta ao executá-los. Ela deve estar pronta para os mais diferentes cenários de aplicação.

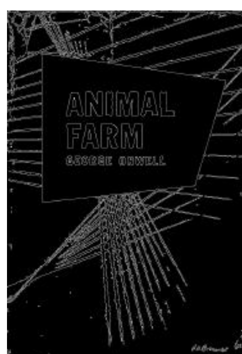
Para esta análise, foi feito uma seleção das imagens processadas anteriormente para efetivamente fazer a análise visual. Das 100 imagens processadas, 6 foram selecionadas. As figuras 23 e 24 contém estas imagens, separando-as em três colunas: "Original", para imagens antes do processamento, "OpenCV" para os resultados do *Canny Edge Detector* do *OpenCV* e "Projeto" para os resultados desta experiência.

Observando as três colunas temos a visualização de que a quantidade de ruídos produzidos pela implementação deste projeto se mostra menor que a do *OpenCV*. Uma sugestão que poderia vir a mente do leitor seria de apenas ajustar os parâmetros do *OpenCV* para que os resultados dele se equiparem à qualidade visual da terceira coluna, entretanto na última linha da figura 23 e também na primeira linha da figura 24 é possível perceber que o *OpenCV* não obtém todas as bordas de interesse com os parâmetros já impostos, o que significa que diminuir a sensibilidade dele a fim de equiparar os ruídos obtidos poderá ter um revés na qualidade de detecção de tais imagens.

Esta perda de qualidade do *OpenCV* versus a busca de mais detalhes pode ser visualizada no exemplo da figura 25. A imagem utilizada para esta experiência foi a mesma da figura 24, a bola de golfe. É possível perceber que apenas na primeira das imagens resultantes obtidas através do processamento do *OpenCV*, que foram apresentadas, a circunferência da bola é obtida completamente, e mesmo assim, muitos ruídos são exibidos. Importante notar que os parâmetros em que houve variação nos resultados de processamento do *Canny Edge Detector* do *OpenCV* foram os limiares superior (denominado na figura como "higher") e inferior (denominado na figura como "lower") do *Hysteresis Thresholding* implementado nesta aplicação. Há outros parâmetros opcionais do *Canny Edge Detector* do *OpenCV*: o tamanho da abertura (máscara) do filtro Sobel e o tipo de cálculo utilizado para obter as magnitudes dos gradientes do mesmo. Estes parâmetros opcionais também foram testados para esta experiência, mas sem alteração significativa nos resultados.

Original | OpenCv | Projeto





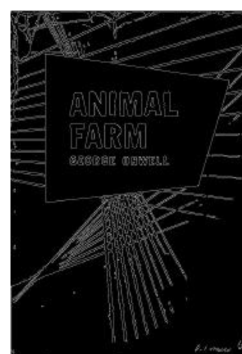


Figura 23 – Imagens comparativas entre os resultados de localidade dos pixels do *OpenCV* e do presente projeto (Primeira parte).

