



MODELAGEM COMPUTACIONAL PARA ESTIMAÇÃO DE CARACTERÍSTICAS DE ONDAS MARÍTIMAS EM BEIRA DE PRAIA

David Estevam de Britto Junior

Projeto de Graduação apresentado ao Curso de Engenharia Eletrônica e de Computação da Escola Politécnica, Universidade Federal do Rio de Janeiro, como parte dos requisitos necessários à obtenção do título de Engenheiro.

Orientador: Flávio Luis de Mello

Rio de Janeiro
Julho de 2017

MODELAGEM COMPUTACIONAL PARA ESTIMAÇÃO DE
CARACTERÍSTICAS DE ONDAS MARÍTIMAS EM BEIRA DE
PRAIA

David Estevam de Britto Junior

PROJETO DE GRADUAÇÃO SUBMETIDO AO CORPO DOCENTE DO CURSO
DE ENGENHARIA ELETRÔNICA E DE COMPUTAÇÃO DA ESCOLA PO-
LITÉCNICA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO
PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU
DE ENGENHEIRO ELETRÔNICO E DE COMPUTAÇÃO

Autor:

David Estevam de Britto Junior

Orientador:

Prof. Flávio Luis de Mello, Ph. D.

Examinador:

Prof Frances Elizabeth Allen, D. Sc.

Examinador:

Prof. Alan Jay Perlis, D. E.

Rio de Janeiro
Outubro de 2008

Declaração de Autoria e de Direitos

Eu, *David Estevam de Britto Junior* CPF 137.221.577-81, autor da monografia *Modelagem Computacional para Estimação de Características de Ondas Marítimas em Beira de Praia*, subscrevo para os devidos fins, as seguintes informações:

1. O autor declara que o trabalho apresentado na disciplina de Projeto de Graduação da Escola Politécnica da UFRJ é de sua autoria, sendo original em forma e conteúdo.
2. Excetuam-se do item 1. eventuais transcrições de texto, figuras, tabelas, conceitos e idéias, que identifiquem claramente a fonte original, explicitando as autorizações obtidas dos respectivos proprietários, quando necessárias.
3. O autor permite que a UFRJ, por um prazo indeterminado, efetue em qualquer mídia de divulgação, a publicação do trabalho acadêmico em sua totalidade, ou em parte. Essa autorização não envolve ônus de qualquer natureza à UFRJ, ou aos seus representantes.
4. O autor pode, excepcionalmente, encaminhar à Comissão de Projeto de Graduação, a não divulgação do material, por um prazo máximo de 01 (um) ano, improrrogável, a contar da data de defesa, desde que o pedido seja justificado, e solicitado antecipadamente, por escrito, à Congregação da Escola Politécnica.
5. O autor declara, ainda, ter a capacidade jurídica para a prática do presente ato, assim como ter conhecimento do teor da presente Declaração, estando ciente das sanções e punições legais, no que tange a cópia parcial, ou total, de obra intelectual, o que se configura como violação do direito autoral previsto no Código Penal Brasileiro no art.184 e art.299, bem como na Lei 9.610.
6. O autor é o único responsável pelo conteúdo apresentado nos trabalhos acadêmicos publicados, não cabendo à UFRJ, aos seus representantes, ou ao(s) orientador(es), qualquer responsabilização/ indenização nesse sentido.
7. Por ser verdade, firmo a presente declaração.

David Estevam de Britto Junior

UNIVERSIDADE FEDERAL DO RIO DE JANEIRO
Escola Politécnica - Departamento de Eletrônica e de Computação
Centro de Tecnologia, bloco H, sala H-217, Cidade Universitária
Rio de Janeiro - RJ CEP 21949-900

Este exemplar é de propriedade da Universidade Federal do Rio de Janeiro, que poderá incluí-lo em base de dados, armazenar em computador, microfiltrar ou adotar qualquer forma de arquivamento.

É permitida a menção, reprodução parcial ou integral e a transmissão entre bibliotecas deste trabalho, sem modificação de seu texto, em qualquer meio que esteja ou venha a ser fixado, para pesquisa acadêmica, comentários e citações, desde que sem finalidade comercial e que seja feita a referência bibliográfica completa.

Os conceitos expressos neste trabalho são de responsabilidade do(s) autor(es).

DEDICATÓRIA

Opcional.

AGRADECIMENTO

Sempre haverá. Se não estiver inspirado, aqui está uma sugestão: dedico este trabalho ao povo brasileiro que contribuiu de forma significativa à minha formação e estuda nesta Universidade. Este projeto é uma pequena forma de retribuir o investimento e confiança em mim depositados.

RESUMO

Inserir o resumo do seu trabalho aqui. O objetivo é apresentar ao pretendido leitor do seu Projeto Final uma descrição genérica do seu trabalho. Você também deve tentar despertar no leitor o interesse pelo conteúdo deste documento.

Palavras-Chave: trabalho, resumo, interesse, projeto final.

ABSTRACT

Insert your abstract here. Insert your abstract here. Insert your abstract here.
Insert your abstract here. Insert your abstract here.

Key-words: word, word, word.

SIGLAS

UFRJ - Universidade Federal do Rio de Janeiro

WYSIWYG - *What you see is what you get*

Sumário

1	Introdução	1
1.1	Tema	1
1.2	Delimitação	1
1.3	Justificativa	1
1.4	Objetivos	3
1.5	Metodologia	3
1.6	Descrição	4
2	Fundamentação Teórica	5
2.1	Trabalhos Relacionados	5
2.2	Processamento de Vídeo e Imagens	7
2.3	Identificação de objetos	8
2.4	Métodos de Segmentação de Imagens	11
2.5	Modelo Geométrico de Câmeras	12
2.6	Modelo Geométrico de Uma Praia	13
3	Algoritmo de Medição de Altura das Ondas do Mar	15
3.1	Pré-Processamento	15
3.2	<i>Timestack</i>	16
3.3	Aparato Instrumental	18
3.4	Estabilização de Vídeo	20
3.5	Conversão para Nível de Cinza	21
3.6	Equalização de Histograma	22
3.7	Detecção da Linha de Horizonte	22
3.8	Remoção do Céu	23

3.9	Processamento Principal	24
3.10	Suavização	26
3.11	<i>Thresholding</i>	26
3.12	Segmentação e Detecção de Bordas	28
3.13	Análise e Identificação das Ondas Marítimas	30
3.14	Rastreamento da Linha de Ondas	31
3.15	Identificação das Ondas	33
4	Implementação do Algoritmo	36
4.1	Pré-Processamento	37
4.2	Processamento Principal	46
4.3	Análise e Identificação de Ondas Marítimas	50
5	Dados Experimentais	56
6	Conclusões	59
Bibliografia		60
A	O que é um apêndice	63
B	Encadernação do Projeto de Graduação	64
C	O que é um anexo	66

Lista de Figuras

2.1	Modelo de Câmera Furo-de-Akulha. Fonte: School of Informatics/University of Edinburgh [11].	13
2.2	Modelo Geométrico de uma Praia e Câmera. Fonte: Griffith University [5].	14
3.1	Diagrama de Bloco da etapa de pré-processamento.	15
3.2	<i>Timestack</i> resultante do pré-processamento a partir do vídeo de uma praia.	17
3.3	Círculo que conecta os sensores à central de processamento.	20
3.4	Comparação entre <i>timestack</i> gerado a partir de um vídeo gravado manualmente e o mesmo vídeo estabilizado. O céu foi preservado para facilitar a comparação da estabilização.	21
3.5	Comparação do resultado do passo de <i>thresholding</i> sem a remoção do céu.	23
3.6	Comparação de um <i>frame</i> original do vídeo com o mesmo <i>frame</i> com as bordas detectadas pelo método de Canny. A primeira borda transversal é a linha do horizonte.	24
3.7	<i>Frame</i> equalizado com o céu removido	24
3.8	Diagrama de Bloco da etapa de processamento principal.	25
3.9	Linha de arrebentação detectada pelo processamento principal.	25
3.10	<i>Timestack</i> gerado pelo pré-processamento suavizado pelo filtro gaussiano. .	26
3.11	Comparação da região de espuma detectada com e sem suavização.	27
3.12	Trecho de um <i>timestack</i> após aplicação de um <i>threshold</i> com valores diferentes.	28
3.13	<i>Timestack</i> e <i>threshold</i> do vídeo de um dia nublado.	29
3.14	Comparação dos métodos de Canny e Suzuki para detecção de bordas. . .	30
3.15	Diagrama de Bloco da etapa de análise. Fonte: Autor	31
3.16	Autômato Finito Determinístico que identifica uma onda em uma linha de arrebentação.	34

3.17	Ondas identificadas pelo autômato.	35
4.1	Diagrama das classes Trajectory e Trajectory::Point.	40
4.2	Diagrama da classe Wave.	51
5.1	Comparativo entre onda detectada manualmente e onda detectada automaticamente com interferência de pessoas na areia.	57
5.2	Comparativo entre onda detectada manualmente e onda detectada automaticamente com efeito de <i>spray</i> de espuma.	58
B.1	Encadernação do projeto de graduação.	65

Lista de Tabelas

Capítulo 1

Introdução

1.1 Tema

O tema deste trabalho é o uso de inteligência de máquina e visão computacional para estimar dados sobre ondas em uma cabeça de praia. Neste sentido, o problema a ser resolvido é construir uma solução de hardware e software capaz de inferir a altura de ondas e sua periodicidade.

1.2 Delimitação

Esse estudo será realizado com base em ondas na praia de Itacoatiara, Niterói. Os dados foram coletados inicialmente utilizando uma câmera de celular, e posteriormente utilizando um aparato de hardware desenvolvido para o projeto. A captura de dados usando um dispositivo móvel introduz algumas dificuldades no projeto, como instabilidades nos vídeos capturados. Entretanto, utilizar um hardware em um local fixo apresenta outros problemas, como produzir um hardware que aguente exposição ao tempo e encontrar um local seguro para fixá-lo.

1.3 Justificativa

Hoje, os métodos mais difundidos para medição de ondas do mar são: método gráfico e método sensorial. O método gráfico é utilizado principalmente em campeonatos de surf, onde o tamanho do surfista é usado como referência para calcular

a altura da onda. Já o método sensorial utiliza boias com sensores de movimento, instaladas em um ponto na praia, de forma que a passagem das ondas altera a altura do sensor de movimento, e com isso pode-se calcular a diferença da altura inicial para a final.

Ambos os métodos descritos anteriormente não são práticos de serem implementados em larga escala. O método gráfico necessita sempre de uma referência para realizar a medição de cada onda, além de necessitar o ajuste humano em cada medida. O método sensorial é custoso para adquirir, instalar e manter o equipamento necessário.

Uma das características desejadas para esse sistema de monitoramento é que ele deveria ser autônomo e distribuído, podendo monitorar diversas praias em tempo real. Logo de início ficou claro que medir a altura das ondas de uma forma automatizada e barata é essencial para o sistema, e ainda, que todos os métodos existentes não atendem esses requisitos.

Inicialmente, este projeto começou como parte de um sistema de monitoramento de praias voltado para avaliação da condição de surf, chamado GoSurf. O sistema foi desenvolvido na disciplina Projeto Integrado. Entretanto, no desenvolvimento inicial do sistema não foi possível implementar uma forma de medição de ondas do mar satisfatório.

O monitoramento de praias possui outras aplicações fora do sistema descrito anteriormente. Um sistema similar, também baseado em imagens, foi desenvolvido na Griffin University, Australia, para indicar nível de perigo para nado em praias e que será objeto de estudo na fundamentação teórica deste trabalho [1].

O estudo de processamento de imagens voltado a ondas do mar apresenta um problema interessante em análise de imagens. Como se está trabalhando com imagens reais e altamente dinâmicas, deve-se empregar diversas técnicas para reduzir o ruído nos dados de entrada e garantir uma medição confiável.

Assim, torna-se desejável criar e implementar um algoritmo para medir a altura de ondas do mar utilizando técnicas de processamento de imagens. A análise de cenários naturais dinâmicos é um desafio na área de análise de imagens, uma vez que não se pode alterar o cenário para facilitar a análise desejada. Por isso é necessário um algoritmo inteligente que consiga se adaptar a variações nos dados de entrada para extrair as informações relevantes.

1.4 Objetivos

Informar qual é o objetivo geral do trabalho, isto é, aquilo que deve ser atendido e que corresponde ao indicador inequívoco do sucesso do seu trabalho. Pode acontecer que venha a existir um conjunto de objetivos específicos, que complementam o objetivo geral (tamanho do texto: livre, mas cuidado para não fazer uma literatura romanceada, afinal esta seção trata dos objetivos).

- Construir um aparato de aquisição de vídeos montado em uma praia.
- Adquirir os vídeos e transmiti-los para um servidor.
- Realizar medições (este item pode ser fragmentado em outras metas)
- Disponibilizar os dados.

1.5 Metodologia

As medições obtidas pelo algoritmo serão comparadas com medições manuais tanto nas imagens geradas quanto por estimativas in loco. As medições manuais são feitas na imagem por um operador humano. Ispencionando a imagem, é fácil perceber onde é o ponto de máximo e mínimo de uma onda, e com o auxílio de um programa de edição de imagem pode-se contar o número de pixels entre esses dois pontos. As medições in loco são realizadas no mesmo momento que as imagens são obtidas. Como não é possível medir a onda através de instrumentos, vale-se da experiência da pessoa realizando a aquisição dos dados e de outras no local para estimar a altura das ondas naquele momento. Essa estimativa serve para validar

se o processo de calcular dimensões reais baseadas nas dimensões em pixels está na ordem de grandeza correta.

Para certificar a confiabilidade do algoritmo, serão avaliados inúmeros dados de uma mesma praia, contemplando variações de clima, iluminação, angulação e posicionamento da câmera, maré e ocupação da praia. Todos esses fatores podem interferir com o resultado do algoritmo, principalmente as mudanças de maré e posicionamento da câmera. A escolha de uma posição para a câmera adequada é fundamental para minimizar o efeito dos outros fatores no resultado final.

1.6 Descrição

No capítulo 2 será

O capítulo 3 apresenta ...

Os são apresentados no capítulo 4. Nele será explicitado ...

E assim vai até chegar na conclusão.

Capítulo 2

Fundamentação Teórica

2.1 Trabalhos Relacionados

A análise de praias e mares através de imagens não é algo novo. O uso de câmeras oferece uma grande vantagem em relação a outros tipos de sensores, como descrito por Holland[2], "Técnicas de vídeo são particularmente atraentes na documentação de processos oceanográficos próximos a costa uma vez que a localização subaquática do instrumento (distante da superfície do oceano) alivia algumas das dificuldades associadas com instrumentação *in situ*, como a perturbação de correntes, bioincrustação e deterioração dos sensores em condições adversas de ondas". Entretanto, não existem muitos projetos diretamente ligados à análise de ondas marítimas utilizando processamento de imagem. O desenvolvimento mais notório nessa área é feito na Griffith University, Austrália, onde foram desenvolvidos alguns projetos e técnicas de monitoramento de praias que serão descritos a seguir.

Browne *et al.* [1] descrevem um sistema inteligente que monitora e prediz as condições de uma praia para banho. O objetivo desse sistema é, em caso de perigo, alertar aos banhistas e às autoridades sobre o estado da praia em tempo real. O sistema obtém dados da praia de duas fontes: de câmeras posicionadas *in loco* e de servidores do *Bureau of Meteorology* australiano. As imagens obtidas são pré-processadas, extraíndo dados como tamanho e frequência das ondas e a localização da arrebentação. Dos servidores são obtidos dados em tempo real sobre as marés, vento e *swell*, que são as ondas formadas por tempestades e ventos distantes, e não

por vento local. Uma vez obtidos os dados, o sistema alimenta uma rede neural treinada que determina se a praia é segura para banho.

