Universidade Federal do Rio de Janeiro

Escola Politécnica

Departamento de Eletrônica e de Computação

Estudo da Biblioteca OpenCV

Autor:

Daniel Ponciano dos Santos Barboza

Orientador:

Prof. Mariane Rembold Petraglia

Examinador:

Prof. Antonio Petraglia

Examinador:

Prof. José Gabriel Rodriguez Carneiro Gomes

DEL

Julho de 2009

DEDICATÓRIA

Dedico este trabalho à minha família pelo suporte, compreensão e valores que me guiaram durante a jornada do curso de graduação à conclusão deste projeto. Em especial, agradeço à minha mãe, Lucia Marina Ponciano dos Santos Barboza por sempre priorizar minha formação e sem a qual não estaria aqui – não pelo motivo natural, mas por me conduzir até onde cheguei; à minha amiga e namorada, Mariana Dias Costa, por estar ao meu lado desde o momento que me transfere do IME até concluir meu último trabalho na UFRJ. Participou ativamente de cada momento acadêmico sempre compreensiva e decisiva em cada passo importante. E finalmente, à minha querida avó Rita Ponciano Estrada, que é mais uma mãe e um pai. Ao lado do meu avô Neil Garcia Estrada, fez todo o possível para que seguisse meus estudos, sendo um apoio, e uma inspiração.

AGRADECIMENTO

Agradeço ao povo brasileiro por investir em minha formação até hoje. Aos mestres que me transmitiram conhecimento e experiência. Mesmo fora de uma sala de aula, ensinaram e contribuíram para minha formação. Agradeço à minha orientadora, Prof. Mariane Petraglia por sua compreensão e apoio durante o desenvolvimento deste trabalho, neste momento em que deixamos a faculdade e ingressamos em novos desafios. Aos meus colegas que contribuíram com experiências, ensinamentos, pensamentos para que essa jornada se tornasse mais suave, sem os quais não teria sido tão prazerosa, enriquecedora e proveitosa. Agradeço ao meu coordenador e paraninfo, Prof. Carlos José Ribas D'Ávila, por estar sempre pronto a nos auxiliar, tangenciando o limite da regulamentação ao seu alcance em benefício dos alunos.

RESUMO

Neste trabalho apresentamos um estudo sobre a biblioteca OpenCV, visando o desenvolvimento de algoritmos para aumentar a detecção de bordas utilizando a linguagem C ou C++. O estudo é composto de uma análise de funções disponíveis, detalhando seu uso, aplicações, e códigos. Um algoritmo simples para a detecção de bordas de imagens submarinas é apresentado, combinando as funções estudadas.

Palavras-Chave: OpenCV, tutorial, C/C++, processamento de imagens, detecção de bordas.

ABSTRACT

This work presents a study on the OpenCV library, for the development of algorithms in order to improve the edge detection in images using the language C or C ++. The study consists of an analysis of functions, tailing its use, applications, and codes. A simple algorithm for detecting edges of submarine images is presented, combining the studied functions.

Key-words: OpenCV, tutorial, C / C ++, image processing, edge detection.

SIGLAS

UFRJ – Universidade Federal do Rio de Janeiro

OpenCV – Open Computer Vision Library

IME – Instituto Militar de Engenharia

COPPE – Instituto Alberto Luiz Coimbra de Pós-Graduação e Pesquisa em Engenharia - Coordenação dos Programas de Pós Graduação em Engenharia

CMU – Carnegie Mellon University

LABIEM – Laboratório de Imagem e Instrumentação Eletrônica

UTFPR – Universidade Técnica Federal do Paraná

TecGraf – Grupo de Tecnologia em Computação Gráfica

PUC-RJ – Pontifícia Universidade Católica do Rio de Janeiro

MIT – Massachusetts Institute of Technology

GUI – Graphical User Interface

Sumário

Capítulo 1

Introdução

1.1 – Tema

O processamento de imagens tem aumentado seu campo de atuação e, consequentemente, novas ferramentas são criadas visando atender a propósitos específicos. Trata-se da captura, tratamento e melhoramento da imagem visando aplicações científicas correlatas como: segmentação da informação, isolando parte da imagem; parametrização, determinando grandezas; e reconhecimento, identificando semelhanças entre objetos. Esta última se abre em várias partes: captura, tratamento, aprendizado de máquinas, e inteligência artificial, buscando alcançar os parâmetros necessários para que se associe o reconhecimento feito pelo cérebro humano a um problema tangível à programação.

Dentro de processamento de imagens, a detecção de bordas é um campo em constante crescimento. O estudo aborda, portanto, este tema e suas aplicações através de uma ferramenta de processamento de imagens que está em crescente desenvolvimento em grandes centros científicos.

A ferramenta é uma biblioteca multiplataforma para desenvolvimento de *softwares*. Dentre as linguagens destaca-se o C por sua propriedade poder ser executada em *hardwares* embarcados.

1.2 – Delimitação

Academicamente, uma das ferramentas mais utilizadas para o desenvolvimento de técnicas de processamento de imagens é o MATLAB®. E assim como ocorre com todas as ferramentas, novas aplicações surgem para fins mais específicos. O objeto de estudo deste projeto é uma ferramenta relativamente recente, que permite o processamento de imagens – o OpenCV®.

A biblioteca OpenCV®, desenvolvida pela Intel em 2000, permite manipulação de dados de imagens, manipulação de matrizes e vetores, desenvolvimento de rotinas de álgebra linear, estruturas de dados dinâmicas, desenvolvimento de algoritmos de processamento de imagem, análise estrutural, calibração de câmera, análise de movimento (*tracking*), reconhecimento de objetos, GUI Básica, e rotulagem de imagem. Sua vantagem em relação ao MATLAB® é o fato de poder ser usada para programar em várias plataformas, como C/C++, Python, Visual Basic, Ruby, permitindo uma integração mais fácil com outros programas, evitando problemas de integração e facilitando no caso de desenvolvimento de *softwares* embarcados.

Este estudo visa analisar algumas funções aplicáveis à detecção de bordas e ponderar a viabilidade do uso da mesma em algoritmos de detecção de bordas em *softwares* embarcados.

1.3 – Justificativa

Este estudo partiu de um projeto que visa desenvolver um sistema de visão estéreo para um veículo submarino controlado remotamente para inspeção de instalações submarinas. Além de controle e robótica, o processamento de imagens está ligado diretamente à necessidade de orientação do robô e captura de imagens das instalações. Este projeto encontra-se em vários níveis acadêmicos, envolvendo de estudantes de graduação a professores da COPPE. Atualmente, os algoritmos são desenvolvidos em MATLAB e em uma próxima fase serão desenvolvidos em C.

Devido à necessidade de embarcar o *software*, deu-se a proposta de um estudo de algumas funcionalidades da biblioteca em C de desenvolvimento de imagens. A partir de um algoritmo simples de detecção de bordas em MATLAB, foi estruturado o estudo de funções que poderiam ser implementadas substituindo-as por funções em C utilizando a biblioteca OpenCV.