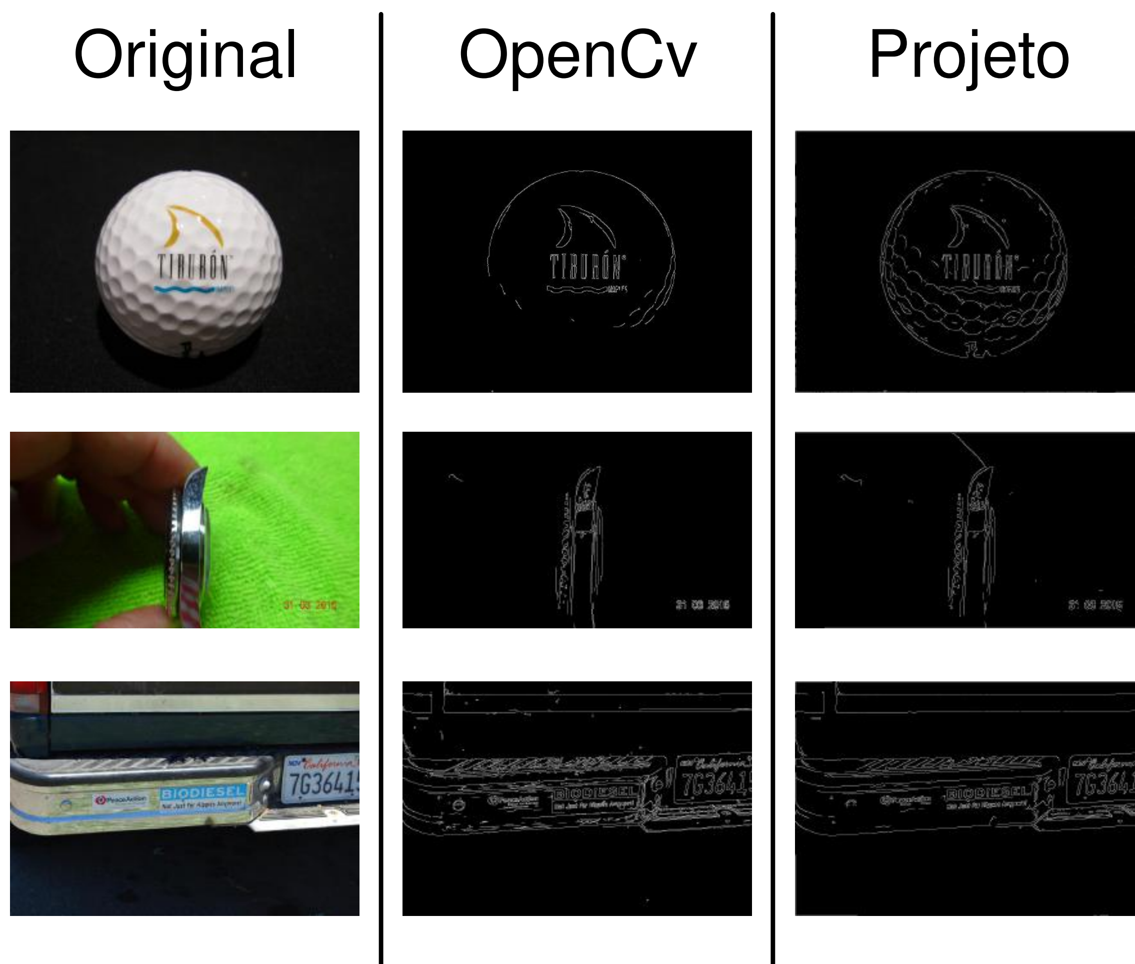
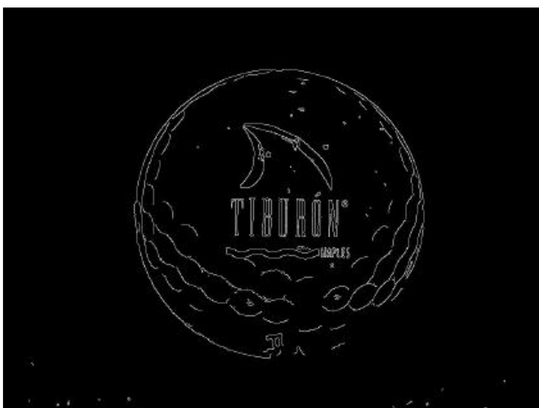


Figura 24 – Imagens comparativas entre os resultados de localidade dos pixels do *OpenCV* e do presente projeto (Segunda parte).

a - OpenCV: higher 50, lower 10

b - OpenCV: higher 100, lower 10



c - OpenCV: higher 100, lower 50

d - OpenCV: higher 200, lower 50

e - OpenCV: higher 200, lower 100

f - OpenCV: higher 255, lower 50

Figura 25 – Imagens de resultados do *OpenCV*, variando os limiares do *Histeresis Thresholding*

7 CONCLUSÃO E CONSIDERAÇÕES FINAIS

A proposta inicial deste projeto foi a de construir uma versão do *Canny Edge Detector* que pudesse ser utilizada como entrada para o algoritmo do projeto *Whiteboard It!*, mas além disso, que pudesse ser tecnicamente utilizada, servindo como alternativa, em substituição ao filtro Sobel em qualquer aplicação em que ele estivesse presente.