Outro sistema estudado é chamado de *WavePack* [3], descrito nesse artigo apenas de forma não-técnica. Esse sistema tem como objetivo apenas medir a altura, frequência e localização do momento que uma onda quebra, utilizando câmeras montadas em pontos baixos, dez metros acima da praia. O sistema é descrito em quatro etapas: obtenção das imagens, conversão do *stream* de vídeo em *timestack*, análise do *timestack*, apresentação dos resultados. O artigo ainda compara os resultados obtidos com outras fontes de dados de ondas marítimas, e comprova que o sistema produz dados confiáveis.

O algoritmo implementado no sistema *WavePack* é descrito em um artigo [4] posterior. Esse artigo descreve o método de: 1) identificar a arrebentação, e 2) identificar cada onda individual na arrebentação e calcular a sua altura em *pixels*, filtrando a perturbação de objetos indesejados na imagem (como barcos e pessoas). Em seguida, é discutida e apresentada a relação entre a altura de uma onda medida em *pixels* e a sua altura no mundo real, em metros. Por último, apresenta-se os resultados obtidos, novamente comparados com outros métodos já existentes. Esse método é também apresentado também é apresentado em um artigo [5] mais recente. Esse artigo descreve com maiores detalhes o pré-processamento que ocorre no *timestack*, e a relação entre a altura da onda encontrada em *pixels* e a altura em metros no mundo real.

Essas pesquisas serviram de grande inspiração para o desenvolvimento deste projeto, mostrando que ele é factível. Os resultados desses projetos servirão como parâmetro de validação dos aqui resultados encontrados.

No Brasil não é conhecido algum sistema de monitoramento de praias automatizado. No Rio de Janeiro o site RicoSurf[6] provém um monitoramento manual de algumas praias locais e de cidades próximas, se expandindo até Guarapari, ES. O monitoramento é feito através de boletins disponíveis no site, que normalmente são feitos uma ou duas vezes ao dia por uma pessoa que vai até a praia e reporta a

condição encontrada. Este método é pouco prático se aplicado em grande escala, é subjetivo e demanda uma quantidade cada vez maior de repórteres para analisar cada praia.

2.2 Processamento de Vídeo e Imagens

Processamento de Imagem é uma sequência de operações realizadas em uma ou mais imagens de entrada, resultado em uma imagem de saída ou características extraídas das imagens de entrada. Uma imagem, conforme definido por Rafael Gonzalez [7] é: "[...] uma função bidimensional $f(x, y)$, onde x e y são coordenadas espaciais (plano), e a amplitude de f de qualquer par de coordenadas (x, y) é chamada de intensidade ou nível de cinza da imagem naquele ponto.".

Neste projeto, como em outras aplicações de oceanografia [8] [2], imagens estáticas do objeto de estudo são pouco produtivas para extrair informações. Por isso, uma solução encontrada foi capturar e trabalhar com um vídeo da arrebentação de uma praia, ao invés de usar apenas fotografias.

Um vídeo é definido como uma sequência temporal de imagens. De forma análoga a imagem, um vídeo pode ser descrito como uma função tridimensional $g(x, y, t)$, onde x e y são coordenadas espaciais, como em uma imagem, e t é uma coordenada temporal. A amplitude g é a intensidade ou nível de cinza do ponto (x, y) no instante de tempo t .

A principal vantagem de utilizar um vídeo é eliminar o problema de identificar qual é o melhor momento para analisar a onda fotografada. Utilizando uma sequência de imagens, é mais fácil de determinar qual foi o momento em que a onda estava maior, e ai sim realizar a medição de sua altura. Será demonstrado adiante que o *timestack* também ajuda a tornar as imagens analisadas mais uniformes, criando regiões bem definidas para segmentação.

2.3 Identificação de objetos

Para extrair os dados desejados de uma imagem, é necessário separar o que é o objeto de estudo – no caso específico deste projeto, uma onda do mar – do restante da imagem. Isto é, através de transformações na imagem original deseja-se identificar o que é o primeiro plano e o plano de fundo. Este processo de extração de objetos ou planos de uma imagem é chamado na literatura de segmentação [7].

Antes de aplicar as técnicas de segmentação, precisa-se melhorar a imagem. O processo de aprimoramento de imagens, segundo Rafael Gonzalez [7], é fundamental para torná-la adequada para a aplicação desejada. O aprimoramento permite realçar alguma característica de interesse da imagem, enquanto minimiza a presença de outras características não relevantes.

Os métodos de aprimoramento de imagens são classificados em dois domínios: domínio espacial e domínio da frequência.

Falar sobre métodos no domínio da frequência

O aprimoramento no domínio espacial trabalha com os próprios *pixels* de uma imagem, e podem ser representados pela expressão: $g(x, y) = T[f(x, y)]$, onde $g(x, y)$ é a imagem transformada, $f(x, y)$ é a imagem original e $T[\cdot]$ é a transformação que será aplicada na imagem original.

Alguns métodos de aprimoramento de imagens serão utilizados nesse projeto: transformações em níveis de cinza, filtros de nitidez (*sharpening*), filtros de suavização (*blur*), filtros de detecção de bordas. Esses métodos são importantes para melhorar o contraste entre as regiões da imagem analisada e reduzir o ruído, facilitando assim o processo de segmentação que será realizado posteriormente.

As transformações em níveis de cinza englobam métodos que alteram o contraste de uma imagem. Em geral, são métodos onde cada pixel (x, y) da saída depende apenas do pixel correspondente da entrada, ou seja, não é influenciado pelos seus vizinhos. O principal método que será utilizado nesse projeto é o de *thresholding*, onde a transformação aplicada na entrada é da forma:

$$T[f(x, y)] = \begin{cases} f(x, y), & \text{se } f(x, y) > C \\ 0, & \text{se } f(x, y) < C \end{cases}$$

Onde C é uma constante escolhida arbitrariamente. A utilidade dessa função é facilitar a definição das regiões da imagem, quais fazem parte do céu, do mar e da arrebentação.

Os filtros de suavização, também conhecidos como filtros de *blur* são importantes para reduzir o nível de ruído na imagem. Outra função importante para esse projeto é homogeneizar as regiões da imagem, isto é, eliminar, ou pelo menos reduzir, pontos mais claros em regiões escuras, ou pontos escuros em regiões claras. Dessa forma, os métodos de *thresholding* são mais efetivos.

Neste projeto o filtro de suavização utilizado é o filtro passa-baixas gaussiano. Este filtro, como definido em [7], possui a seguinte forma:

$$H(u, v) = e^{-D^2(u, v)/2D_0^2}$$

$D(u, v)$ é definido como a distância da origem da transformada de Fourier, e D_0^2 é definido como σ^2 , ou a variância da curva gaussiana.

Segundo Bernd Jähne [8], um filtro de suavização deve atender algumas propriedades específicas para ser aplicável a identificação de objetos e extração de dados de uma imagem. As propriedades são:

1. Desvio de fase zero, isto é, o filtro não deve causar desvio na fase da imagem a fim de não alterar a posição dos objetos na mesma.
2. Preservação da média, isto é, a soma de todos os fatores da máscara no domínio espacial deve ser igual a 1.
3. Monotonicidade, isto é, a função de transferência do filtro de suavização deve decrescer monotonicamente.
4. Equidade, isto é, a função de transferência deve ser isotrópica, a fim de não privilegiar nenhuma direção na imagem.

Por último, uma classe de métodos de suma importância são os métodos de nitidez, ou *sharpening*. A função desses métodos é aumentar o contraste da imagem na fronteira de regiões, isto é, onde ocorre uma descontinuidade de *pixels*. O método utilizado neste projeto é o método Laplaciano. Segundo Gonzalez [7], o Laplaciano de uma imagem $f(x, y)$ é definido por:

$$\nabla^2 f = \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

As derivadas parciais da equação acima podem ser escritas na forma discreta:

$$\frac{\partial^2 f}{\partial x^2} = f(x+1, y) + f(x-1, y) - 2f(x, y)$$

e

$$\frac{\partial^2 f}{\partial y^2} = f(x, y+1) + f(x, y-1) - 2f(x, y)$$

Dessa forma, o Laplaciano bidimensional discreto é dado por:

$$\nabla^2 f = f(x+1, y) + f(x-1, y) + f(x, y+1) + f(x, y-1) - 4f(x, y)$$

A aplicação de um operador Laplaciano em uma imagem resulta em suas apenas as suas descontinuidades. Pela definição do Laplaciano discreto, é fácil perceber que regiões com valores intensidades próximas se anulam, resultado em um Laplaciano próximo de zero, enquanto regiões com descontinuidade a diferença de intensidade se reforça. Para recuperar as regiões anuladas pelo Laplaciano e enfim aplicar o efeito de *sharpening* na imagem, basta somar o resultado do Laplaciano com a imagem original:

$$g(x, y) = f(x, y) + \nabla^2 f(x, y)$$

Onde $g(x, y)$ é a imagem de saída e $f(x, y)$ é a imagem de entrada.

Equalização de Histograma...

[Adicionar teoria sobre histogramas e equalização de histograma](#)

2.4 Métodos de Segmentação de Imagens

Segmentação é definido como um processo que, a partir de uma imagem de entrada, a subdivide em objetos ou nas suas regiões constituintes. O nível de subdivisões que serão obtidas depende de cada aplicação, e segundo Gonzalez: ”A segmentação deve parar quando os objetos de interesse em uma aplicação estejam isolados”[7]. Além diso, Gonzalez continua: ”A segmentação de imagens não-triviais é uma das tarefas mais difíceis em processamento de imagem”[7].

O primeiro passo para realizar a segmentação é aplicar um método de detecção de bordas. O método escolhido para esse projeto é o método de Canny [9], que é hoje o método mais eficaz de detecção de bordas [10]. Esse método basea-se em criar um modelo matemático para uma borda e definir três restrições que o modelo deve atender, e então utilizar um método numérico para otimizar o modelo. As restrições definidas por Canny [9] são:

1. Boa detecção. Deve haver uma baixa probabilidade de detectar bordas inexistentes ou não detectar bordas existentes.
2. Boa localização. A localização das bordas detectadas deve estar próxima da localização das bordas reais.
3. Resposta única para cada borda. Uma borda não pode ser detectada multiplas vezes.

Outro método de detecção de bordas considerado foi o método de Sobel [7], que é utilizado em [4] para evidenciar bordas verticais. Uma diferença importante entre os dois método é que o método de Sobel atua em cada direção separadamente, podendo-se somar o resultado de cada direção para obter uma resposta unificada, enquanto o método de Canny atua em todas as direções simultaneamente. Devido a maior presença de ruído nas imagens capturadas, o método de Canny mostrou-se mais eficaz em eliminar o ruído e detectar apenas as bordas desejadas.

O segundo passo do processo de segmentação é aplicar um método de extração de regiões. Este projeto utiliza o método de watershed [7] para tal tarefa. O algoritmo

de watershed baseia-se na idéia de enxergar a imagem em três dimensões como um vales que serão ”inundados”, sendo x e y coordenadas espaciais e a intensidade de cada pixel $g(x, y)$ a ”altura” de cada ponto neste vale. Os pontos de mínimo locais, ou seja, os pontos onde $g(x, y)$ é mínimo localmente, consideramos este um ponto mais baixo de um vale, e a região ao redor que será segmentada sua zona de influência. A partir dos pontos de mínimo locais inunda-se a imagem, preenchendo-a com valores gradativamente maiores de intensidade de pixel em sua zona de influência, como seria o nível de água subindo em uma inundação. Eventualmente, a inundação da zona de influência de um ponto mínimo local irá ”vazar” para a zona de influência de outro ponto mínimo local. Quando isso ocorre, constrói-se uma ”barragem” com largura de um pixel entre as duas zonas de influência, para evitar o vazamento. Quando o nível de inundação atinge o ponto máximo da imagem, isto é, o valor máximo de $g(x, y)$, o algoritmo de watershed é interrompido. As regiões segmentadas serão definidas pelas ”barragens” formadas durante a execução do algoritmo. Na forma descrita acima, o algoritmo de watershed pode levar a segmentação de mais regiões que se desejava originalmente. Isso é corrigido através do uso de marcadores, indicadores de quais são os pontos de mínimo local que serão utilizados pelo algoritmo.

2.5 Modelo Geométrico de Câmeras

Todos os métodos descritos anteriormente foram apresentados com o intuito de aplicá-los no cálculo da altura de uma onda em pixels. Para obter a altura em metros, precisa-se de um modelo matemático que relacione as duas medidas. O modelo mais simples de uma câmera é o modelo furo-de-agulha. Segundo Fusiello [11], o modelo é descrito pelo seu centro ótico C e o plano da imagem. A distância entre C e o centro do plano da imagem, é chamado de distância focal f . O eixo perpendicular ao plano da imagem que passa pelo centro ótico é chamado de eixo principal.

Este modelo apresenta uma forma simples de calcular a relação entre a altura aparente de um objeto em uma imagem e a sua altura real, dependendo apenas da distância focal da câmera e da distância real do objeto. Esta relação é:

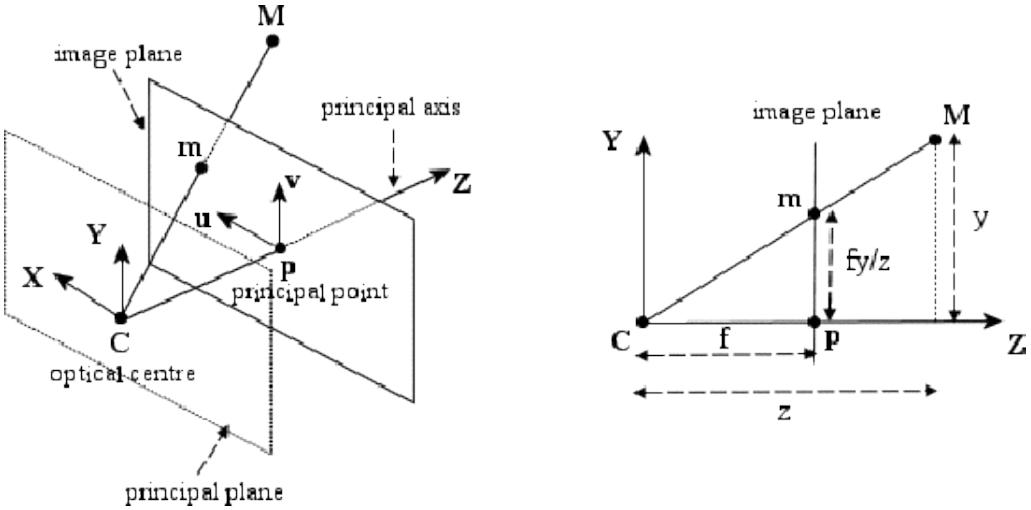


Figura 2.1: Modelo de Câmera Furo-de-Agulha. Fonte: School of Informatics/University of Edinburgh [11].

$$h_{real} = z * h_{aparente} / f$$

onde z é a distância real do objeto, f é a distância focal, h_{real} e $h_{aparente}$ são, respectivamente, a altura real e a altura aparente.

2.6 Modelo Geométrico de Uma Praia

Browne *et al.* [5] descreve um modelo geométrico simplificado de uma praia, onde temos uma relação direta entre a altura em pixels de uma onda e a sua altura real. Para isso, são necessários três parâmetros de montagem da câmera: o ângulo da câmera em relação ao ponto de montagem (ϕ_t), o ângulo de *zoom* da câmera (ϕ_z) e a altura da câmera em relação ao nível do mar (h_c).