1.4 – Objetivos

O objetivo do estudo é em um primeiro momento analisar as vantagens, aplicações, incompatibilidades e limitações da biblioteca, e em um segundo momento criar códigos comparativos mostrando o desempenho de funções. Finalmente, apresentar um algoritmo em C, análogo ao proposto no primeiro momento. Além destas

técnicas, serão apresentadas funcionalidades potencialmente interessantes para detecção de bordas de dutos.

1.5 – Metodologia

O estudo será realizado a partir de uma análise das funções da linguagem OpenCV descrevendo aquelas necessárias a reprodução do algoritmo proposto por um aluno de mestrado envolvido no projeto da COPPE. Para análise, será executada individualmente cada função com um resumo sobre suas funcionalidades, aplicações, parâmetros e detalhes práticos observados durante a sua execução.

Será apresentado a seguir o *software* pronto, criado a partir do algoritmo proposto. Os pontos principais do código serão comentados.

Finalmente, cinco imagens submarinas serão utilizadas como referência para qualificar o desempenho do *software*.

Capítulo 2

Estudo da Biblioteca

2.1 – Estrutura do Estudo

Primeiramente, será analisada cada função, e para executá-la, utilizar-se-ão duas imagens (Figs. 1 e 2) para padronizar a métrica, visto que sua avaliação é qualitativa. Para cada função haverá um resumo sobre suas funcionalidades, aplicações, parâmetros e detalhes práticos observados durante a execução.

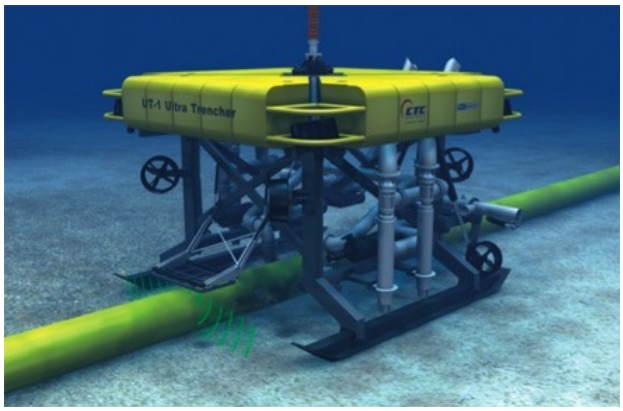


Fig. 1 – Imagem 1.

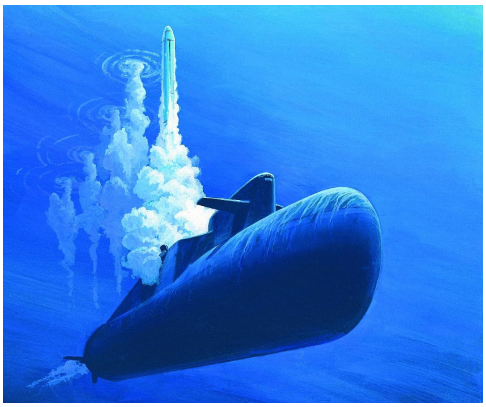


Fig. 2 – Imagem 2.

O desempenho do programa será analisado através de cinco imagens submarinas e sua rotina será descrita através dos comentários do código.