Entretanto, em termos de qualidade de resultados, ele não foi projetado para uma aplicação de uso geral. A ideia foi otimizá-lo para diminuir ruídos prejudiciais ao tentar perfilar possíveis linhas em etapas seguintes do *Whiteboard It!*.

Neste projeto, foi explicado em detalhes o que é necessário para construir um *Canny Edge Detector* e foi discutido opções alternativas de implementação ao que foi de fato realizado. Também foi discutido sobre a implementação de filtros de imagem e operações de convolução, e de que maneiras elas são importantes, principalmente para o próprio projeto.

Apesar do projeto aqui desenvolvido ser reutilizável para outros fins, seu propósito não é generalizado, muito pelo contrário, ele tem um objetivo muito bem definido: conseguir a menor quantidade possível de bordas de modo que ainda seja possível para as etapas seguintes do *Whiteboard It!* detectar um objeto quadrangular, o documento. Pensando neste viés, analisando as imagens resultantes podemos ver que são poucas as vezes que o *Canny Edge Detector* implementado aqui apresenta um ruído significativamente ruim para as próximas etapas da detecção.

A grande diferença entre as implementações, como já mencionado, é a etapa de *Hysteresis Thresholding*, e este também é o motivo pelo qual os resultados do OpenCV tem mais ruídos que este projeto. O *OpenCV* executa um algoritmo muito parecido com o implementado por Sofiane Sahir, avaliando pixels dentro de uma região limitada pela máscara de vizinhos. Essa abordagem deixa o algoritmo “cego” quanto ao contexto do que está em volta do pixel analisado. O projeto aqui implementado utiliza busca em largura para realizar a mesma análise, o que pode deixá-lo mais difícil para otimizar, mas ao mesmo tempo deixa-o a par do contexto inteiro da imagem.

Desta forma, é possível dizer que para a finalidade deste trabalho, em termos de qualidade dos resultados, o presente projeto se mostra, em média, mais promissor que o *Canny Edge Detector* do *OpenCV*.

Levando em consideração as etapas do processo e em como foram implementadas, sugere-se que, para trabalhos futuros, haja a reescrita do código utilizando C++ como linguagem principal, bem como o uso de tecnologias de otimização gráficas, e uso de concorrência para aproveitar todo o desempenho do processamento, como mencionadas na seção 6.1. Além disso, sugere-se também a implementação das etapas restantes do *Whiteboard It!*.

Através dos testes aqui realizados foi possível demonstrar a performance do que foi implementado, tanto em relação a tempo de execução quanto em relação a qualidade de imagem, e pode-se dizer que um resultado bastante promissor foi obtido, principalmente quanto à qualidade das imagens resultantes do processo, se comparado com a aplicação de mercado deste algoritmo, o *OpenCV*.

Por tudo isso, é possível afirmar que o projeto cumpriu com os objetivos propostos, atendendo às necessidades do *Whiteboard It!* e apresentando uma alternativa viável e eficaz ao filtro Sobel. É gratificante contribuir para o avanço da tecnologia e para a solução de problemas concretos em uma área tão importante e em constante evolução.

REFERÊNCIAS

ADVANCED MICRO DEVICES, INC. **AMD Ryzen™ 7 7700X**. [S.l.], 2022. Disponível em: https://www.amd.com/en/product/12161. Acesso em: 5 abr.2023.

BRACEWELL, R. N. **The Fourier Transform and Its Applications**. 3. ed. [S.l.]: McGraw-Hill Science/Engineering/Math, 2000.

BURGER, W.; BURGE, M. J. **Digital Image Processing**: An algorithmic introduction using java. 1. ed. [S.l.]: Springer, 2009.