A altura real de uma onda na imagem pode ser calculada pela equação:

$$h_w = h_c \left(1 - \frac{\tan(\theta_i)}{\tan(\theta_j)} \right)$$

Onde h_w é a altura real de uma onda, θ_i é o ângulo calculado do ponto mais baixo de uma onda e θ_j é o ponto mais alto de uma onda. Utilizando uma câmera com baixa distorção, é possível calcular o ângulo θ_k de um pixel (x, k) através da equação:

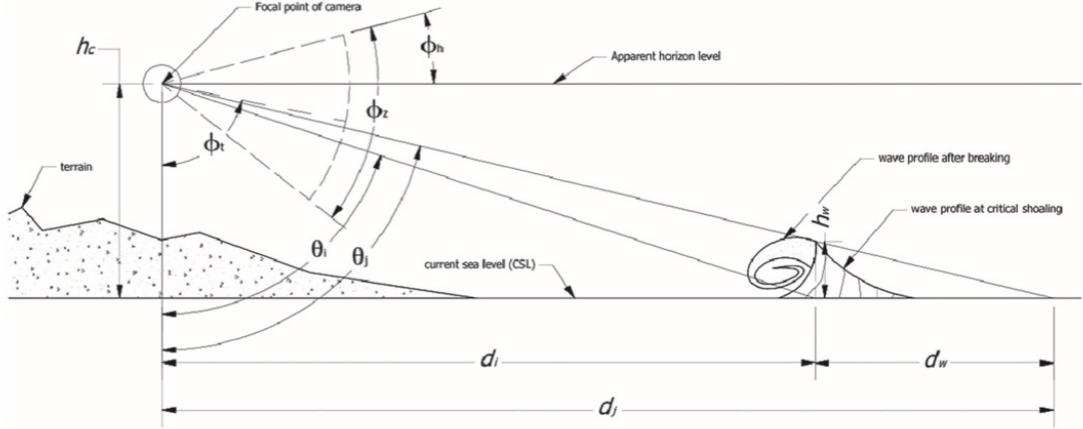


Figura 2.2: Modelo Geométrico de uma Praia e Câmera. Fonte: Griffith University [5].

$$\theta_k = \phi_t - \frac{\phi_z}{2} + \frac{k\phi_z}{V}$$

onde V é a altura da imagem em pixels. A vantagem deste modelo em relação ao modelo furo-de-agulha é ser não só mais preciso, mas não depender da distância da câmera até a onda (valor z do Modelo de Câmera Furo-de-Agulha), que pode ser altamente variável.

Capítulo 3

Algoritmo de Medição de Altura das Ondas do Mar

Os métodos apresentados no Capítulo 2 agora serão utilizadas em conjunto para estimar a altura de ondas do mar. Este algoritmo é composto de três etapas: o pré-processamento, o processamento principal, e a análise da imagem resultante. Estas etapas serão detalhadas nas seções a seguir.

3.1 Pré-Processamento

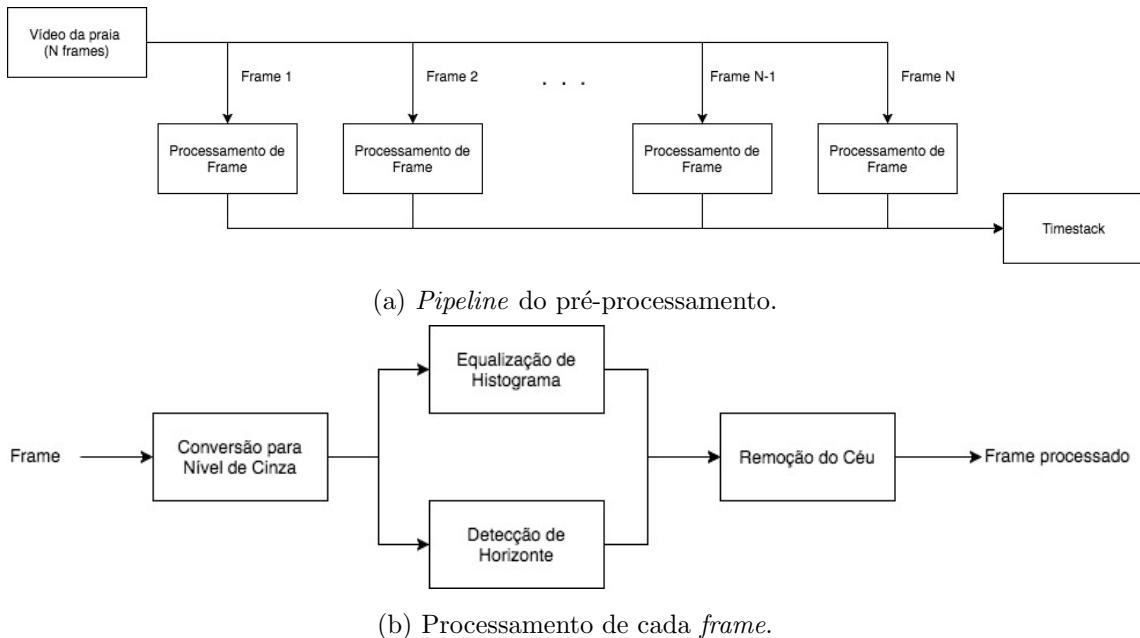


Figura 3.1: Diagrama de Bloco da etapa de pré-processamento.

A etapa de pré-processamento tem como objetivo transformar os dados de entrada (o vídeo capturado de uma praia) em uma imagem *timestack* pronta para ser processada e analisada pelos passos subsequentes. O pré-processamento atua sobre cada *frame* do vídeo, realizando as seguintes operações: conversão para nível de cinza, equalização de histograma, detecção da linha de horizonte, remoção do céu. Após operar cada *frame* do vídeo é gerado um *timestack* das imagens processadas. A figura 3.1 ilustra os passos realizados.

O passo mais importante nessa etapa é a criação do *timestack*, que é uma representação espaço-temporal de um vídeo. O *timestack* é a imagem que será processada pelo processamento principal e analisada para extração dos dados das ondas do mar. Os demais passos servirão para garantir a confiabilidade do *timestack* e o prepararão para a etapa de processamento principal.

O pré-processamento inicia a partir do vídeo capturado de uma praia. Cada *frame* do vídeo é processado, primeiro convertendo o *frame* para uma imagem em níveis de cinza, pois todos os processamentos posteriores só operam sobre o nível de intensidade de cada *pixel*. A seguir, equaliza-se o histograma do *frame*, a fim de normalizar a iluminação da imagem. Esta operação é necessária para possibilitar que filmagens realizadas em qualquer hora do dia podem ser tratadas da mesma forma. Os passos seguintes são a detecção da linha do horizonte e a remoção do céu. Para detectar o horizonte, analisa-se o original em nível de cinza. Uma vez detectada a linha do horizonte, o céu é removido simplesmente alterando todos *pixels* acima da linha para intensidade zero. O último passo é gerar o *timestack* a partir dos *frames* do vídeo. O *timestack* é gerado selecionando a coluna central de cada *frame*, acumulando-as sequencialmente em uma única imagem. Todos esses passos serão descritos detalhadamente a seguir.

3.2 *Timestack*

Um *timestack* é uma representação bidimensional de um vídeo, isto é, trata-se de uma forma de transformar tal vídeo em uma só imagem. O *timestack* é útil para analisar tais vídeos pois é possível olhar para uma sequência completa de

quadros observando uma única imagem. Dessa forma, pode-se aplicar métodos de processamento de uma única imagem *timestack*, em vez de todo o vídeo.

Para construir um *timestack*, é necessário fixar uma das dimensões espaciais de cada imagem do vídeo, obtendo assim um conjunto de funções unidimensionais. O *timestack* deste vídeo é definido então como uma função bidimensional $s_Y(x, t)$ ou $s_X(t, y)$, onde x e y são coordenadas espaciais; t é uma coordenada temporal; as amplitudes s_X e s_Y são a intensidade ou nível de cinza do vídeo $g(x, y, t)$ quando são fixados os valores $x = X$ e $y = Y$, respectivamente. Dessa forma, a relação entre um *timestack* e um vídeo é dada por: $s_Y(x, t) = g(x, Y, t)$ e $s_X(t, y) = g(X, y, t)$.

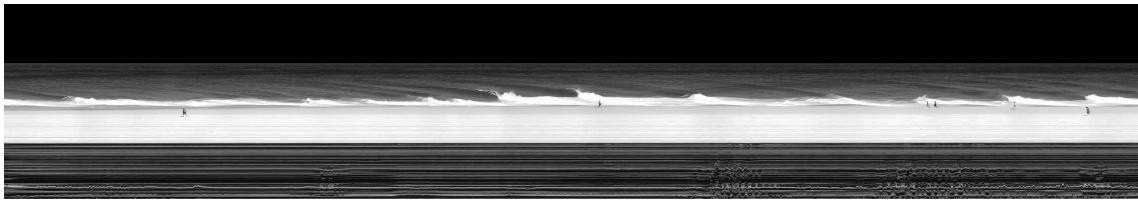


Figura 3.2: *Timestack* resultante do pré-processamento a partir do vídeo de uma praia.

Note que, na prática, a coordenada temporal t cumpre o papel de uma coordenada espacial quando o *timestack* é exibido como uma imagem, mas sua interpretação continua sendo temporal. Como é possível notar na figura 3.2, cada onda aparenta crescer a medida que se desloca da esquerda para a direita e de cima para baixo. Isso ocorre pois o deslocamento horizontal representa a evolução temporal da onda, representa eventos que estão ocorrendo sequencialmente, primeiro a onda cresce até que chega no seu ponto máximo e quebra. O deslocamento vertical representa o avanço da onda até quebrar na areia da praia.

É importante lembrar que a perda de uma dimensão resulta em perda de informação no vídeo, entretanto, nos casos em que o vídeo se mantém constante na dimensão espacial fixada não há perda de informação. Expandindo esse conceito, nos casos em que o vídeo varie muito pouco em uma de suas dimensões espaciais, não há necessariamente uma perda de conteúdo significativa.

A análise de ondas marítimas apresenta condição similar a descrita anteriormente. Com a posição da câmera escolhida com cuidado, isto é, a zona de arrebentação deve

centralizada na imagem, tanto horizontalmente quanto verticalmente. A câmera deve estar perpendicular em relação a zona de arrebentação, e montada em um ponto acima do nível da praia, de forma que banhistas na areia não atrapalhem a visão da zona de arrebentação. Assim, uma onda quebrando ocupará maior parte horizontal da imagem. Além disso, não é importante para a análise desejada entender o tamanho horizontal da onda, apenas o seu tamanho vertical. Precisa-se apenas determinar o seu ponto mais baixo e seu ponto mais alto. Sendo assim, o *timestack* mostra-se um método adequado de análise de vídeos para esse projeto.

3.3 Aparato Instrumental

Como mencionado anteriormente este projeto foi concebido como parte do sistema de monitoramento de praias *GoSurf*, sendo o monitoramento por imagens apenas uma parte do sistema. Esse sistema foi criado para a disciplina Projeto Integrado no ano letivo 2015.1, em conjunto com outro aluno, Filipe Barretto. Para este fim foi desenvolvido um aparato instrumental que adquire dos dados e os analisa previamente por meio de uma central de processamento de dados, e por fim os envia para um servidor na nuvem. O servidor finaliza o processamento dos dados e disponibiliza para os resultados para os usuários por um aplicativo *Android*.

No projeto de graduação as imagens foram adquiridas de duas formas: através de câmeras de *smartphones* comuns e através da câmera montada no *hardware* apresentado anteriormente. Embora para este projeto a câmera de *smartphones* atenda o requisito, o aparato instrumental é fundamental para o projeto *GoSurf* completo porque agrupa sensores extras além da câmera. Outra vantagem do aparato sensorial é a capacidade de realizar o pré-processamento localmente, o que reduz o volume de dados que devem ser enviado ao servidor. Por exemplo, um vídeo de 125MB é reduzido para uma imagem de 657KB.

A central de processamento utilizada no projeto GoSurf e no projeto de graduação é um Raspberry Pi, um pequeno computador implementado em uma única placa. O Raspberry Pi é um computador completo, sendo possível instalar diversos sistemas operacionais diferentes. Neste projeto seguiu-se a recomendação oficial e foi utilizado

o sistema operacional Raspbian - um sistema operacional Linux baseado em Debian. Por ser baseado em Linux, pode-se utilizar a biblioteca de processamento gráfico OpenCV para agilizar a implementação do projeto de graduação.

Um dos motivos para a escolha do Raspberry Pi foi a facilidade de integrar com uma câmera. Existe um módulo de câmera oficial para o Raspberry Pi [12], que foi a câmera escolhida para este projeto. Utilizando este módulo e o sistema operacional Raspbian, a integração com o projeto é trivial, basta conectar a câmera à central de processamento. Esta câmera é capaz de capturar vídeos com resolução 1080x720 a 30 *frames* por segundo, a mesma qualidade de vídeo utilizada em capturas de vídeo com *smartphones*. O módulo de câmera possui ponto focal fixo na faixa de 1 metro até o infinito, o que não é um problema para o algoritmo pois os objetos de estudo, as ondas do mar, estão localizados nesta faixa. O campo de visão vertical desta câmera é de 41.41 graus (mais ou menos 0.11 graus). Este dado será importante durante a conversão da altura das ondas estimada em *pixels* para metros.

O Raspberry Pi ainda conta com um conjunto de pinos de entrada e saída de uso genérico (*GPIOs*). Uma desvantagem do Raspberry Pi com relação a outras centrais de processamento é que ele não possui *GPIOs* analógicos, somente pinos digitais. Entretanto, como será comentado a seguir, é possível contornar esta desvantagem utilizando conversores analógico-digitais. Estes pinos foram utilizados para ligar os sensores periféricos no projeto GoSurf, um termômetro/higrômetro e um anemômetro. Um circuito simples (Figura 3.3) foi projetado para ligar os sensores à central de processamento. O termômetro / higrômetro utilizado foi o DHT22, que pode ser lido diretamente por um pino digital. Por este motivo ele pode ser ligado diretamente aos pinos da central de processamento, acrescentando somente um resistor *pullup* de $1k\Omega$ entre sua alimentação e o pino de leitura. O anemômetro escolhido foi o C2192, que possui um pino de leitura analógico. Para utilizá-lo com o Raspberry Pi foi necessário adicionar um conversor analógico-digital (MCP 3201). Este conversor foi escolhido por apresentar resolução adequada ao projeto (12 *bits*) e possuir interface de saída SPI (*Serial Peripheral Interface* ou Interface Periférica Serial). O Raspberry Pi implementa uma comunicação SPI em pinos específicos, o que torna fácil integrar os dois componentes. Por fim, foi necessário adicionar uma

alimentação externa ao circuito, pois o anemômetro precisa de uma alimentação de 7 a 12 volts e a central de processamento só é capaz de fornecer 5V. Foram adicionados um par de pilhas de 4.5V, resultando em uma alimentação total de 9V que atende ao requisito.

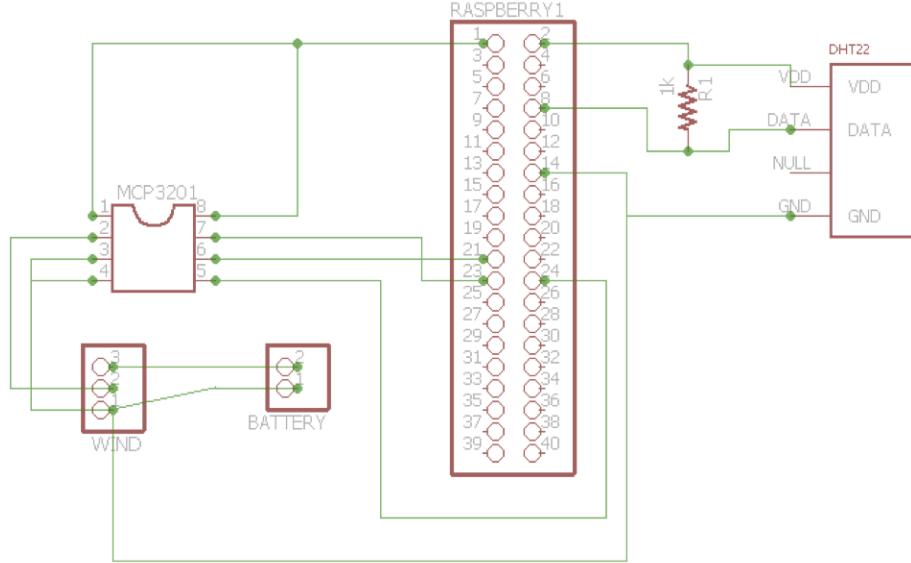


Figura 3.3: Circuito que conecta os sensores à central de processamento.