2.2 – Sobel

A função abaixo calcula a primeira, segunda ou terceira derivada usando o filtro de Sobel.

```
void cvSobel( const CvArr* src, CvArr* dst, int xorder, int yorder,
int aperture_size );
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem destino.

`xorder`

Ordem da derivada ao longo da direção X. Normalmente vai usar 0, quando não houver derivada de x, e 1, no máximo 2.

`yorder`

Ordem da derivada ao longo da direção Y. O uso é análogo ao de *xorder*.

`aperture_size`

O tamanho do Kernel de Sobel deve ser 1, 3, 5 ou 7. Em todos os casos, o kernel de tamanho *aperture_size* x *aperture_size* é utilizado separadamente para calcular a derivada. Quando o *aperture_size* for 1, é utilizado um kernel 3x1 ou 1x3 para x e y.

O operador Sobel é uma das mais importantes convoluções. Trata-se de uma aproximação da derivação. A função realiza a diferenciação parcial de x e y, ou seja:

$$dst(x,y) = \partial^{xorder+yoder} src / \partial x^{xorder} \bullet \partial y^{yorder} |_{(x,y)}$$

A função *cvSobel* calcula a derivada (aproximadamente) pela convolução da imagem com Kernel. Trata-se de uma aproximação polinomial, pois o operador é definido em um espaço discreto. O operador funciona da seguinte forma:

$$G = K * A;$$

onde G é a aproximação da derivada; K é o Kernel - uma matriz de tamanho *aperture_size* por *aperture_size*; e A é a imagem fonte.

Caso a entrada da função fosse *cvSobel (const CvArr* src, CvArr* dst, 1, 0, 3),* o *kernel* seria*:*

$$\begin{vmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{vmatrix}.$$

E este seria

$$\begin{vmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{vmatrix},$$

ou

$$\begin{vmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{vmatrix},$$

caso a entrada fosse *cvSobel(const CvArr* src, CvArr* dst, 0, 1, 3),* dependendo da origem da imagem (campo *origin* da *IplImage* structure).

A função Sobel é associada à suavização gaussiana para aumentar mais ou menos a resiliência ao ruído.

Não ocorre nenhuma normalização da imagem durante a execução da função, implicando maiores valores absolutos na imagem destino do que na imagem fonte.

Segundo Gary Bradski e Adrian Kaehlerno, no livro Learning OpenCV *"[...]IF src is 8-bit then the dst must be of depth IPL_DEPTH_16S to avoid overflow. [...]"*. Entretanto, durante esse estudo utilizando sobel, nenhum caso de *overflow* ocorreu com diferentes tipos de imagens utilizadas em testes da função.

Para imagens escuras, como as analisadas a seguir, podem-se utilizar imagens do mesmo tipo (8-bits). A função também permite utilizar a mesma imagem como *imagem fonte* e *imagem destino* (nota-se que imagem original é perdida no processo). Vale ressaltar ainda, que a imagem fonte deve ser de 8-bits ou 32-bits de ponto flutuante, e as imagens destino devem possuir o mesmo formato e apenas um canal; o resultado pode ser convertido de volta para 8-bits usando as funções *cvConvertScale* ou *cvConvertScaleAbs*.

Originalmente as imagens são arquivos do tipo CvArr, entretanto a função aceita arquivos do tipo IplImage.

Filtro Scharr

O tamanho do kernel Sobel deve ser 1, 3, 5, ou 7. Entretanto, há uma implementação especial quando o aperture_size é **-1** (ou CV_SCHARR) - o Filtro Scharr.

Trata-se de um filtro que usa uma matriz 3x3 específica, dada por:

$$\begin{vmatrix} -3 & 0 & 3 \\ -10 & 0 & 10 \\ -3 & 0 & 3 \end{vmatrix}$$

para derivada x ou sua transposta para derivada y.

A função Sobel tem a propriedade de utilizar diversos tamanhos de kernel de forma rápida e iterativa. Uma das maiores vantagens de biblioteca OpenCV é seu uso em tempo real.

Uma desvantagem da função é o desempenho da aproximação para pequenos kernels.

A seguir, nas Figs. 3 a 15, observaremos o comportamento das imagens das Figs. 1 e 2 processadas por cvSobel. Estas estão ordenadas em linhas pelo tamanho do aperture size, visando à comparação dos parâmetros X e Y em cada coluna. Primeiramente, serão exibidos os resultados da imagem da Fig. 2 e posteriormente da imagem da Fig. 1, seguindo os mesmos critérios.

Em kernels maiores, em que são utilizados vários pontos na aproximação, este problema torna-se menos significativo. O kernel maior é aplicado em um número maior de pixels. O tamanho do kernel é correlacionado à sua relisiência ao ruído, sendo que os maiores oferecem uma aproximação melhor da derivada, como pode ser observado nas imagens das Figs. 5, 8 e 11.

Em aproximações unidirecionais, ou seja, filtros X ou Y construídos com a função *cvSobel,* há alinhamento perfeito com os eixos x ou y. Os erros de precisão surgem com filtros bidirecionais. Aproximações de derivadas bidirecionais exigem medições angulares e a dificuldade surge ao fazer medições com precisão. As medidas inexatas do ângulo de inclinação diminuirão o desempenho do reconhecimento do classificador. Utilizando um Kernel 3x3, estas imprecisões são mais aparentes para ângulos maiores, a partir da horizontal ou da vertical. O desempenho das aproximações unidirecionais e bidirecionais de derivadas pode ser analisado nas imagens das Figs. 6, 7 e 8.

A maior imprecisão da função ocorre com filtros de Sobel 3x3 bidirecionais. Entretanto, são os mais rápidos.

O Filtro Scharr é tão rápido quanto o de Sobel 3x3, mas oferece melhor precisão, como pode ser visto nas imagens das Figs. 4 e 12 ou nas imagens das Figs. 13 e 14.

Este filtro é recomendado nos casos em que se queira utilizar um processamento mais rápido através da redução do tamanho do Kernel para 3x3.

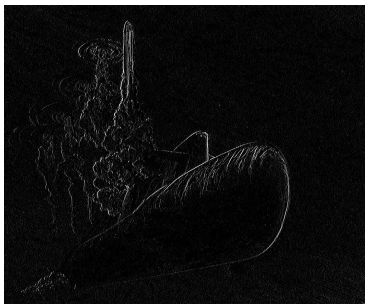


Fig.3 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=0 e aperture size = 3.

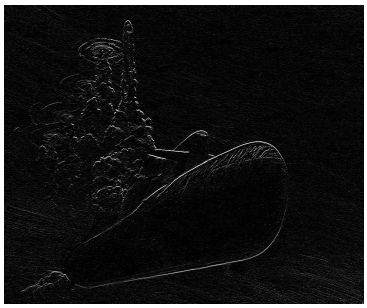


Fig.4 – Resultado da imagem 2 processada por cvSobel com parâmetros X=0, Y=1 e aperture size = 3.

Fig.5 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=1 e aperture size = 3.

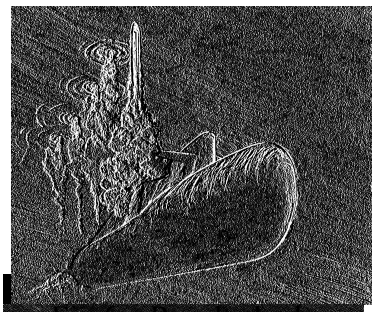


Fig.6 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=0 e aperture size = 5.

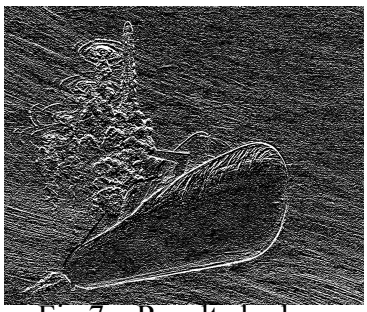


Fig.7 – Resultado da imagem 2 processada por cvSobel com parâmetros X=0, Y=1 e aperture size = 5.

Fig.8 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=1 e aperture size = 5.

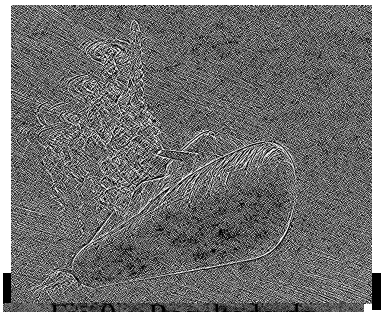


Fig.9 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=1 e aperture size = 7.

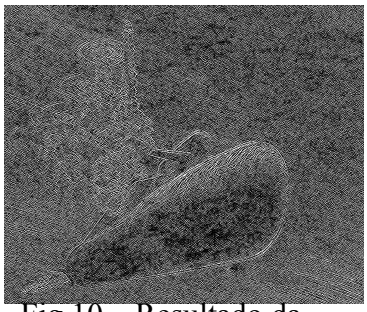


Fig.10 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=2 e aperture size = 7.

Fig.11 – Resultado da imagem 2 processada por cvSobel com parâmetros X=1, Y=3 e aperture size = 7.

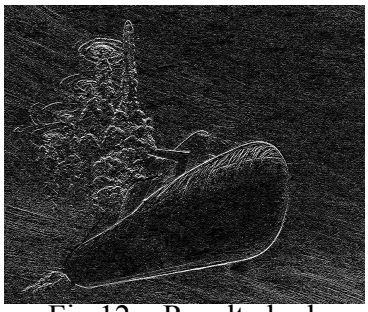
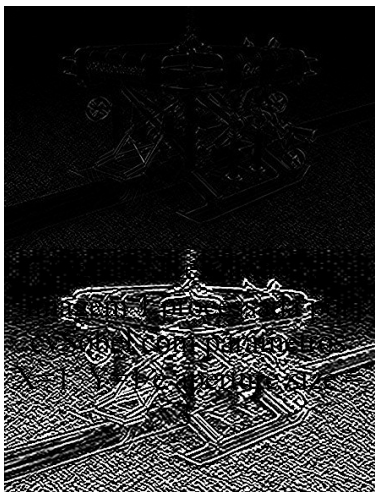


Fig.12 – Resultado da imagem 2 processada por cvSobel com parâmetros X=0, Y=1 e aperture size = SCHARR.



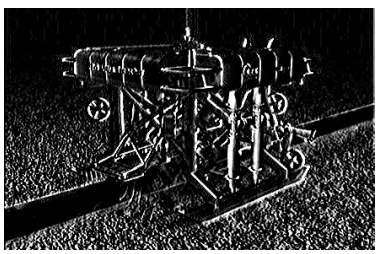


Fig.14 – Resultado da imagem 1 processada por cvSobel com parâmetros X=1, Y=1 e aperture size = SCHARR.

Fig.15 – Resultado da imagem 1 processada por cvSobel com parâmetros X=1, Y=2 e aperture size = 7.

2.3 – Laplace

A função abaixo calcula o Laplaciano da imagem.