CANNY, J. A computational approach to edge detection. **IEEE Transactions on Pattern Analysis and Machine Intelligence**, PAMI-8, n. 6, p. 679–698, 1986. Disponível em: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4767851. Acesso em: 13 ago.2022.

COMPUTERPHILE. **Canny Edge Detector**. 2015. Disponível em: https://www.youtube.com/watch?v=sRFM5IEqR2w. Acesso em: 30 jul.2022.

COMPUTERPHILE. **Finding the Edges (Sobel Operator)**. 2015. Disponível em: https://www.youtube.com/watch?v=uihBwtPIBxM. Acesso em: 13 ago.2022.

FEBRABAN, I. **Transações bancárias por canais digitais crescem 23% em 2021 e já são 7 em cada 10 operações no país**. 2022. Disponível em: https://noomis.febraban.org.br/temas/inovacao/transacoes-bancarias-por-canais-digitais-crescem-23-em-2021-e-ja-sao-7-em-cada-10-operacoes-no-pais. Acesso em: 17 set.2022.

FOUNDATION, G. **Gnome**. [S.l.], 2023. Disponível em: https://www.gnome.org/. Acesso em: 5 abr.2023.

MICRO STAR INTERNATIONAL. **GeForce RTX™ 3070 GAMING X TRIO**. [S.l.], 2023. Generated 2023-03-29. Disponível em: https://storage-asset.msi.com/datasheet/vga/global/GeForce-RTX-3070-GAMING-X-TRIO.pdf. Acesso em: 5 abr.2023.

MüLLER, P. et al. **Manjaro Linux**. [S.l.], 2023. Disponível em: https://manjaro.org/. Acesso em: 5 abr.2023.

PADRãO, M.; YUGE, C. **Brasileiros das classes C e D usam mais o celular para trabalho**. 2022. Disponível em: https://canaltech.com.br/smartphone/brasileiros-das-classes-c-e-d-usam-mais-o-celular-para-trabalho-217346/. Acesso em: 5 abr.2023.

PARIS, S. et al. A gentle introduction to bilateral filtering and its applications. In: **ACM SIGGRAPH 2007 Courses**. New York, NY, USA: Association for Computing Machinery, 2007. (SIGGRAPH '07), p. 1–es. ISBN 9781450318235. Disponível em: https://doi.org/10.1145/1281500.1281602.

SAHIR, S. **Canny Edge Detection Step by Step in Python — Computer Vision**. 2019. Disponível em: https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123. Acesso em: 7 jan.2023.

SONKA M.; HLAVAC, V.; BOYLE, R. **Image Processing, Analysis, and Machine Vision**. 4. ed. [S.l.]: Cengage Learning, 2014.

TEAM, O. **Canny Edge Detection**. 2022. Disponível em: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html. Acesso em: 5 nov.2022.

TEAM, O. **Image Filtering - Bilateral Filter**. 2023. Disponível em: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga9d7064d478c95d60003cf839430737ed. Acesso em: 22 mar. 2023.

TEAM, O. **Image Filtering - MedianBlur**. 2023. Disponível em: https://docs.opencv.org/4.x/d4/d86/group__imgproc__filter.html#ga564869aa33e58769b4469101aac458f9. Acesso em: 23 mar. 2023.

TEMER, M. et al. **Lei Geral de Proteção de Dados Pessoais (LGPD)**. 2019. Disponível em: https://www.planalto.gov.br/ccivil_03/_ato2015-2018/2018/lei/l13709.htm. Acesso em: 20 jan.2023.

WANG, H. et al. Least-squares images for edge-preserving smoothing. **Computational Visual Media**, v. 1, n. 1, p. 27–35, Mar 2015. ISSN 2096-0662. Disponível em: https://doi.org/10.1007/s41095-015-0004-6.

ZHANG, Z.; HE, L. Whiteboard it! In: . [s.n.], 2002. Disponível em: https://www.microsoft.com/en-us/research/publication/whiteboard-it/. Acesso em: 13 ago.2022.