3.4 Estabilização de Vídeo

A estabilização do vídeo é fundamental para a criação de um *timestack* fiel ao cenário real. Esta necessidade surge uma vez que o *timestack* é criado selecionando a coluna central de cada *frame* do vídeo, tornando importante que esta coluna corresponda ao mesmo conjunto de *pixels* de quadro a quadro. Caso a câmera se desloque horizontalmente no decorrer do vídeo, o conjunto de *pixels* não será o mesmo, e com isso o *timestack* formado não será uma representação fiel em duas dimensões do vídeo capturado. A estabilidade na direção vertical é importante para que a estimativa de altura das ondas seja fiel a altura real. Um deslocamento vertical da câmera resulta em um deslocamento das colunas do *timestack*, que por sua vez podem embutir erros na identificação dos pontos de máximo e mínimo de cada onda.

Primeiro experimentou-se realizar a estabilização do vídeo utilizando técnicas de processamento de imagens. Foi testado um procedimento simples descrito por Nghia

Ho [13]. Embora este procedimento apresente bons resultados para vídeos gravados manualmente (figura 3.4a e 3.4b), foi observado que quando aplicado o procedimento em vídeos gravados já com alta estabilidade o procedimento não apresentava resultados significativos (Figura 3.4c e 3.4d). Entretanto, o processo de estabilização é custoso, aumentando o tempo de execução do pré-processamento (na figura 3.4 o tempo de execução do pré-processamento foi 8.5 vezes maior.). Como o pré-processamento será executado em um dispositivo com poder de processamento limitado, a performance desta etapa é crucial. Então, concluiu-se que basta utilizar um ponto de montagem para a câmera, como um tripé, para atingir a estabilidade necessária para a extração fiel dos dados desejados.



(a) *Timestack* gerado a partir de um vídeo gravado manualmente.



(b) *Timestack* gerado a partir de um vídeo gravado manualmente e estabilizado por *software*.



(c) *Timestack* gerado a partir de um vídeo gravado com ponto de montagem fixo.



(d) *Timestack* gerado a partir de um vídeo gravado com ponto de montagem fixo e estabilizado por *software*.

Figura 3.4: Comparação entre *timestack* gerado a partir de um vídeo gravado manualmente e o mesmo vídeo estabilizado. O céu foi preservado para facilitar a comparação da estabilização.

3.5 Conversão para Nível de Cinza

Após gerar o *timestack*, o mesmo é convertido para nível de cinza. Os passos e etapas subsequentes do algoritmo ora proposto não necessitam trabalhar com cores,

apenas a intensidade de cada *pixel* é necessária para estimar a altura das ondas. O processamento das imagens se torna mais eficiente, pois a representação de uma imagem em nível de cinza possui um terço do tamanho de representação de imagem em cores. Com isso, o algoritmo utilizara menos memória para executar e seu tempo de execução é reduzido.

A conversão para nível de cinza é realizada de acordo com a seguinte fórmula [5]:

$$I(t, v) = 0.35R(t, v) + 0.5G(t, v) + 0.15B(t, v)$$

onde $I(t, v)$ é a intensidade do pixel na posição (t, v) na imagem convertida, e $R(t, v)$, $G(t, v)$ e $B(t, v)$ são, respectivamente, os valores das componentes vermelhas, verdes e azuis do pixel (t, v) da imagem original.

3.6 Equalização de Histograma

A equalização de histograma tem como funcionalidade normalizar o nível intensidade da imagem. Desta forma, reduz-se tanto o efeito do clima (como a variação da iluminação em dias nublados e dias de céu aberto) quanto o efeito do horário do dia (como a variação da iluminação entre o começo da manhã e o meio do dia). Além disso, nuvens e objetos externos podem alterar a iluminação durante a aquisição do vídeo em si. Portanto, deve-se equalizar o histograma de cada *frame* do vídeo antes de gerar o *timestack*.

3.7 Detecção da Linha de Horizonte

Uma etapa importante é a detecção da linha de horizonte a fim de viabilizar a detecção e remoção do céu na imagem. Esta operação é utilizada para facilitar o passo de *thresholding* na etapa de processamento de imagem. Em alguns cenários, é difícil distinguir a região de céu da região de arrebentação utilizando apenas o nível de intensidade dos *pixels* de cada região, pois a faixa de valores de intensidade de cada região se sobrepõem. A figura 3.5a exibe um *timestack* com o céu incluído após aplicado um *threshold* com valor de limite 150. Pode-se observar que parte do céu não foi removido pelo *threshold* (faixa branca na parte superior da imagem). Já a figura

3.5b exibe um *timestack* com o céu incluído após aplicado um *threshold* com valor de limite 210. Com este valor o céu foi removido completamente, mas a delimitação da região de espuma não foi bem definida, dificultando ou até impossibilitando sua identificação (observe a faixa preta estreita presente na faixa branca da imagem). Por este motivo, a segmentação é baseada não só no nível de intensidade dos *pixels* mas como na sua localização na imagem. A detecção da linha de horizonte se mostra fundamental para obter localização desta região.

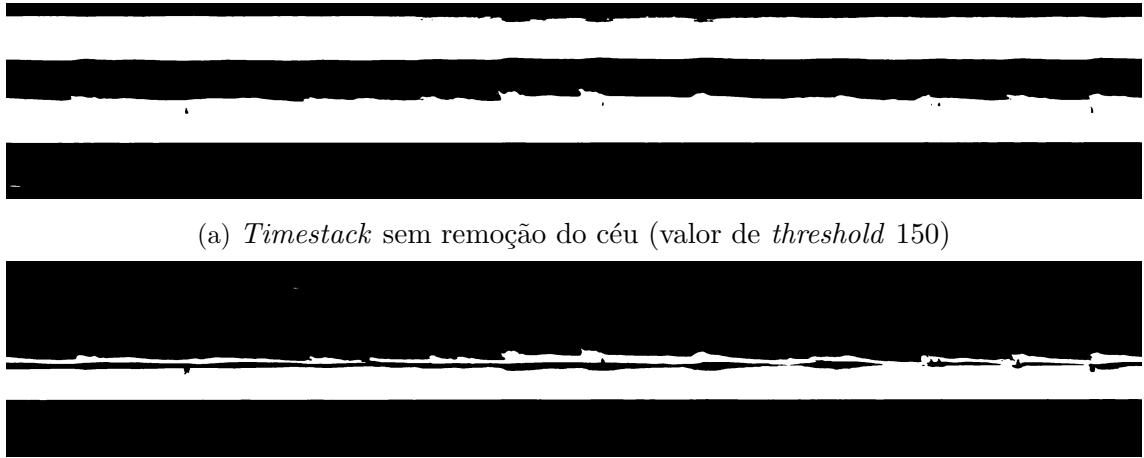


Figura 3.5: Comparação do resultado do passo de *thresholding* sem a remoção do céu.

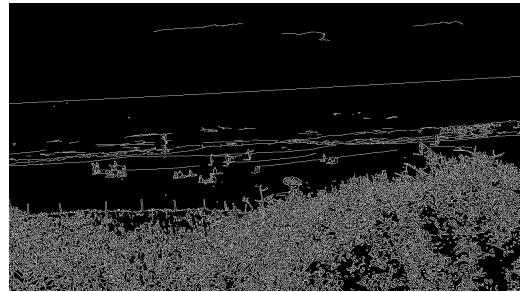
A detecção do céu é realizada utilizando um detector de bordas de Canny para identificar a borda entre o céu e o mar em um *frame* do vídeo (Figura 3.6). Então, procura-se a primeira borda transversal detectada pelo método de Canny, através do algoritmo descrito em 3. A borda detectada corresponde a linha do horizonte. Todos os *pixels* acima dessa linha são considerados como *pixels* da região do céu, e serão removidos no passo seguinte.

3.8 Remoção do Céu

O último passo da etapa de pré-processamento é a remoção do céu. Este passo utiliza duas entradas, a linha do horizonte detectada pelo passo anterior, e o *frame* equalizado. A partir da linha do horizonte, considera-se todos os *pixels* do *frame* equalizado acima dessa linha como preto. A figura 3.7 exibe um *frame* equalizado do vídeo original com o céu removido.



(a) *Frame* do vídeo original.



(b) *Frame* do vídeo com bordas detectadas pelo método de Canny.

Figura 3.6: Comparação de um *frame* original do vídeo com o mesmo *frame* com as bordas detectadas pelo método de Canny. A primeira linha transversal na figura b é a linha do horizonte.



Figura 3.7: *Frame* equalizado com o céu removido

Como dito anteriormente, sem a remoção do céu, é difícil definir um valor único de *threshold* que será utilizado na etapa de processamento principal. Uma tarefa essencial é ser capaz de separar o céu, o mar e a faixa de espuma da arrebentação. Com o céu removido, ao operação de *threshold* produzirá resultados mais adequados para dar suporte às etapas de medição.

3.9 Processamento Principal

A etapa de processamento principal tem como objetivo extrair a linha de arrebentação (figura 3.9) de um *timestack* gerado pelo pré-processamento. A linha de arrebentação é definida como a linha que delimita a região de espuma do restante

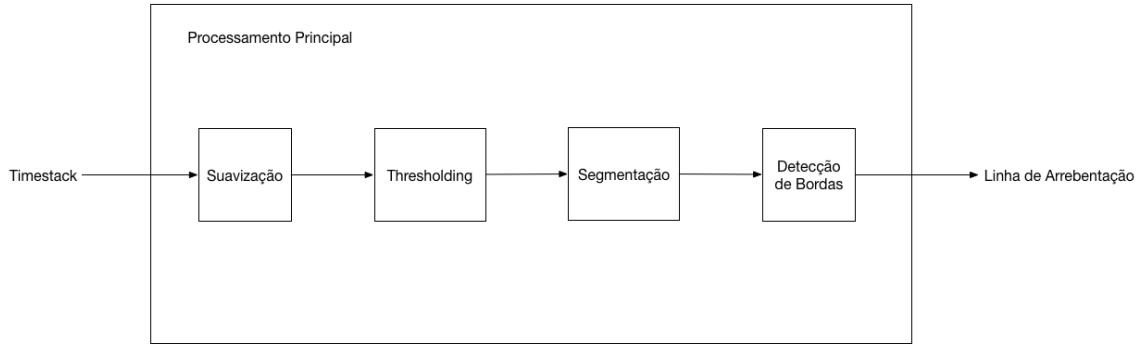
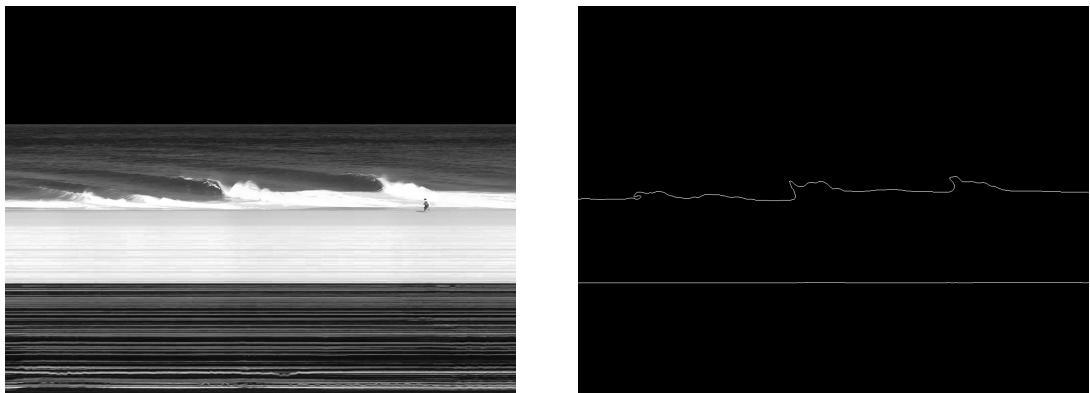


Figura 3.8: Diagrama de Bloco da etapa de processamento principal.

do mar. Uma característica marcante dessas duas regiões é que a região de espuma apresenta a maioria dos *pixels* com intensidade alta, enquanto o restante do mar apresenta a maioria dos *pixels* com intensidade baixa. Esta característica será explorada para separar a região de espuma do restante da imagem. Em seguida, a borda superior desta região, que corresponde a linha de arrebentação, será analisada para identificar as ondas do mar presentes na imagem e medir a sua altura e periodicidade.



(a) *Timestack* gerado pelo pré-processamento.

(b) Linha de arrebentação detectada pelo processamento principal.

Figura 3.9: Linha de arrebentação detectada pelo processamento principal.

As etapas de suavização, *thresholding*, segmentação e detecção de bordas, que compõe o Processamento Principal, serão descritas nas seções a seguir.

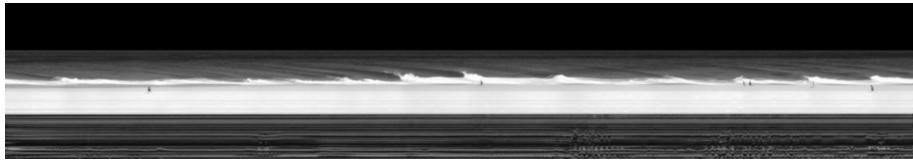


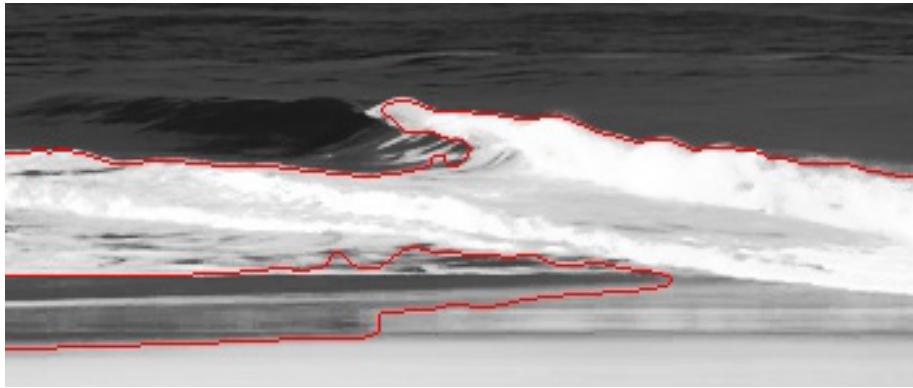
Figura 3.10: *Timestack* gerado pelo pré-processamento suavizado pelo filtro gaussiano.

3.10 Suavização

A primeira etapa de suavização é realizada utilizando um filtro passa-baixas gaussiano, implementado no domínio do espaço com máscara de tamanho 15, valor escolhido empiricamente após experimentar com imagens de diversos dias diferentes. A figura 3.10 ilustra um *timestack* suavizado pelo processo descrito. Este efeito de suavização é importante para reduzir o nível de ruído na imagem, facilitando a identificação das bordas relevantes nos passos subsequentes. Outro efeito desejado é homogeneizar o nível de intensidade das regiões da imagem, de forma que o passo de *thresholding* consiga segmentar a região de arrebentação do *timestack* consistentemente. A figura 3.11 ilustra a diferença na segmentação da região de espuma com e sem o passo de suavização. A região de espuma detectada está marcada em vermelho. Nota-se que na figura 3.11b a região de espuma detectada é influenciada por pequenas faixas de alta intensidade espalhadas na região do mar, enquanto a figura 3.11a consegue filtrar essa influência e detectar uma região de espuma mais uniforme.