```
void cvLaplace(const CvArr* src, CvArr* dst, int
aperture_size);
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem destino.

`aperture_size`

O tamanho do Kernel Laplace deve ser 1, 3, 5 ou 7. Em todos os casos, o kernel de tamanho aperture_size x aperture_size é utilizado separadamente para calcular a derivada. Quando o aperture_size for 1, é utilizado 3x1 ou 1x3 para x e y.

A função *cvLaplace* calcula o laplaciano, somando as segundas derivadas em x e y da função, ou seja:

$$\text{Laplace}\,(f) \equiv \frac{\partial^2 f}{\partial x^2} + \frac{\partial^2 f}{\partial y^2}$$

sendo as derivadas parciais obtidas usando o operador de Sobel, considerando o aperture_size=1. O resultado gera a derivada rápida, que é análogo a convoluir a imagem original com o kernel:

$$\begin{vmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{vmatrix}$$

Portanto, o OpenCV, na execução do Laplaciano, usa o operador Sobel no seu cálculo. Assim como na função *cvSobel*, não ocorre nenhuma alteração no tamanho da imagem, o que poderia gerar sobrecarga devido à imagem destino possuir números de maior valor absoluto. Entretanto neste caso, utiliza-se o mesmo tipo de imagem fonte e destino como ocorre na função *cvSobel.*

Da mesma forma, é possível processar imagens de 8-bits e 32-bits de ponto flutuante, e as imagens destino devem possuir o mesmo formato e apenas um canal; o resultado pode ser convertido de volta para 8-bits usando as funções *cvConvertScale* ou *cvConvertScaleAbs*. É possível utilizar variáveis de imagens do tipo *IplImage*.

A 1ª derivada identifica os máximos e mínimos de um sinal e a segunda derivada identifica as suas inflexões. O operador Laplaciano é uma composição de segundas derivadas ao longo do eixo-x e do eixo-y, que exibem zeros quando valores pequenos são adjacentes a valores altos. E nesses zeros, a função *cvLaplace* exibe os valores brancos. Desta maneira, uma das suas aplicações mais comuns é a identificação de bordas. A identificação baseia-se na diferença de valores entre pixels vizinhos, e a intensidade do branco é proporcional a essa diferença.

Associado ao operador de Laplace, usam-se filtros para suavizar o efeito de ruídos ou diferenças indesejáveis de gradiente devido à maior sensibilidade ajustada de acordo com o tamanho do Kernel – os maiores, por processarem um número maior de pixels, apresentam maior sensibilidade. Podemos observar nas imagens das Figs. 16 a 21 este efeito gradualmente proporcional aos tamanhos possíveis de kernels sobre as imagens de dutos 1 e 2.

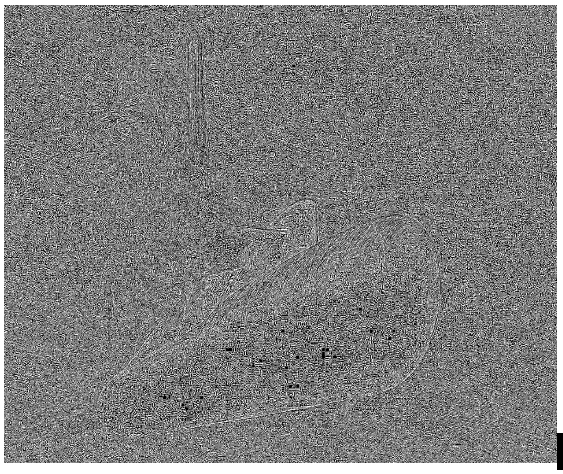


Fig.16 – Resultado da imagem 2 processada por cvLaplace com aperture size = 1.

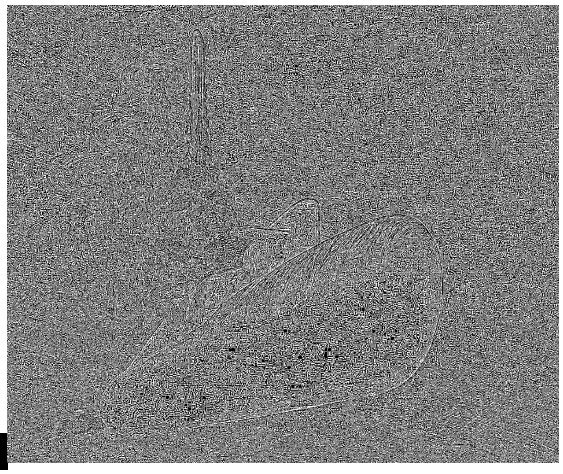


Fig.17 – Resultado da imagem 2 processada por cvLaplace com aperture size = 3.

Fig.18 – Resultado da imagem 2 processada por cvLaplace com aperture size = 5.

Fig.19 – Resultado da imagem 2 processada por cvLaplace com aperture size = 7.

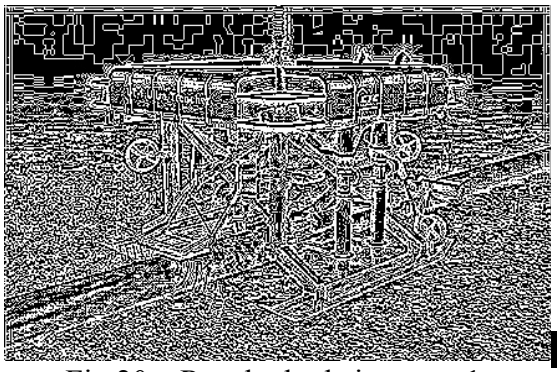


Fig.20 – Resultado da imagem 1 processada por cvLaplace com aperture size = 1.

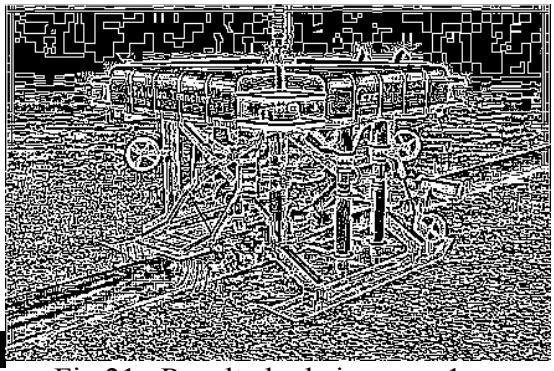


Fig.21– Resultado da imagem 1 processada por cvLaplace com aperture size = 3.

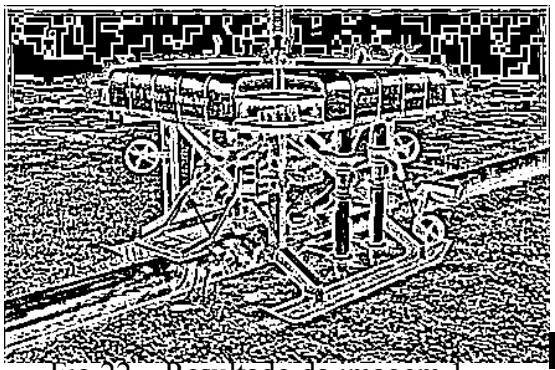


Fig.22 – Resultado da imagem 1 processada por cvLaplace com aperture size = 3.

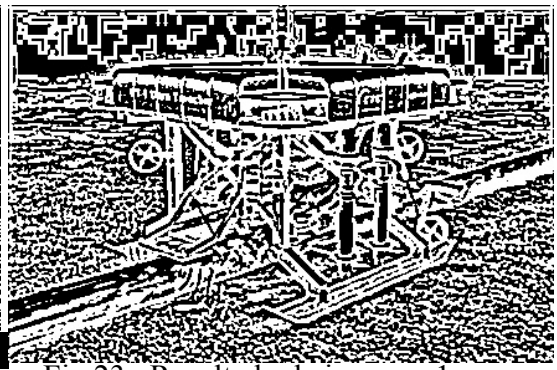


Fig.23– Resultado da imagem 1 processada por cvLaplace com aperture size = 5.

2.4 – Canny

Esta função implementa o algoritmo para detecção de bordas Canny

```
void cvCanny( const CvArr* src, CvArr* edges, double
threshold1, double threshold2, int aperture_size );
```

sendo:

`src`

Imagem fonte.

`edges`

Imagem do Contorno.

`threshold1`

O primeiro threshold.

`threshold2`

O segundo threshold.

`aperture_size`

Aperture size análogo ao parâmetro da função *cvSobel*.

Este método corresponde a um aperfeiçoamento do algoritmo de detecção de bordas feito por J. Canny (*A Computational Approach to Edge Detection)*. Diferente do operador Laplaciano, após uma suavização com o algoritmo de *smooth*, realiza a 1ª deri-

vada em x e em y e é combinado com 4 derivadas direcionais: através da hipotenusa e da função arctg y/x aproxima-se a função do setor que possui o maior gradiente e realiza-se a supressão das derivadas nos demais setores. Desta forma, encontra-se o máximo no setor de maior gradiente e desconsideram-se as derivadas nas demais direções. Analogamente, este algoritmo apresenta na *imagem destino* os pontos de descontinuidade da *imagem fonte*, sendo utilizado amplamente para detecção de bordas.

A maior diferença em relação à *cvLaplace*, no entanto, são os dois *thresholds.* Esses valores compõem os limites inferior e superior da chamada "*hysteresis threshold*". Trata-se de eliminar valores abaixo do menor dos dois limiares, e identificar os acima do maior dos dois. Este último é responsável por definir a borda, encontrando os segmentos mais fortes do contorno. Os valores entre os dois só serão considerados caso estejam conectados a um pixel forte, ou seja, um pixel com valor acima do limite superior.

Segundo Gary Bradski e Adrian Kaehlerno, no livro Learning OpenCV *"[...] Canny recommended a ratio of high:low threshold between 2:1 and 3:1. [...]"*. Nas Figs. 24 a 31, podemos observar o comportamento das imagens das Figs. 1 e 2 quando passam pela função com os filtros de Canny com os parâmetros indicados nas figuras respectivamente pelo valor do primeiro e segundo threshold, seguido do *aperture size*. Podemos observar, agora analisando estas figuras, que a relação 55:0 mostra um desempenho com mais ruídos comparada com as imagens com a relação 155:110. Entretanto, nota-se que para o caso particular dos canos, o limiar inferior elevado faz com que haja muita descontinuidade nas bordas. Para a imagem da Fig.1, o desempenho é superior para razão 155:110. Já para a imagem da Fig.2, esta razão mantém o ruído no duto e não apresenta a sua borda inferior. Nota-se desta forma que a solução para este tipo de imagem não é trivial, assim como o uso da proporção 2:1 a 3:1 é limitado. Ao selecionar limiares, cada caso deve ser analisado particularmente.

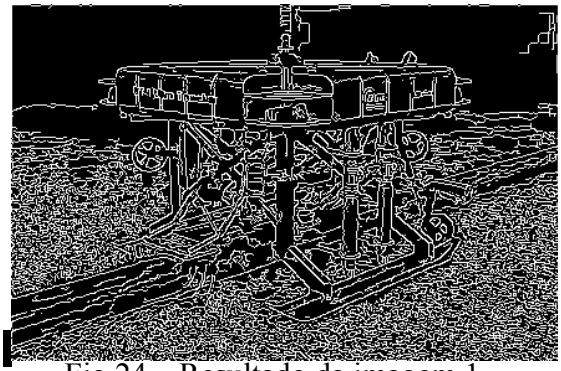


Fig.24 – Resultado da imagem 1 processada por cvCanny com T1= 0, T2= 55 e aperture size = 3.

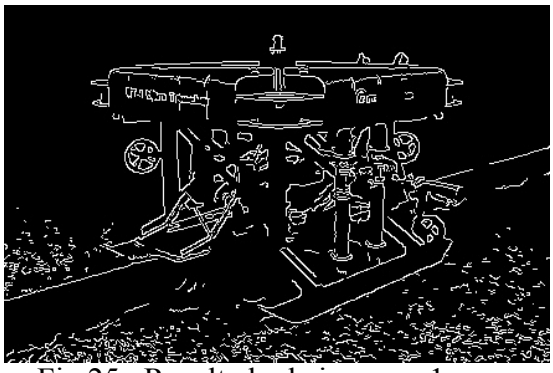


Fig.25– Resultado da imagem 1 processada por cvCanny com T1= 110, T2= 155 e aperture size = 3.

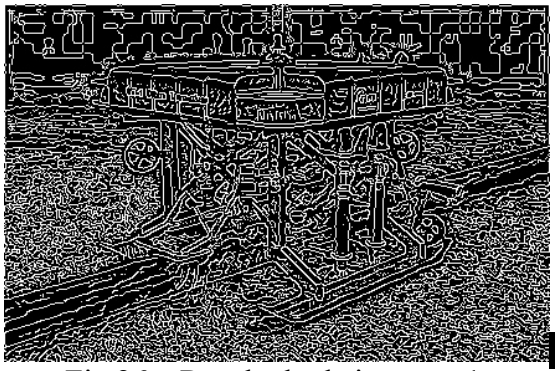


Fig.26 – Resultado da imagem 1 processada por cvCanny com T1= 0, T2= 55 e aperture size = 5.