3.11 *Thresholding*

Thresholding é o primeiro passo para realizar a segmentação. Neste passo separa-se a região de espuma do restante do mar, alterando todos os *pixels* com intensidade abaixo de 150 para 0, e todos os *pixels* acima deste valor para o valor de intensidade máximo. O valor limite foi escolhido empiricamente após a análise de diversas imagens, contemplando dias variados. A figura 3.12 compara a região de espuma detectada após o passo de *threshold* com valores diferentes. Pode-se notar na figura 3.12a uma borda mais estreita sobre a região de espuma observada, enquanto a figura 3.12b exibe uma região detectada não condizente com o resultado esperado. Inicialmente foi considerado um método de decisão do valor de *threshold* dinâmico,



(a) Região de espuma detectada em um *timestack* suavizado.



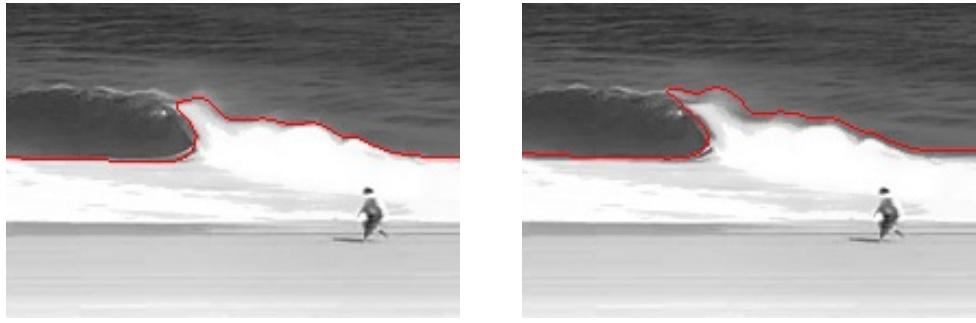
(b) Região de espuma detectada em um *timestack* não-suavizado.

Figura 3.11: Comparação da região de espuma detectada com e sem suavização.

mas como discutido previamente, os passos de equalização e remoção do céu realizados durante o pré-processamento permitem utilizar um valor de *threshold* estático, independente da condição de iluminação no momento da aquisição do vídeo.

Entretanto, dias muito nublados podem causar problemas para o *thresholding*, como pode ser visto na figura 3.13. Nesta figura os valores de intensidade da região de espuma e da região do mar no entorno das ondas se sobrepõem. Neste caso, o *threshold* com valor estático não é capaz de separar as duas regiões, impossibilitando a identificação das ondas do mar nos passos subsequentes.

É importante notar que o *thresholding* apenas isola as regiões cujo nível de intensidade está acima de um certo valor. Mais de uma região pode ser isolada durante esse processo, por exemplo, partes da região do mar pode estar mais clara que o esperado, e por isso serão preservadas pelo *threshold*. O passo de segmentação é responsável por filtrar as regiões isoladas pelo *thresholding*.



(a) Região de espuma detectada usando *threshold* no valor 150.

(b) Região de espuma detectada usando *threshold* no valor 120.

Figura 3.12: Trecho de um *timestack* após aplicação de um *threshold* com valores diferentes.

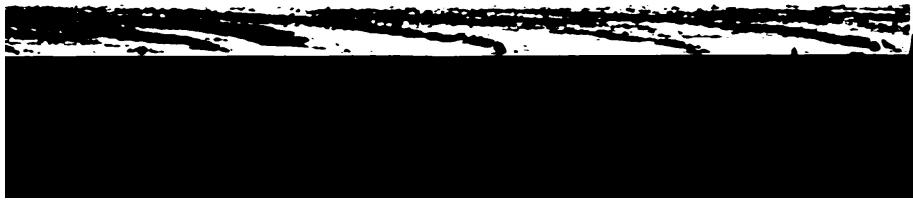
3.12 Segmentação e Detecção de Bordas

O último passo do processamento principal é a segmentação. Este passo tem como objetivo identificar a região de espuma dentre as regiões isoladas pelo *thresholding*, e identificar a sua borda externa para ser analisada pela próxima etapa. Primeiro, deve-se identificar todas as regiões isoladas pelo *thresholding*. Como a imagem está binarizada, isto é, todos *pixels* possuem a intensidade mínima ou máxima, as regiões isoladas pelo *thresholding* serão os conjuntos de *pixels* contínuos com intensidade máxima. Neste momento a biblioteca OpenCV agiliza a implementação pois é capaz de realizar a identificação das regiões e a detecção de bordas das mesmas de uma só vez. Para isso se utilizou a função *findContours*, que produz um vetor com todos contornos das regiões isoladas pelo *thresholding* através do método de detecção de bordas de Suzuki [14].

Em seguida, calcula-se a área de cada região, a fim de encontrar a região com a maior área. Novamente a biblioteca OpenCV se mostra uma escolha correta para este projeto, pois disponibiliza a função *contourArea* que calcula uma área de um contorno qualquer identificado pela função *findContours*. A função *contourArea* calcula a área através do cálculo do momento de ordem zero (m_{00}) da imagem formada pelo contorno. Seja uma imagem descrita por uma matriz $s(x, y)$ contendo a intensidade de cada ponto (x, y) da imagem, o seu momento de ordem zero pode ser descrito pela equação:



(a) *Timestack* gerado pelo pré-processamento do vídeo de um dia nublado.



(b) Resultado do *threshold* aplicado ao *timestack* acima.

Figura 3.13: *Timestack* e *threshold* do vídeo de um dia nublado.

$$m_{00} = \sum_{x,y} (s(x,y))$$

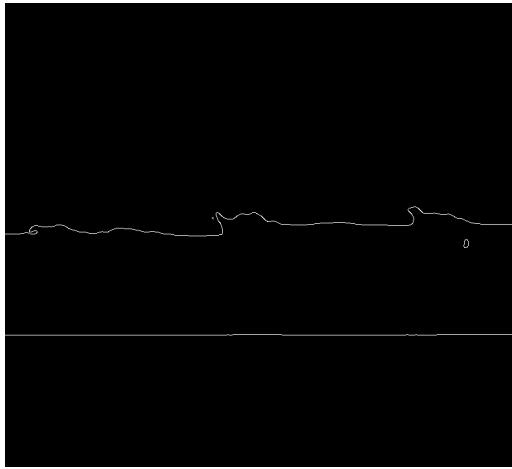
Então pode-se calcular o momento de um contorno ao invés de uma imagem utilizando o Teorema de Green, que relaciona o cálculo da integral de linha de um contorno fechado com o cálculo da integral dupla da superfície limitada por este contorno. Finalmente, para encontrar a maior região basta percorrer o vetor de contornos obtidos anteriormente, calcular a área de cada elemento do vetor e preservar o índice o elemento de maior área (algoritmo 1).

Em condições normais a região de maior área será a região de espuma da imagem, então pode-se descartar todas as outras regiões. Por fim basta identificar as bordas da região restante utilizando um método de detecção de bordas. Novamente, a utilização da biblioteca OpenCV permitiu realizar a detecção de bordas e identificação das regiões isoladas em um único passo, ou seja, não foi necessário realizar mais nenhuma operação para obter as bordas da região de espuma. Outro método considerado para esta tarefa foi o detector de bordas de Canny (figura ??), já utilizado no passo de detecção da linha do horizonte, entretanto pela agilidade fornecida pelo

Algorithm 1 Algoritmo de Busca da Região de Maior Área

```
procedure ENCONTRAMAIORAREA(vetor de contornos)
    indice ← 0
    maiorArea ← 0
    for contornos c no vetor de contornos do
        area ← contourArea(c)           ▷ Função contourArea do OpenCV
        if area > maiorArea then
            maiorArea ← area
            indice ← indicedec
    retorna indice
```

OpenCV o método de Canny foi preterido.



(a) Região de espuma detectada pelo método de Canny.



(b) Região de espuma detectada pelo método de Suzuki (função *findContours*).

Figura 3.14: Comparação dos métodos de Canny e Suzuki para detecção de bordas.

3.13 Análise e Identificação das Ondas Marítimas

A última etapa do algoritmo de estimativa da altura de ondas do mar é analisar a linha de arrebentação extraída do *timestack* e identificar as ondas do mar presentes nesta imagem. Feito isso, é possível encontrar o ponto mínimo e máximo locais da linha de arrebentação, que correspondem ao ponto mínimo e máximo da onda em questão. Estas etapas são descritas nas seções a seguir.

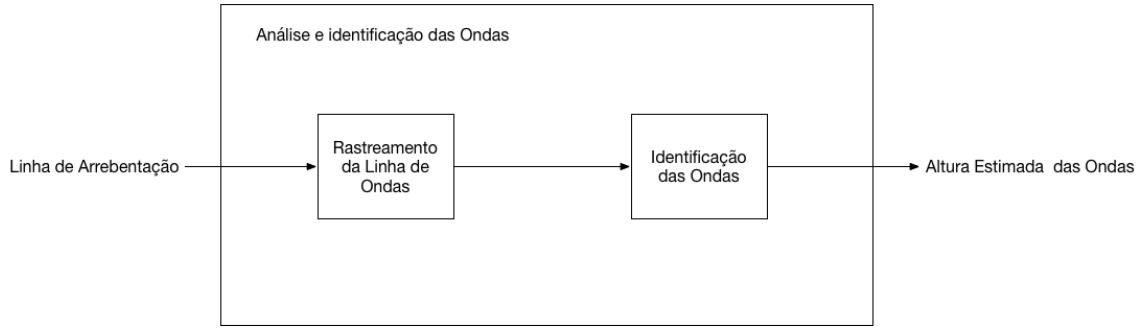


Figura 3.15: Diagrama de Bloco da etapa de análise.

3.14 Rastreamento da Linha de Ondas

O primeiro passo desta etapa consiste em rastrear a linha de arrebentação encontrada, isto é, construir uma sequência $s[n]$ que indica a sequência ordenada dos pontos (x, y) que compõe a linha de arrebentação. Esta sequência ordenada permite caminhar por cima da linha de arrebentação e assim determinar os pontos de máximo e mínimo locais da linha.

Para rastrear a linha de arrebentação foi utilizado um algoritmo que, dado um ponto de origem $s[0] = (x_i, y_i)$, encontra o próximo ponto da linha, o adiciona a sequência e busca o próximo ponto, até que não encontre mais nenhum ponto disponível para adicionar na sequência $s[n]$. O primeiro ponto é encontrado simplesmente percorrendo coluna a coluna da imagem, a partir da coluna mais à esquerda até a coluna mais à direita até que se encontre um ponto de intensidade maior que zero. Então cria-se uma nova sequência com o ponto inicial igual ao ponto encontrado. O pseudo-código 2 ilustra o algoritmo que encontra e popula o primeiro ponto ($s[0]$) de uma sequência $s[n]$.

A busca do próximo ponto ocorre na vizinhança imediata do último ponto da sequência, isto é, todos os oito pontos diretamente conectados ao ponto último ponto da sequência. Para cada ponto da sequência espera-se encontrar outros dois pontos em sua vizinhança, o ponto anterior e o ponto subsequente. Portanto, a análise da vizinhança deve desconsiderar o ponto anterior para não adicionar o mesmo ponto duas vezes na sequência. O ponto restante encontrado na vizinhança será o próximo ponto, então este ponto é adicionado ao final da sequência e o algoritmo se repete,

até que mais nenhum ponto seja encontrado. Entretanto, é possível ocorrer descontinuidades na linha, isto é, um ponto na sequência possuir apenas um ou nenhum outro ponto em sua vizinhança imediata. Nesse caso, nenhum novo ponto será encontrado na vizinhança do último ponto da sequência, e o algoritmo será interrompido prematuramente. Para solucionar este problema se aumenta o raio da vizinhança e o algoritmo é repetido com o mesmo ponto central, buscando pontos mais distantes. Embora esta abordagem resolva os casos de descontinuidades pequenas, abre-se margem para que falsos positivos ocorram, isto é, pontos futuros sejam encontrados pulando o verdadeiro ponto subsequente. Para minimizar este problema, o raio de busca é limitado em um valor máximo igual a três.

Outro problema que atrapalha o funcionamento do algoritmo de rastreamento são casos onde um ponto possui mais de dois pontos em sua vizinhança. Este problema acontece quando existe um "laço" na linha de arrebentação, e costuma ocorrer no momento que a onda quebra, e se torna mais comum a medida que o raio de busca aumenta. Nessa situação, considera-se o próximo ponto simplesmente o primeiro ponto novo encontrado na vizinhança, e os demais pontos encontrados não são considerados. Por último, existem regiões de espuma que o algoritmo não é capaz de rastrear a linha de arrebentação completamente, interrompendo a sequência prematuramente. Nesses casos, simplesmente inicia-se uma nova sequência a partir da coluna a direita do último ponto encontrado. Esse processo pode ser repetido inúmeras vezes até que se atinja a última coluna da imagem. O pseudo-código 3 ilustra o algoritmo de rastreamento que popula os pontos restantes de uma sequência $s[n]$.

Algorithm 2 Algoritmo de Identificação do Primeiro Ponto da Linha

procedure ENCONTRAPRIMEIROPONTO

```

for cada coluna  $c$  na borda da região de espuma do
    for cada elemento  $e$  na coluna  $c$  do
        if intensidade do elemento  $e > 0$  then
             $s[0] =$  posição do elemento  $e$ 
            fim
```

Algorithm 3 Algoritmo de Rastreamento de Linha

```
procedure RASTREIALINHA(raioDeBusca)
    PontoAtual  $\leftarrow s[N]$                                  $\triangleright$  último ponto de  $s[n]$ 
    for pontos p ao redor de PontoAtual em um raio de raioDeBusca do
        if p  $\neq 0$  e p não está em  $s[n]$  then
             $s[N + 1] \leftarrow p$ 
            if algum ponto foi adicionado a  $s[n]$  then
                repetir o procedimento para o último ponto encontrado
            else
                if raioDeBusca  $\leq 3$  then
                    repetir o procedimento com raioDeBusca + 1
                else
                    fim
```

3.15 Identificação das Ondas

O último passo para estimar a altura das ondas em um *timestack* é identificar as ondas na linha rastreada no passo anterior. Para isso, calcula-se a derivada da sequência discreta $s[n]$, da seguinte forma:

$$s'[n] = s[n] - s[n - 1], \text{ para } n > 0$$

A sequência $s'[n]$ indica quais trechos da sequência original $s[n]$ são crescentes, decrescentes ou constantes. Então, ao caminhar sobre a sequência derivada $s'[n]$, é trivial determinar quais são as faces crescentes das ondas. Em um cenário ideal, o ponto inicial de uma face crescente corresponde ao ponto mínimo local de uma onda, e o último ponto crescente desta face corresponde ao ponto de máximo local, determinando assim a altura da onda. Entretanto, em uma imagem não ideal podem existir trechos que a derivada é 0, ou até mesmo negativa, antes da sequência tornar a crescer. A figura 3.16 descreve um autômato que reconhece uma onda considerando as condições descritas anteriormente.

O estado 0 do autômato procura pela primeira derivada $s'[n]$ positiva, marcando o ponto equivalente de $s[n]$ como o mínimo da onda e avançando para o estado 1. O estado 1 procura pela primeira derivada negativa de $s'[n]$, marcando o ponto

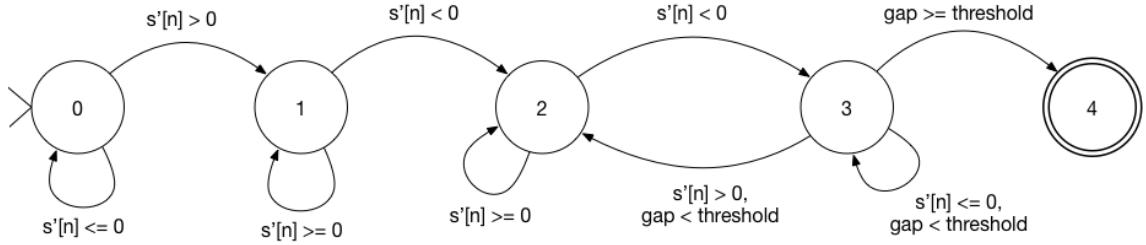


Figura 3.16: Autômato Finito Determinístico que identifica uma onda em uma linha de arrebentação.

equivalente de $s[n]$ como candidato ao ponto máximo da onda e andando para o estado 2. Então, os estados 2 e 3 buscam por "buracos" ou *gaps* no trecho de derivada positiva, onde a derivada se torna negativa por um curto período e depois $s[n]$ torna a crescer. O estado 2 procura por novos pontos de máximo, trechos onde a derivada é positiva, e o estado 3 procura por trechos de derivada negativa. Quando o estado 3 encontrar uma derivada positiva, o autômato retorna para o estado 2. Enquanto o estado 3 encontrar derivadas não-positivas um contador de tamanho do *gap* é incrementado. Quando o tamanho do *gap* é superior a um certo limite, o autômato avança para o estado 4, identificando assim que um candidato a uma onda foi encontrado, delimitado pelos pontos de mínimo e máximo encontrados anteriormente. Este candidato apenas será considerado uma onda se sua altura estiver dentro de uma faixa de controle, isto é, são descartadas ondas muito pequenas ou muito grandes. A figura 3.17 mostra os pares de pontos máximo e mínimo identificados pelo autômato em uma linha de arrebentação aplicados no *timestack* original. As linhas verdes unem os pares de pontos encontrados, e as linhas vermelhas evidenciam a altura em *pixels* da onda identificada.