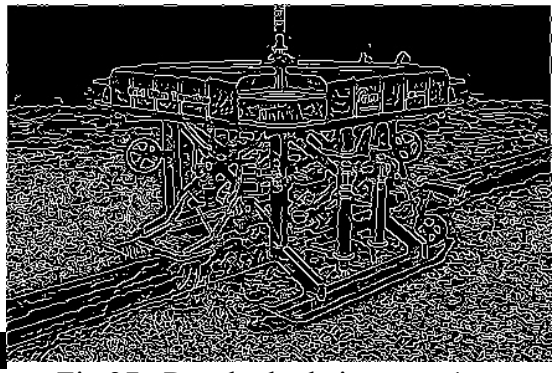


Fig.27– Resultado da imagem 1 processada por cvCanny com T1= 110, T2= 155 e aperture size = 5.

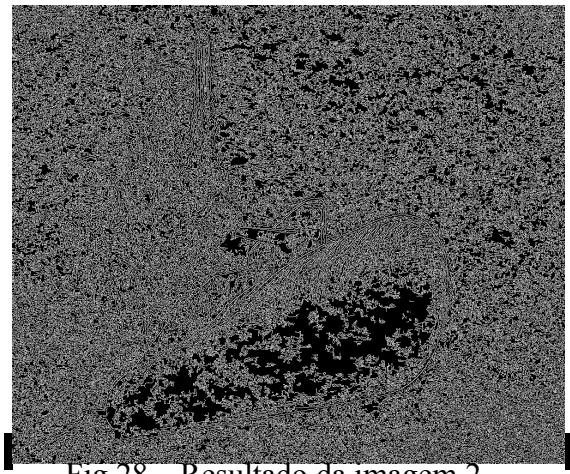


Fig.28 – Resultado da imagem 2 processada por cvCanny com T1= 0, T2= 55 e aperture size = 3.

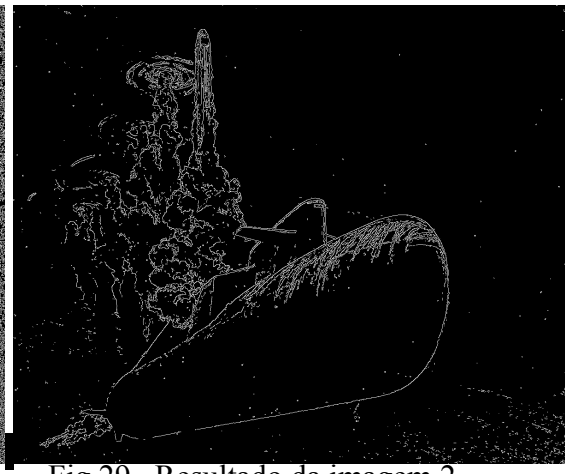


Fig.29– Resultado da imagem 2 processada por cvCanny com T1= 110, T2= 155 e aperture size = 3.

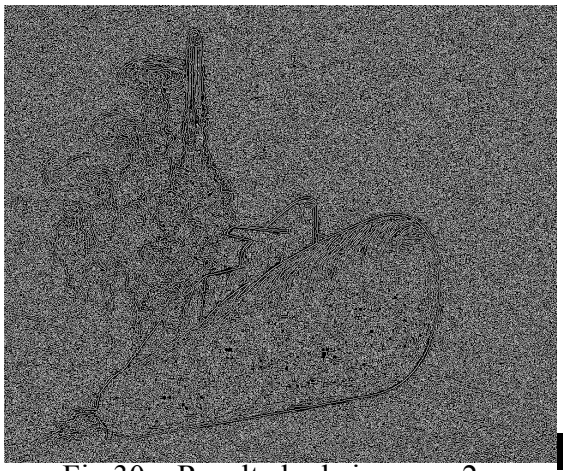


Fig.30 – Resultado da imagem 2 processada por cvCanny com T1= 0, T2= 55 e aperture size = 5.

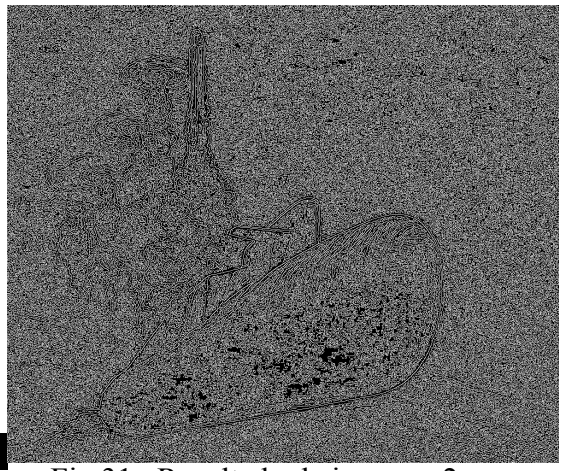


Fig.31– Resultado da imagem 2 processada por cvCanny com T1= 110, T2= 155 e aperture size = 5.

Originalmente as imagens são arquivos do tipo CvArr; entretanto, a função aceita arquivos do tipo IplImage. A *imagem fonte* e a *imagem destino* devem possuir um só canal e ser em tons de cinza, embora o desempenho seja ótimo para imagens binárias.

2.5 – PreCornerDetect

Esta função calcula as características do mapa para detecção de quinas.

```
void cvPreCornerDetect(  const CvArr* src, CvArr* corners,
int aperture_size );
```

sendo:

`src`

Imagem fonte.

`corners`

Imagem da borda.

`aperture_size`

Aperture size análogo ao parâmetro da função *cvSobel*.

A função cvPreCornerDetect calcula a função:

$$D_x^2 D_{yy} + D_y^2 D_{xx} - 2 D_x D_y D_{xy},$$

onde D_k é a primeira derivada e D_{kj} é a segunda derivada da imagem.

Nas Figs. 32 e 33, observamos o comportamento da imagem da Fig. 1 quando passa pela função *PreCornerDetect* com *aperture_size* igual a 5 e 7 – com o valor igual a 3, o algoritmo não tem sensibilidade para a imagem.

Fig.32 – Resultado da imagem 1 processada por cvPreCornerDetect com aperture size = 5.

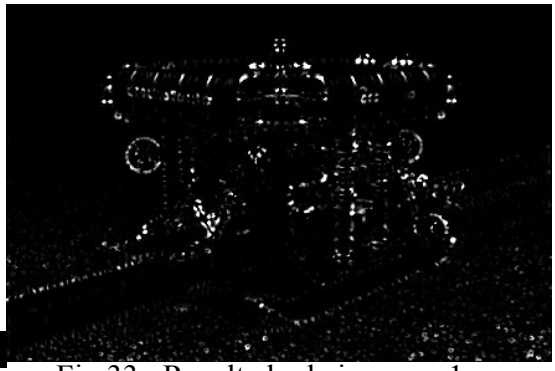


Fig.33– Resultado da imagem 1 processada por cvPreCornerDetect com aperture size = 7.

Originalmente as imagens são arquivos do tipo CvArr, entretanto, a função aceita arquivos do tipo IplImage. A mesma variável para *imagem fonte* pode ser utilizada como *imagem destino,* recebendo a informação processada e perdendo a original. A imagem de entrada deve ser de 1 canal e 8-bit (sem sinal). A *imagem destino* deve, além de ser do mesmo tipo que a *imagem fonte*, ter o mesmo tamanho.

2.6 – CornerHarris

Esta função detecta cantos usando algoritmo de Harris.

```
void cvCornerHarris( const CvArr* src, CvArr* edges, int
block_size, int aperture_size, double k);
```

sendo:

`src`

Imagem fonte.

`edges`

Imagem que grava a resposta da detecção de Harris. (Deve ter o mesmo tamanho que `src`)

`block_size`

Tamanho do bloco quadrado sobre o qual é calculado a derivada média.

`aperture_size`

Aperture size análogo ao parâmetro da função *cvSobel*.

`k`

Parâmetro livre de Harris.

A função *cvCornerHarris* detecta cantos de Harris. Para cada pixel é calculada a matriz de covariância 2x2 (M) sobre a vizinhança block_size x block_size. A seguir é gravado:

`det(M) - k*trace(M)`^{2};

em *edges*. M é uma matriz, definida como:

$$M(x,y)=\begin{bmatrix} \sum_{-K\le i,j\le K} w_{i,j} I_x^2(x+i,y+j) & \sum_{-K\le i,j\le K} w_{i,j} I_x(x+i,y+j) I_y(x+i,y+j) \\ \sum_{-K\le i,j\le K} w_{i,j} I_x(x+i,y+j) I_y(x+i,y+j) & \sum_{-K\le i,j\le K} w_{i,j} I_y^2(x+i,y+j) \end{bmatrix}$$

onde $w_{i,j}$ é um peso que pode ser uniforme, mas é frequentemente usado para criar uma ponderação Gaussiana.

Os cantos, pela definição de C. Harris, em "*A combined corner and edge detector*", são lugares na imagem em que a matriz de autocorrelação das segundas derivadas tem dois grandes autovalores. Isto significa dizer que existe uma diferença de gradiente (borda) em pelo menos duas direções distintas, centradas em torno do ponto em questão. Trata-se da identificação de cantos através de sua definição: o encontro de duas arestas (no caso, duas bordas).

Uma propriedade interessante deste algoritmo descrita por Gary Bradski e Adrian Kaehlerno, no livro "Learning OpenCV" é a sua aplicabilidade em objetos em movimento. Ao comparar imagens com objetos em posições diferentes, é possível acompanhá-los apenas por seus cantos. Estes, ao serem calculados possuem autovalores específicos.

Analogamente às funções descritas anteriormente, embora as imagens dos parâmetros sejam arquivos do tipo CvArr, a função aceita arquivos do tipo IplImage.

Nas Figs. 34 a 36, podemos observar o comportamento da imagem da Fig. 1 quando passa pela função *cvHarris*; sendo que as figuras são nomeadas pelos valores, respectivamente, do block size, aperture size e do parâmetro k.

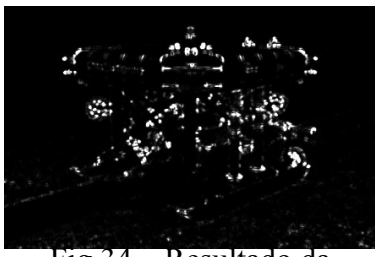


Fig.34 – Resultado da imagem 1 processada por cvCornerHarris com block size = 3, aperture size = 7 e K= $4x10^{-12}$.

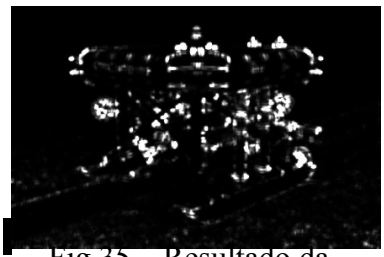


Fig.35 – Resultado da imagem 1 processada por cvCornerHarris com block size = 5, aperture size = 7 e K= $4x10^{-12}$.

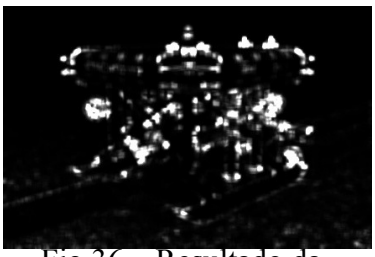


Fig.36 – Resultado da imagem 1 processada por cvCornerHarris com block size = 7, aperture size = 7 e K= $4x10^{-12}$.

2.7 – Resize

Esta função redimensiona a imagem.

```
void cvResize( const CvArr* src, CvArr* dst, int interpola-
tion);
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem Destino.

`interpolation`

Método de Interpolação

- CV_INTER_NN – interpolação com a vizinhança mais próxima;

- CV_INTER_LINEAR – interpolação bilinear (padrão);
- CV_INTER_AREA – reamostragem usando relação de área do pixel. É o método mais usado por dar resultados sem ondulações (moire-free). Para ampliação, é similar ao método CV_INTER_NN;
- CV_INTER_CUBIC – interpolação com spline cúbico.

A função redimensiona *src* para caber exatamente em *dst*. Quando redimensionamos uma imagem para um tamanho menor, por exemplo, ocorre perda de *pixels* no processo. Quando ocorre um aumento de imagem, serão necessários mais *pixels*. Para solucionar esses problemas, realizamos uma interpolação.

Esta função apresenta quatro tipos de interpolações, que são usadas tanto para reduzir quanto para aumentar uma imagem. O CV_INTER_NN realiza uma interpolação baseada no valor de seu pixel mais próximo. Já no CV_INTER_LINEAR, ocorre uma ponderação entre os pixels mais próximos. No CV_INTER_AREA, o pixel em questão é a média dos pixels antigos em uma área cujo centro fica onde está o novo pixel. Por último, há um ajuste baseado em uma *spline* cúbica – curva definida por pontos de controle e nós (pontos que pertencem à curva) – dos pixels adjacentes 4x4, e finalmente utiliza-se o valor correspondente na curva à região do pixel. Este método é chamado CV_INTER_CUBIC.

Originalmente as imagens são arquivos do tipo CvArr, entretanto a função aceita arquivos do tipo IplImage.

Nas Figs. 38 a 39, podemos observar o comportamento da imagem da Fig. 37 quando é redimensionada.

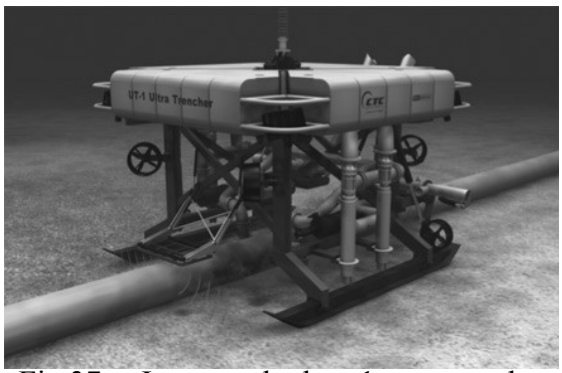


Fig.37 – Imagem do duto 1 em tamanho original.

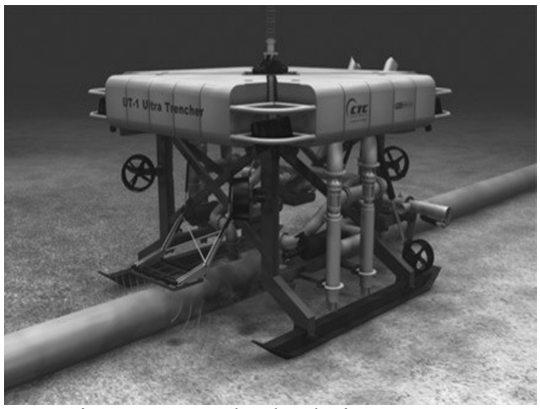


Fig.38– Resultado da imagem 1 processada por cvResize com tamanho 3000x4000.

Fig.39– Resultado da imagem 1 processada por cvResize com tamanho 4000x3000.

2.8 – Erode

Esta função corrói a imagem usando um kernel.