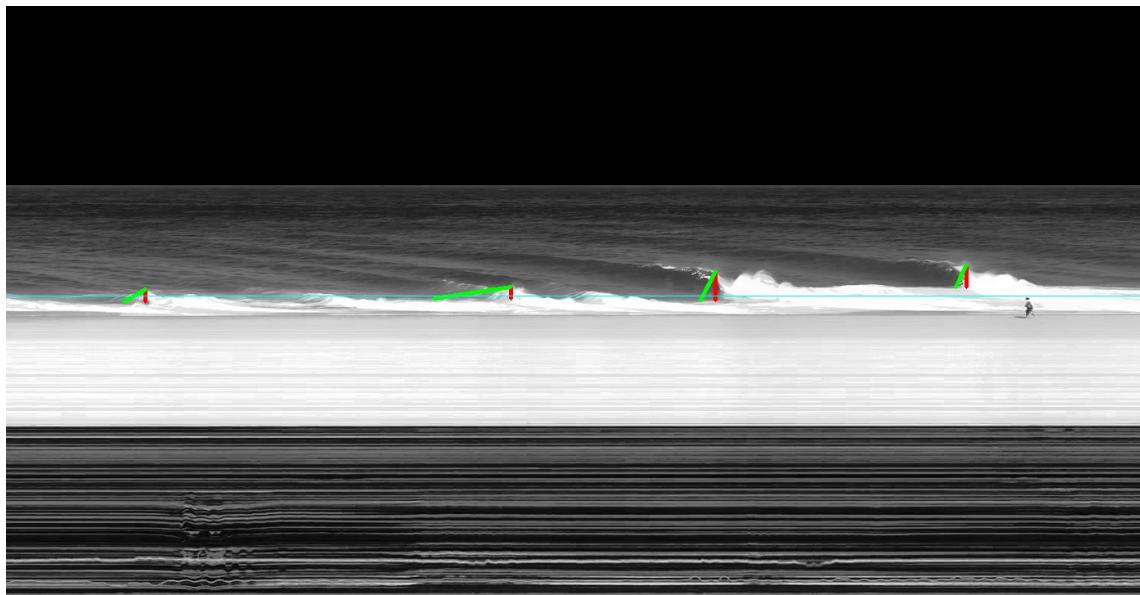


Figura 3.17: Ondas identificadas pelo autômato.

Capítulo 4

Implementação do Algoritmo

Este capítulo apresentará a implementação do algoritmo discutido no Capítulo 3. Como introduzido anteriormente, a biblioteca de processamento gráfico OpenCV foi utilizada para agilizar a implementação do projeto, as próximas seções apresentarão como a biblioteca foi utilizada para implementar cada etapa do projeto.

A biblioteca OpenCV [15] (*Open Source Computer Vision Library*) é uma biblioteca de aprendizado de máquina e visão computacional, distribuída sob a licença BSD, livre para o desenvolvimento de aplicações acadêmicas e comerciais. Ela foi projetada para ser computacionalmente eficiente e tem foco em aplicações em tempo real. Esta biblioteca implementa diversos algoritmos de visão computacional e aprendizado de máquina (mais de 2500 algoritmos implementados [16]), incluindo a grande maioria dos algoritmos descritos nos Capítulos 2 e 3. A biblioteca oferece interface de desenvolvimento nas linguagens de programação C++, Python e Java. Para este projeto de graduação a linguagem de programação escolhida foi a linguagem C++, por dois motivos: das três linguagens é a com melhor performance e é a linguagem que o autor possui maior familiaridade.

O OpenCV apresenta uma estrutura chamada Mat [17] para representar uma matriz qualquer. Esta estrutura será amplamente utilizada no projeto pois é a estrutura que representa uma imagem. As suas propriedades mais importantes são:

1. Número de linhas e número de colunas - determina o comprimento e largura da imagem.

2. Tipo de elemento da matriz - determina se cada elemento da matriz será um char sem sinal ou um *float* / *double*.
3. Número de canais - determina se cada elemento da matriz será representado com um único valor (nível de cinza) ou com três valores (sistema de cores RGB).

Neste projeto a maioria dos Mats utilizados serão do tipo char sem sinal e com apenas um canal. Em determinados momentos as matrizes são convertidas para três canais apenas para facilitar a depuração de algum processo intermediário do projeto, como na figura 3.12 onde é desejável exibir uma imagem com cores para visualizar etapas diferentes do projeto sobre a mesma imagem (nessa figura a região de espuma em vermelho sobre o *timestack* original para verificar se a identificação foi correta).

4.1 Pré-Processamento

O pré-processamento inicia pela leitura de um vídeo no *filesystem*. A leitura do vídeo é realizada pela função *imread*. Esta função aceita um parâmetro: uma *string* contendo o caminho no *filesystem* para encontrar o vídeo desejado. Esta função pode retornar um objeto do tipo Mat (quando o arquivo lido é uma imagem) ou um objeto do tipo VideoCapture (quando o objeto lido é um vídeo). O objeto VideoCapture é uma estrutura de dados que representa um vídeo. Para este projeto as propriedades mais importantes dessa estrutura são o número de *frames* por segundo e a duração do vídeo. O número de *frames* por segundo será utilizado posteriormente para converter a posição no eixo horizontal das ondas encontradas no *timestack* em valores temporais, ou seja, quantos segundos após o início do vídeo as ondas quebraram.

A partir de um objeto VideoCapture é possível obter um objeto Mat que representa cada *frame* do vídeo, através do operador de deslocamento a esquerda. Pode-se iterar por todos os *frames* do vídeo em um laço *while* da seguinte forma:

```
// Lendo vídeo do filesystem
cv::VideoCapture videoCapture = cv::imread("caminho_no_filesystem");
while (true) { // Laço para iterar sobre as frames do vídeo
```

```

Mat frame;
videoCapture >> frame;

// Se frame.data == NULL o laço chegou no final do vídeo
if (frame.data == NULL)
    break;

// frame pronto para realizar alguma operação
}

```

Uma vez obtido cada *frame* do vídeo, pode-se implementar o *pipeline* exibido na figura 3.1b. Primeiro, converte-se o *frame* originalmente colorido para nível de cinza. Isto é realizado pela função *cv::cvtColor*. Esta função aceita três parâmetros: um objeto Mat de entrada, um objeto Mat de saída (onde será guardado o resultado da função) e um valor enumerado representando a representando a operação de conversão que será realizada - *COLOR_BGR2GRAY* para converter uma imagem no sistema RGB em nível de cinza e *COLOR_GRAY2BGR* para converter uma imagem em nível de cinza para sistema RGB. O código abaixo exemplifica o uso desta função ao converter um *frame* colorido para nível de cinza:

```

Mat greyFrame;
cv :: cvtColor (frame ,greyFrame ,COLOR_BGR2GRAY );

```

Após alterar o sistema de cores do *frame*, o próximo passo é equalizar o seu histograma. O uso da biblioteca OpenCV apresenta uma vantagem neste passo pois a função *equalizeHist* calcula o histograma de um objeto Mat de entrada e já o equaliza com apenas uma única chamada de função. Esta função aceita dois parâmetros: o objeto Mat de entrada que será equalizado em um outro objeto Mat onde será guardado o resultado da operação. É importante observar que esta função opera apenas sobre objetos Mat com um canal, isto é, em nível de cinza. O código abaixo exemplifica o uso desta função, equalizando um *frame* em nível de cinza:

```

Mat equalizedFrame ;
cv :: equalizeHist (greyFrame ,equalizedFrame );

```

A detecção da linha do horizonte é realizada sobre o *frame* em nível de cinza não equalizado. Como discutido no capítulo anterior, é necessário realizar primeiro um passo de detecção de bordas com o método de Canny e então detectar a primeira linha transversal na imagem através do algoritmo 3. Primeiro, para detectar as bordas com o algoritmo de Canny utiliza-se a função *Canny* da biblioteca OpenCV. Esta função recebe quatro parâmetros: um objeto Mat de entrada, um objeto Mat de saída, um *double* que representa o valor de *threshold* inferior e um *double* que representa o valor de *threshold* superior. O código abaixo exemplifica o uso desta função:

```
Mat edgeFrame;
double lowThreshold = 50.0;
double highThreshold = 150.0;
cv::Canny(greyFrame, edgeFrame, lowThreshold, highThreshold);
```

Em seguida busca-se pelo primeiro ponto não nulo na primeira coluna da imagem. Em condições normais espera-se que este ponto esteja logo na primeira coluna da imagem, uma vez que a linha de horizonte deve sempre abranger toda o comprimento horizontal da imagem. O código abaixo demonstra como esta busca é implementada:

```
for (int y = 0; y < edgeFrame.rows; y++) {
    uchar value = edgeFrame.at<uchar>(y, 0);
    if (value > 0) { // valor não nulo
        // primeiro ponto encontrado
        break;
    }
}
```

Uma vez encontrado o primeiro ponto, utiliza-se o algoritmo 3 para rastrear o restante da linha até o final do comprimento horizontal da imagem. Para representar as linhas detectadas por este algoritmo foi criado uma classe *Trajectory*. Esta classe cumpre três funções básicas para este projeto:

1. Implementa os algoritmos de rastreamento de uma linha 2 e 3.
2. Mantém um vetor ordenado com os pontos encontrados pelos algoritmos de rastreamento de linha.

3. Calcula a derivada do seu vetor de pontos ordenados.

Para representar cada ponto individual criou-se uma classe interna a classe Trajectory denominada Trajectory::Point. O diagrama 4.1 apresenta interface de cada classe e demonstra a relação entre as duas.

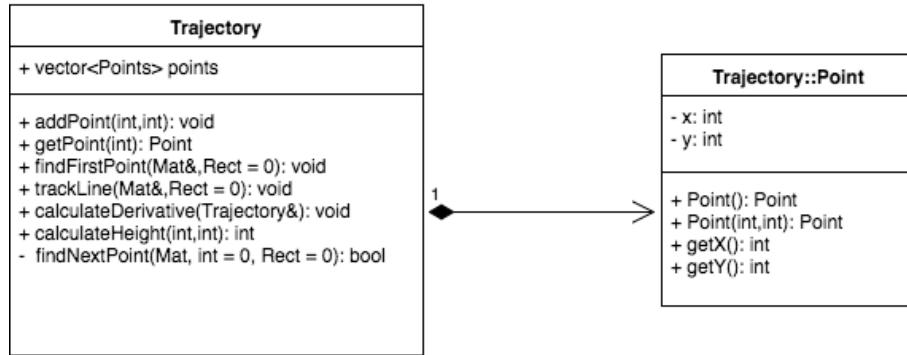


Figura 4.1: Diagrama das classes Trajectory e Trajectory::Point.

O algoritmo 2 é implementado no método Trajectory::findFirstPoint(cv::Mat&,cv::Rect) conforme o código abaixo:

```

void Trajectory :: findFirstPoint ( cv :: Mat& mat , cv :: Rect roi ) {
    int width = mat . cols ;
    int height = mat . rows ;

    for ( int i = (1+roi . x); i < (roi . x + roi . width - 1); i++) {
        Mat col = mat . col ( i );

        bool found = false ;
        for ( int j = (roi . y); j < (roi . y+roi . height - 1); j++) {
            if ( col . at<uchar>(j) > 0 ) {
                found = true ;
                addPoint ( i , j );
                break ;
            }
        }

        if ( found )
            break ;
    }
}

```

```

    }
}

```

O parâmetro mat define qual a imagem onde o método deve procurar pelo primeiro ponto. O método Trajectory::addPoint verifica se este ponto já foi adicionado ao vetor de pontos do objeto Trajectory, e em caso negativo o adiciona ao final do vetor. O parâmetro roi define uma região retangular que é a região de interesse sobre a qual o método operará. O primeiro ponto deve ser procurado dentro da região definida pelo parâmetro roi. Caso o parâmetro roi seja omitido, a região interesse considerada será a imagem inteira.

O algoritmo 3 é implementado nos métodos Trajectory::trackLine(cv::Mat&,cv::Rect) e Trajectory::findNextPoint(cv::Mat&,int,cv::Rect), conforme o código abaixo:

```

void Trajectory :: trackLine( Mat& m, Rect roi ) {
    while(true)
        if (! findNextPoint(m,1 ,roi ))
            break;
}

bool Trajectory :: findNextPoint( Mat& m, int radius , Rect roi ) {
    Trajectory :: Point currentPoint = points.back();

    if (currentPoint.getX() >= (roi.x + roi.width - 1))
        return false;

    int MAX_RADIUS = 3;

    int beginY = currentPoint.getY() - radius > (roi.y) ? currentPoint.getY() - radius : beginY;
    int endY = currentPoint.getY() + radius < (roi.y + roi.height - 1) ? currentPoint.getY() + radius : endY;

    int beginX = currentPoint.getX() - radius > (roi.x) ? currentPoint.getX() - radius : beginX;
    int endX = currentPoint.getX() + radius < (roi.x + roi.width - 1) ? currentPoint.getX() + radius : endX;

    for (int i = beginX; i <= endX; i++)
        for (int j = beginY; j <= endY; j++)
            if (m. at<uchar>(j ,i ) > 0 && addPoint(i ,j ))
                return true;
}

```

```

if ( radius < MAX_RADIUS)
    return findNextPoint(m, radius+1, roi );

return false;
}

```

Novamente o parâmetro mat define qual a imagem onde o método deve procurar pelo próximo ponto, e o parâmetro roi é a região de interesse na imagem que delimita a região onde o método deve procurar pelo ponto. O parâmetro radius indica qual é o raio de busca do algoritmo, isto é, quão distante do último ponto encontrado o método deve procurar por um ponto novo. Como no método Trajectory::findFirstPoint(cv::Mat&,cv::Rect) o parâmetro roi pode ser omitido, assumindo então que a região de interesse é a imagem inteira. O parâmetro radius pode ser omitido também, neste caso assume-se o valor de raio igual a um, que é menor raio possível.

O último passo realizado sobre cada *frame* é a remoção do céu. Este passo é realizado baseado na linha do horizonte rastreada no passo anterior e no *frame* equalizado obtido na etapa de equalização de histograma. Basta então percorrer todos os *pixels* do *frame* equalizado, e alterar o valor de intensidade dos *pixels* acima da linha de horizonte detectada para zero. O código abaixo ilustra como esse passo é realizado:

```

Trajectory trajectory;
Rect roi(0,0,1,edgeFrame.rows);
trajectory.findFirstPoint(edgeFrame,roi);
trajectory.trackLine();
for (int i = 0; i < trajectory.points.size(); i++) {
    Trajectory::Point p = trajectory.getPoint(i);
    for (int j = 0; j < p.getY(); j++) {
        equalizedFrame.at<uchar>(j,i) = 0;
    }
}

```

Onde a variável edgeFrame é um objeto Mat resultante do passo de detecção de linha do horizonte, e a variável equalizedFrame é o objeto Mat resultante do passo de equalização de histograma.

Como este passo é realizado para todos os *frames* do vídeo, é possível otimizá-lo, aproveitando-se do fato que a estabilidade do vídeo é garantida no momento da sua captura. O fato do vídeo ser estável implica que a linha do horizonte sempre estará na mesma posição *frame* a *frame*, então não é necessário detectar uma nova linha do horizonte para cada *frame* do vídeo, basta identificá-la uma única vez e utilizá-la para remover o céu de todos os *frames*. Durante o processamento do primeiro *frame* monta-se um objeto Mat do mesmo tamanho do *frame* que servirá como uma máscara. Então, uma vez detectada a linha do horizonte, altera-se a máscara de forma que todos os *pixels* abaixo da linha do horizonte possuam valor de intensidade máximo e todos os *pixels* acima da linha do horizonte tenham intensidade mínima. O código abaixo mostra como montar a máscara :

```

Trajectory trajectory;
cv::Rect roi(0,0,1,edgeFrame.rows);
trajectory.findFirstPoint(edgeFrame,roi);
trajectory.trackLine();

cv::Mat skyRemoverMask = cv::Mat::ones(equalizedFrame.size(),equalizedFrame.type());

for (int i = 0; i < trajectory.points.size(); i++) {
    Trajectory::Point p = trajectory.getPoint(i);
    for (int j = 0; j < p.getY(); j++) {
        skyRemoverMask.at<uchar>(j,i) = 0;
    }
}

```

Para remover o céu aplica-se a máscara a cada *frame* equalizado através do método Mat::copyTo(Mat,Mat). Este método copia um objeto Mat para outro, sendo que uma máscara pode ser especificada com o segundo parâmetro para definir quais *pixels* do objeto original serão copiados. O código abaixo ilustra como esta função é utilizada em conjunto com a máscara criada anteriormente:

```

cv::Mat skyRemovedFrame;
equalizedFrame.copyTo(skyRemovedFrame,skyRemoverMask);

```

O último passo do pré-processamento é a geração do *timestack* a partir dos *frames* processados. Como discutido no Capítulo 3, um *timestack* é formado pelo acumulo

de uma mesma coluna de cada *frame* de um vídeo. Então, para formar um *timestack* genérico basta iterar por todos os *frames* de um vídeo, selecionar uma coluna com o mesmo índice de cada *frame* e o acumula-lo em um único objeto Mat. O código abaixo exibe como gerar um *timestack* genérico que acumula a coluna central de cada *frame*:

```
cv::VideoCapture videoCapture = cv::imread("caminho_no_filesystem");

// O método VideoCapture::get obtém um propriedade do vídeo
// especificada pelo segundo parâmetro
int frameHeight = videoCapture.get(cv::CV_CAP_PROP_FRAME_HEIGHT);

// O terceiro parâmetro (cv::CV_8UC1) especifica o tipo e número de canais
// do objeto Mat que será criado.
cv::Mat timestack(0, frameHeight, cv::CV_8UC1, 0);

while (true) {
    cv::Mat frame;
    videoCapture >> frame;

    if (frame.data == NULL)
        break;

    // O método Mat::col retorna a coluna do índice especificado
    // como parâmetro.
    cv::Mat middleColumn = frame.col(frame.cols / 2);

    // O método Mat::push_back adiciona parâmetro Mat ao final do
    // objeto Mat que está sendo aplicado.
    timestack.push_back(middleColumn);
}
```

Os passos descritos até aqui compõem juntos o pré-processamento completo, como ilustrado a seguir:

```
// Lendo o vídeo do filesystem
cv::VideoCapture videoCapture = cv::imread("caminho_no_filesystem");

int frameWidth = videoCapture.get(cv::CV_CAP_PROP_FRAME_WIDTH);
```

```

int frameHeight = videoCapture .get (cv ::CV_CAP_PROP_FRAME_HEIGHT);

// Criando objeto Mat que irá guardar o timestack
cv ::Mat timestack (0 ,frameHeight ,cv ::CV_8UC1,0);

// Criando objeto Mat que irá guardar a máscara de remoção do céu
cv ::Mat skyRemoverMask = cv ::Mat ::ones (cv ::Size (frameWidth ,frameHeight ),cv ::CV_8UC1);
bool isMaskReady = false ;

// Iterando sobre os frames do vídeo
while (true) {
    cv ::Mat frame;
    videoCapture >> frame;

    // Verificando se o frame é válido
    if (frame .data == NULL)
        break;

    // Conversão para nível de cinza
    Mat greyFrame;
    cv ::cvtColor (frame ,greyFrame ,COLOR_BGR2GRAY);

    // Se a máscara não estiver pronta (primeira execução),
    // é necessário detectar a linha do horizonte
    if (! isMaskReady) {
        // Detecção da linha do horizonte
        Mat edgeFrame;
        double lowThreshold = 50.0;
        double highThreshold = 150.0;
        cv ::Canny (greyFrame ,edgeFrame ,lowThreshold ,highThreshold);

        // Construção da máscara para remoção da linha do horizonte
        Trajectory trajectory;
        cv ::Rect roi (0 ,0 ,1 ,edgeFrame .rows );
        trajectory .findFirstPoint (edgeFrame ,roi );
        trajectory .trackLine ();

        cv ::Mat skyRemoverMask = cv ::Mat ::ones (equalizedFrame .size () ,equalized

```

```

    for (int i = 0; i < trajectory.points.size(); i++) {
        Trajectory::Point p = trajectory.getPoint(i);
        for (int j = 0; j < p.getY(); j++) {
            skyRemoverMask.at<uchar>(j, i) = 0;
        }
    }

    // Alterando o valor da variável para otimização
    isMaskReady = true;
}

// Equalizando o frame
Mat equalizedFrame;
cv::equalizeHist(greyFrame, equalizedFrame);

// Remoção do Céu
cv::Mat skyRemovedFrame;
equalizedFrame.copyTo(skyRemovedFrame, skyRemoverMask);

// Gerando o timestack
cv::Mat middleColumn = equalizedFrame.col(equalizedFrame.cols / 2);
timestack.push_back(middleColumn);
}

```

4.2 Processamento Principal

O processamento principal, como discutido no capítulo anterior, opera sobre um *timestack* gerado pelo pré-processamento e tem como saída a linha de arrebentação da região de espuma do *timestack*. Também foi discutido que o pré-processamento e o processamento principal podem ocorrer em dispositivos diferentes, o pré-processamento no aparato de *hardware* que realiza a captura do vídeo e o processamento principal em um servidor na nuvem. Dito isso, o primeiro passo do processamento principal é a leitura de um *timestack* no *filesystem*. Assim como no pré-processamento, a leitura é feita pela função *imread*. O código abaixo exemplifica como isto é feito:

```
cv::Mat timestackMat = cv::imread("caminho_para_o_timestack");
```

O passo de suavização é o primeiro passo realizado após a leitura do *timestack* do *filesystem*. Para isso é utilizada a função *GaussianBlur* que aceita quatro parâmetros: um objeto Mat de entrada, um objeto Mat de saída, um objeto Size indicando o tamanho da máscara do filtro gaussiano, um valor double indicando o desvio padrão da máscara do filtro gaussiano. O código abaixo ilustra como esta função é utilizada:

```
cv::Size kernelSize(15,15);
double standardDeviation = 0;
cv::Mat blurredMat;
cv::GaussianBlur(timestackMat, blurredMat, kernelSize, standardDeviation);
```

A seguir é realizada o passo de *thresholding*. Este passo é realizado utilizando a função *threshold*, que recebe cinco parâmetros: um objeto Mat de entrada, um objeto Mat de saída, um valor double indicando o valor limite do *threshold*, um valor double indicando qual é o valor máximo utilizado em *threshold* de binarização, e um valor inteiro indicando qual o tipo de *threshold* que será aplicado. Os tipos de *threshold* disponíveis são:

```
double thresholdLimit = 150;
double maxValue = 255;
cv::Mat thresholdedMat;
cv::threshold(blurredMat, thresholdedMat, thresholdLimit, maxValue, cv::THRESH_BINARY);
```

O último passo do processamento principal é a segmentação e a detecção de bordas. Um pouco da implementação deste passo já foi discutido no Capítulo 3. É utilizada a função *findContours* para identificar as regiões isoladas pelo *thresholding* e detectar as suas bordas, e em seguida a função *contourArea* é utilizada para selecionar dentre as regiões identificadas qual é a região de maior área. Por último, a função *drawContours* é utilizada para formar um objeto Mat com apenas a borda da região de espuma. A função *findContours* recebe cinco parâmetros: um objeto Mat de entrada, um de vetor de vetores de objetos Point indicando os contornos encontrados, um vetor de vetores de inteiros indicando a relação de hierarquia entre os contornos, um inteiro indicando o modo de operação e um inteiro indicando o método de aproximação de contornos. Os modos de operação possíveis são:

```

vector< vector<cv::Point>> contours;
vector<cv::Vec4i> hierarchy;
findContours( thresholdedMat, contours, hierarchy, cv::CV_RETR_CCOMP, cv::CV_CHAIN_APPROX_SIMPLE );

```

O modo de operação organiza os contornos encontrados em dois níveis: contornos externos e contornos internos. Desta forma, é possível eliminar "buracos" dentro da região de espuma que podem surgir na operação de *thresholding*. Em seguida, a área de cada contorno encontrado é calculada e então determina-se a região de maior utilizando a função *contourArea*. Esta função recebe dois parâmetros: um vetor de objetos Point indicando o contorno que será calculado a área, e um valor booleano indicando se a área é orientada ou não. O código abaixo ilustra como esta função é utilizada:

```

vector<cv::Point> contour;
// ...
double area = contourArea(contour, false);

```

Dado um vetor de contornos (vetor de vetores de objetos Point), pode-se determinar o contorno de maior área com o código abaixo:

```

double largest_area = 0;
double largest_contour_index = 0;
for (int i = 0; i < contours.size(); i++) {
    double area = contourArea(contours[i], false);
    if (area > largest_area) {
        largest_area = area;
        largest_contour_index = i;
    }
}

```

Por último é criado um objeto Mat contendo apenas o maior contorno encontrado através da função *drawContours*. Esta função recebe quatro parâmetros: um objeto Mat de saída, um vetor de contornos contendo todos os contornos detectados, um inteiro indicando o índice do contorno que deve ser desenhado no objeto Mat e um objeto Color indicando a cor que deve ser desenhado. O código abaixo ilustra como esta função é utilizada:

```

cv::Scalar color( 255, 255, 255 );
cv::Mat contourMat(thresholdedMat.rows, thresholdedMat.cols, cv::CV_8UC1, cv::Scalar::all(0));
cv::drawContours( contourMat, contours, largest_contour_index, color );

```

O processamento principal pode ser implementado por completo unindo os passos apresentados nesta seção, conforme o código abaixo:

```
// Lendo imagem do filesystem
cv::Mat timestampMat = cv::imread("caminho_para_o_timestamp");

// Suavização
cv::Size kernelSize(15,15);
double standardDeviation = 0;
cv::Mat blurredMat;
cv::GaussianBlur(timestampMat, blurredMat, kernelSize, standardDeviation);

// Threshold
double thresholdLimit = 150;
double maxValue = 255;
cv::Mat thresholdedMat;
cv::threshold(blurredMat, thresholdedMat, thresholdLimit, maxValue, cv::THRESH_BINARY);

// Identificação dos Contornos
vector<vector<cv::Point>> contours;
vector<cv::Vec4i> hierarchy;
findContours(thresholdedMat, contours, hierarchy, cv::CV_RETR_CCOMP, cv::CV_CHAIN_APPROX_SIMPLE);

// Determinação da maior região
double largest_area = 0;
double largest_contour_index = 0;
for (int i = 0; i < contours.size(); i++) {
    double area = contourArea(contours[i], false);
    if (area > largest_area) {
        largest_area = area;
        largest_contour_index = i;
    }
}

// Geração do objeto Mat com apenas o contorno desejado
cv::Scalar color(255, 255, 255);
cv::Mat contourMat(thresholdedMat.rows, thresholdedMat.cols, cv::CV_8UC1, cv::Scalar::all(0));
cv::drawContours(contourMat, contours, largest_contour_index, color);
```

4.3 Análise e Identificação de Ondas Marítimas

A última etapa do algoritmo é a análise e identificação das ondas marítimas. Esta etapa opera sobre a linha de arrebentação resultante do processamento principal e tem como resultado a altura estimada de cada onda identificada.

O primeiro passo a ser realizado é o rastreamento da linha de arrebentação. Neste passo é utilizada a mesma classe de rastreamento de linhas utilizada para detecção da linha do horizonte no pré-processamento, a classe Trajectory. Ao contrário da linha do horizonte, a linha de arrebentação não será uma linha reta, então é possível que o algoritmo de rastreamento implementado na classe Trajectory encontre um trecho da linha que ele não é capaz de rastrear. Nesses casos, interrompe-se o rastreamento até este ponto, as ondas neste trecho são identificadas e então o algoritmo de rastreamento reinicia na coluna seguinte ao ponto em que parou, buscando novamente um ponto inicial e rastreando o restante da linha a partir deste ponto. O rastreamento só termina quando o algoritmo chega até o final da linha, isto é, até que o último ponto da linha rastreada esteja na extremidade direita da imagem analisada. O código abaixo ilustra como o rastreamento da linha de arrebentação é implementado:

```
// Imagem com a linha de arrebentação encontrada na etapa de processamento principal
cv::Mat contourMat = imread("caminho_no_filesystem");

// Variável que guarda qual a última coluna analisada
int lastCol = 0;

while(true) {

    if (lastCol >= contourMat.cols - 1) {
        break;
    }

    Trajectory trajectory(contourMat, Rect(lastCol, 0, contourMat.cols - col, contourMat.rows));

    // Objeto Trajectory pronto para análise

    // Atualizando variável com a coluna seguinte a última coluna do objeto Trajectory
    lastCol = col;
}
```

```

        lastCol = trajectory.points.back().getX() + 1;
    }
}

```

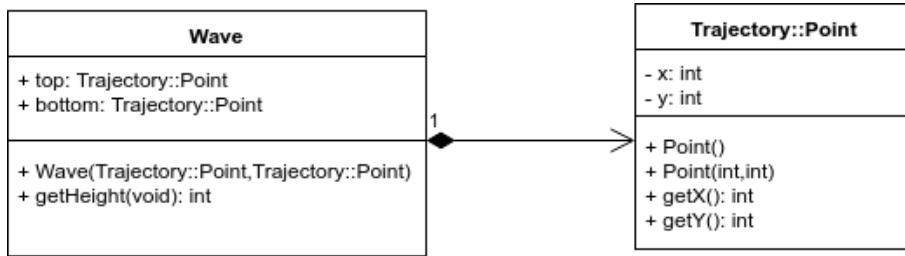


Figura 4.2: Diagrama da classe Wave.

O último passo a ser realizado é a identificação das ondas marítimas. Este passo é realizado implementando o automato descrito no capítulo anterior (Figura 3.16). O automato de identificação das ondas foi implementado em uma função `analyseTrajectory`, conforme o código abaixo:

```

void analyseTrajectory (Trajectory& trajectory) {
    vector<Trajectory :: Point> derivative;

    trajectory.calculateDerivative(derivative);

    int state = 0;
    int bottom_index = 0;
    int top_index = 0;
    int gap_count = 0;

    int GAP_THRESHOLD = 20;

    // Vetor que guarda o resultado com as ondas identificadas
    vector<Wave> detectedWaves;

    for (int i = 1; i < derivative.size(); i++) {
        int dX = derivative[i].getX();
        int dY = derivative[i].getY();

        switch(state) {
            case 0:
                if (dY < 0) {

```

```

        state = 1;
        bottom_index = i+1;
    }

    break;

case 1:
    if (dY > 0) {
        top_index = i+1;
        state = 2;
    }

    break;

case 2:
    if (dY > 0) {

        state = 3;
        gap_count = 0;

    } else {
        if (trajectory.calculateHeight(bottom_index , t)
            top_index = i+1;
    }

    break;

case 3:
    if (gap_count >= GAP_THRESHOLD) {

        detectWave(trajectory , bottom_index , top_index , c);

        gap_count = 0;
        state = 0;
        bottom_index = 0;
        top_index = 0;

    } else {

        if (dY < 0) {
            state = 2;
        }
    }
}

```

```

                if ( trajectory . calculateHeight ( bottom
                                              top_index = i + 1;

                } else {
                    gap_count++;

                }
            }

            break;
        }

        if ( state > 0 && i == ( derivative . size () - 1) ) {
            detectWave ( trajectory , bottom_index , top_index , detectedWaves );

            state = 0;
            gap_count = 0;
            bottom_index = 0;
            top_index = 0;
        }

    }
}

```

A função `detectWave` utilizada na função `analyseTrajectory` verifica se um par de pontos identificados como pontos mínimos e máximos locais são elegíveis para formar uma onda. Para serem elegíveis, cada par de pontos devem atender 2 requisitos:

1. A diferença de altura do par de pontos deve estar dentro de uma faixa limite, para impedir que anomalias sejam identificadas como ondas. Os valores limite máximo e mínimo são, respectivamente: 10 e 200.
2. O ponto mínimo detectado deve estar à direita do último ponto máximo detectado. Isto se deve pois o eixo horizontal da linha de arrebentação representa o eixo temporal, e sempre deve evoluir da esquerda para a direita. Portanto novas ondas devem sempre estar mais à direita de ondas já identificadas. Esta verificação é útil para evitar que a turbulência causada pela arrebentação da

onda gere falsos-positivos, ou seja uma onda seja erroneamente detectada em decorrência da arrebentação de outra onda.

O código abaixo exibe como a função detectWave é implementada:

```
void detectWave( Trajectory& trajectory , int bottomIndex , int topIndex , vector<Wave>& detectedWaves )
{
    int MIN_HEIGHT_THRESHOLD = 10;
    int MAX_HEIGHT_THRESHOLD = 200;
    int height = trajectory . calculateHeight (bottomIndex , topIndex );

    if (( height > MIN_HEIGHT_THRESHOLD && height < MAX_HEIGHT_THRESHOLD ))
        if (detectedWaves . size () == 0 || trajectory . getPoint (bottomIndex) . getY () > height )
            detectedWaves . push_back (Wave( trajectory . points [bottomIndex] , trajectory . points [topIndex] ));

}
```

O resultado deste automato é um vetor de objetos Wave (figura 4.2), um objeto criado pelo autor que representa um par de pontos indicando o ponto mais baixo e o ponto mais alto de uma onda. A classe Wave implementa dois métodos: o método getHeight, que calcula a diferença de altura da onda em *pixels*, e o método getHalfway, que calcula o ponto central horizontal da onda.

Por fim, a altura das ondas em *pixels* é convertida para medidas do mundo real, em metros. Esta conversão é realizada baseado no método apresentado na seção 2.5. O código abaixo ilustra como as equações descritas na seção 2.5 foram implementadas:

```
// Função que calcula o angulo em radianos correspondente de um pixel na imagem.
double calculateAngle(int pixel , double cameraAngle , double focalAngle , int imageHeight )
{
    double ang = cameraAngle - focalAngle/2 + ( (pixel * focalAngle) / imageHeight );
    return ang * 3.14159265 / 180;
}

// Função que calcula a altura real em metros entre dois pixels
double calculateRealHeight(int pixelTop , int pixelBottom , double cameraHeight , double focalAngle )
{
    double angleBottom = calculateAngle(pixelBottom , cameraAngle , focalAngle , imageHeight );
    double angleTop = calculateAngle(pixelTop , cameraAngle , focalAngle , imageHeight );
    return cameraHeight * ( 1 - ( tan(angleTop) / tan(angleBottom) ) );
}
```

A altura das ondas identificadas na linha de arrebentação pode ser calculada conforme o código abaixo:

```
double cameraAngle, focalAngle, cameraHeight;  
int imageHeight = contourMat.rows;  
  
for (int i = 0; i < detectedWaves.size(); i++) {  
    double realWorldHeight = calculateRealHeight(detectedWaves[i].top.getY(), decte  
}  
}
```

As variáveis cameraAngle, focalAngle e cameraHeight são parâmetros conhecidos da filmagem, medidos no momento da captura do vídeo.

Capítulo 5

Dados Experimentais

Este capítulo tem como objetivo apresentar os resultados obtidos ao aplicar o método de estimativa de altura de ondas discutido nos capítulos 2 e 3.

Os vídeos analisados foram capturados na praia de Itacoatiara, em Niterói. A câmera foi posicionada no Quiosque 5, na orla da praia, a uma altura de aproximadamente oito metros do nível do mar. Este quiosque foi escolhido para posicionar a câmera pois uma visão clara do mar, sem a obstrução da vegetação local da praia. A câmera está posicionada com uma angulação vertical de 83° . O campo visual (*field of view*) da câmera do *smartphone* foi calculado através da seguinte equação [18]:

$$fov = 2 * \text{atan}\left(\frac{h}{2f}\right)$$

onde h é a altura da imagem no sensor da câmera e f é a distância focal da câmera. Estes parâmetros foram obtidos no momento da captura através do próprio *smartphone*. Os valores considerados foram:

Parâmetros	Valores
Distância Focal	4.15 mm
Altura do sensor	3.60 mm
Campo de visão vertical	46.92°

Foram analisados

adicionar número de vídeos

vídeos. Para cada vídeo as ondas foram analisadas automaticamente pelo algoritmo e pelo manualmente pelo autor, a fim de verificar quantas ondas foram ou deixaram de ser identificadas, e verificar a precisão da identificação do ponto mínimo e máximo da onda. Estes são os principais erros observados nos dados analisados.



(a) Onda detectada automaticamente.



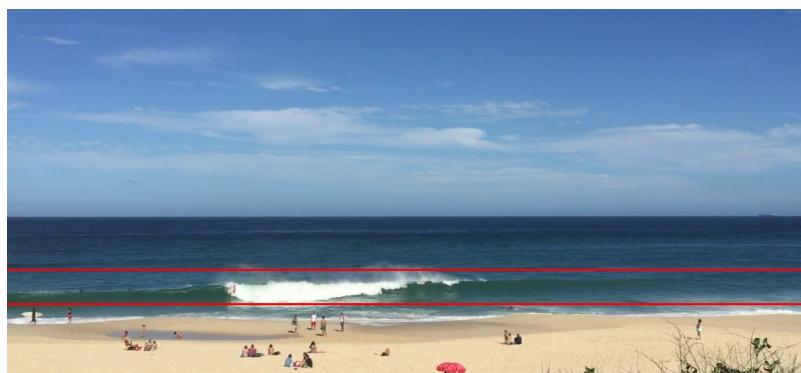
(b) Onda detectada manualmente.

Figura 5.1: Comparativo entre onda detectada manualmente e onda detectada automaticamente com interferência de pessoas na areia.

A identificação dos pontos mínimos e máximos está suscetível a interferência externa, como banhistas ou pessoas próximas da zona de arrebentação (Figura 5.1). A ação do vento [19] também pode introduzir erros na identificação do ponto de máximo, causando um *spray* da espuma do mar que pode ser erroneamente detectado como o ponto máximo da onda (Figura 5.2).



(a) Onda detectada automaticamente.



(b) Onda detectada manualmente.

Figura 5.2: Comparativo entre onda detectada manualmente e onda detectada automaticamente com efeito de *spray* de espuma.

Capítulo 6

Conclusões

Referências Bibliográficas

- [1] BROWNE, M., BLUMENSTEIN, M., TOMLINSON, R., *et al.*, “An intelligent system for remote monitoring and prediction of beach conditions”. In: *Proceedings of the International Conference on Artificial Intelligence and Applications*, pp. 533–537, Innsbruck, 2005.
- [2] HOLLAND, K. T., HOLMAN, R. A., LIPPmann, T. C., *et al.*, “Practical Use of Video Imagery in Nearshore Oceanographic Field Studies”, *IEEE Journal of Oceanic Engineering*, pp. 81–92, 1997.
- [3] LANE, C., GAL, Y., BROWNE, M., *et al.*, “A new system for breakzone location and the measurement of breaking wave heights and periods.” In: *Geoscience and Remote Sensing Symposium (IGARSS), 2010 IEEE International*, pp. 2234–2236, Honolulu, 2010.
- [4] LANE, C., GAL, Y., BROWNE, M., “Automatic Estimation of Nearshore Wave Height from Video Timestacks”. In: *Digital Image Computing Techniques and Applications (DICTA), 2011 International Conference on*, pp. 364–369, Noosa, 2011.
- [5] LANE, C., GAL, Y., BROWNE, M., “Long-Term Automated Monitoring of Nearshore Wave Height From Digital Video”, *IEEE Transactions on Geoscience and Remote Sensing*, v. 52, pp. 3412–3420, 2014.
- [6] SOUZA, R. D., “<http://ricosurf.com.br/boletim-das-ondas/zona-oeste-rj/prainha-grumari/>”, <http://ricosurf.com.br/>, 2017, (Acesso em 06 Junho 2017).
- [7] GONZALEZ R., WOODS, E., *Digital Image Processing, 2nd Edition*. New Jersey, Prentice Hall, 1992.

- [8] JÄHNE, B., *Digital Image Processing*. Berlin, Springer-Verlag, 2002.
- [9] CANNY, J., “A Computational Approach to Edge Detection”, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, v. PAMI-8, pp. 679–698, 1986.
- [10] JUNEJA, M., SANDHU, P. S., “Performance Evaluation of Edge Detection Techniques for Images in Spatial Domain”, *International Journal of Computer Theory and Engineering*, v. 1, pp. 614–621, 2009.
- [11] FUSIELLO, A., “Elements of Geometric Computer Vision”, http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/FUSIELLO4/tutorial.html, 2017, (Acesso em 10 Junho 2017).
- [12] FOUNDATION, R. P., “Camera Module”, <https://www.raspberrypi.org/documentation/hardware/camera/>, 2017, (Acesso em 15 Julho 2017).
- [13] HO, N., “SIMPLE VIDEO STABILIZATION USING OPENCV”, <http://nghiaho.com/?p=2093>, 2014, (Acesso em 04 Abril 2017).
- [14] SUZUKI, S., ABE, K., “Topological structural analysis of digitized binary images by border following.”, *Computer Vision, Graphics, and Image Processing*, v. 30, n. 1, pp. 32–46, 1985.
- [15] OPENCV, “OpenCV Home Page”, <http://opencv.org/>, 2017, (Acesso em 20 Julho 2017).
- [16] OPENCV, “OpenCV About”, <http://opencv.org/about.html>, 2017, (Acesso em 20 Julho 2017).
- [17] OPENCV, “Mat Class Reference”, http://docs.opencv.org/3.1.0/d3/d63/classcv_1_1Mat.html, 2017, (Acesso em 20 Julho 2017).
- [18] BOURKE, P., “Field of View and Focal Length”, <http://paulbourke.net/miscellaneous/lens/>, 2013, (Acesso em 21 Julho 2017).

- [19] P. A. HWANG, I. B. S. E. M. D. A., “Breaking waves and near-surface sea spray aerosol dependence on changing winds: Wave breaking efficiency and bubble-related air-sea interaction processes”. In: *IOP Conf. Series: Earth and Environmental Science 35*, Seattle, 2016.

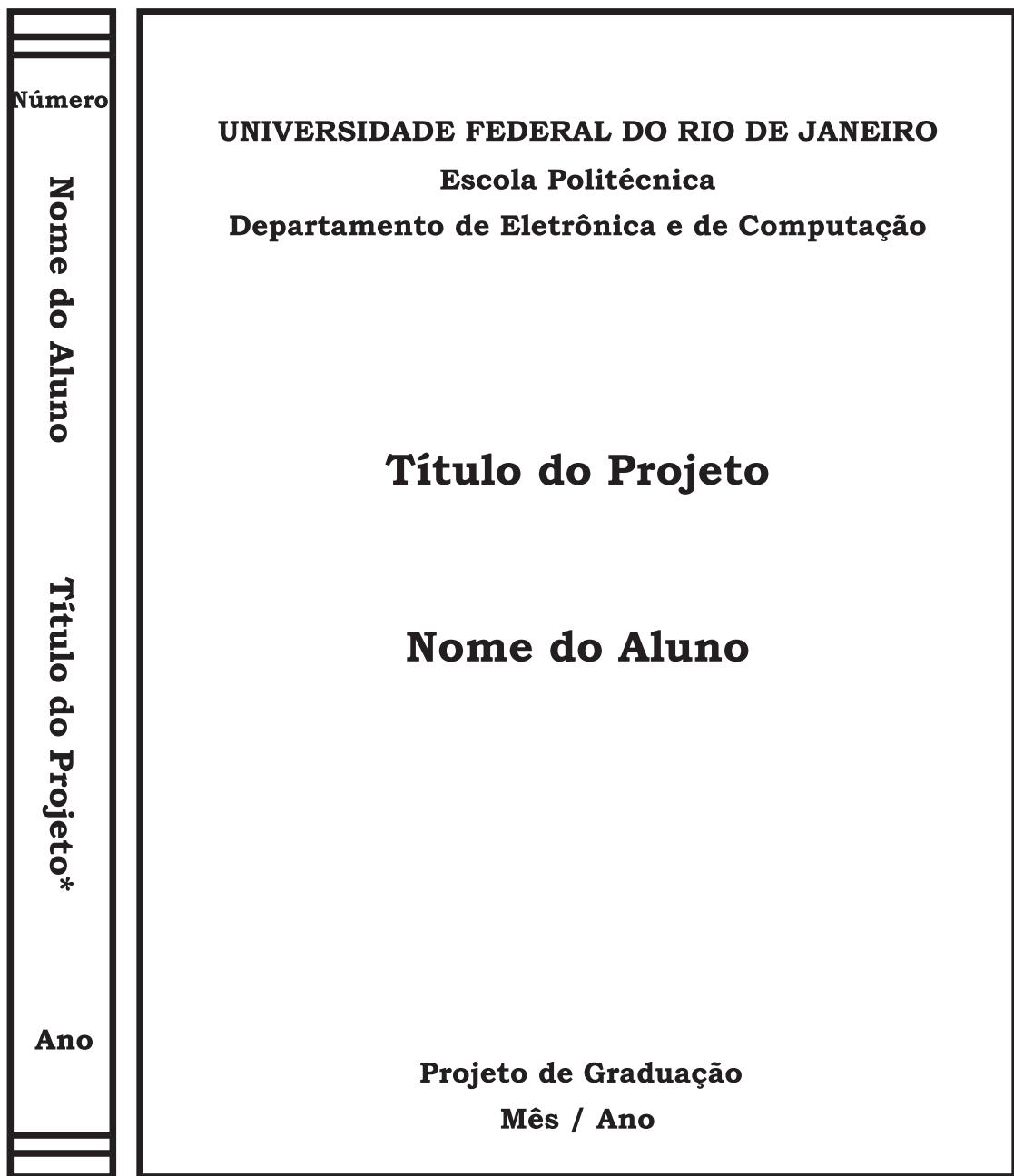
Apêndice A

O que é um apêndice

Elemento que consiste em um texto ou documento elaborado pelo autor, com o intuito de complementar sua argumentação, sem prejuízo do trabalho. São identificados por letras maiúsculas consecutivas e pelos respectivos títulos.

Apêndice B

Encadernação do Projeto de Graduação



* Título resumido caso necessário
Capa na cor preta, inscrições em dourado

Figura B.1: Encadernação do projeto de graduação.

Apêndice C

O que é um anexo

Documentação não elaborada pelo autor, ou elaborada pelo autor mas constituindo parte de outro projeto.