```
void cvErode( const CvArr* src, CvArr* dst, IplConvKernel*
element, int iterations );
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem destino.

`element`

Estrutura usada para fazer a erosão. Se for NULL, é utilizado uma estrutura retangular 3x3.

`iterations`

O número de iterações que é realizado na erosão.

A função cvErode corrói a *src* usando o kernel, elemento que determina a forma de uma vizinhança de pixel sobre qual o mínimo é usado para compor o novo pixel em questão:

dst=erode(src,element): $dst(x,y)=\min_{((x',y') \text{ no elemento})}src(x+x',y+y')$

O algoritmo utiliza um kernel, com uma forma e tamanho específico. O propósito da variável *element* é estabelecer um kernel. O kernel, que pode ter qualquer formato ou tamanho, tem um único ponto definido como âncora. Na maioria das vezes, o kernel é um pequeno quadrado sólido ou disco com o ponto de ancoragem no centro. Podemos pensar que o kernel é como um modelo ou máscara, e seu efeito de erosão é a de um operador que retorna o mínimo local.

As imagens fonte (*src*) e destino (*dst*) possuem o mesmo tamanho. Para preencher determinado pixel da *imagem destino,* selecionamos o menor valor dentre os pixels da *imagem fonte*, que estão dentro da máscara do kernel. Este kernel está ancorado no pixel de *src* com as mesmas coordenadas que o pixel de *dst* que está sendo calculado.

Utilizando a estrutura *IplConvKernel*, é possível criar um kernel com a função *cvCreateStructuringElementEx ()* e os kernels são liberados com *cvRelease StructuringElement ()*. Nesse processo de criação, um dos parâmetros da função permite estabelecer a forma do kernel. Por exemplo, se o parâmetro for CV_SHAPE_RECT faz com que o kernel seja retangular e caso seja necessário um kernel elíptico, a entrada deve ser CV_SHAPE_ELLIPSE.

Esta função permite repetir n vezes este algoritmo, onde n é o valor de iterações. Para imagens coloridas, devido ao fato da função só receber imagens com um canal, cada canal de cor deve ser processado separadamente.

A principal utilidade da função *erode* é eliminar ruídos em uma imagem, desgastando-os a cada iteração de forma que não afete outras regiões (necessariamente maiores), com conteúdo relevante.

Nas Figs. 40 a 48, vemos o efeito da redução de tamanho do cano devido às perdas periféricas pelo fato do mínimo ser externo à figura, eliminando a cada iteração suas bordas. As figuras foram nomeadas com o número de iterações.

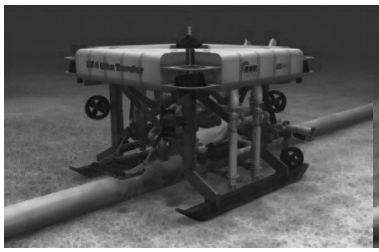


Fig.40 – Resultado da imagem 1 processada por cvErode com 1 iteração.

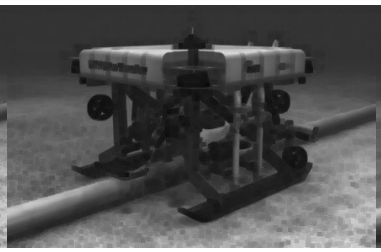


Fig.41 – Resultado da imagem 1 processada por cvErode com 2 iterações.

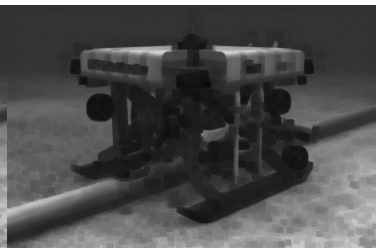


Fig.42 – Resultado da imagem 1 processada por cvErode com 3 iterações.

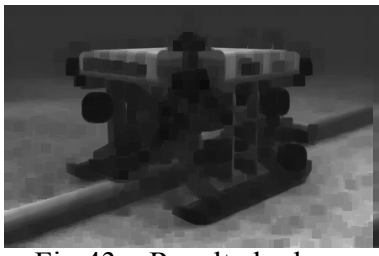


Fig.43 – Resultado da imagem 1 processada por cvErode com 5 iterações.

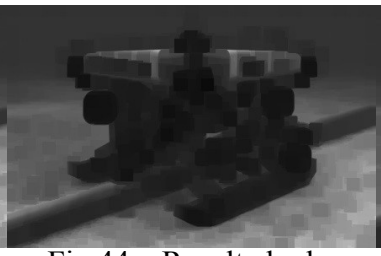


Fig.44 – Resultado da imagem 1 processada por cvErode com 7 iterações.

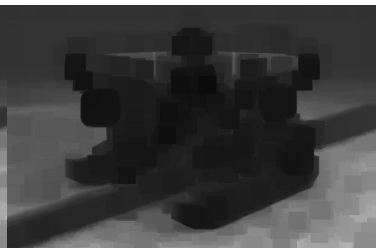


Fig.45 – Resultado da imagem 1 processada por cvErode com 10 iterações.

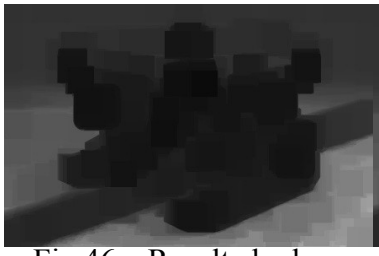


Fig.46 – Resultado da imagem 1 processada por cvErode com 15 iterações.

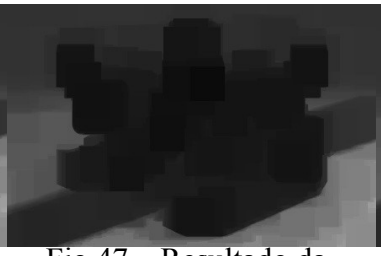


Fig.47 – Resultado da imagem 1 processada por cvErode com 20 iterações.

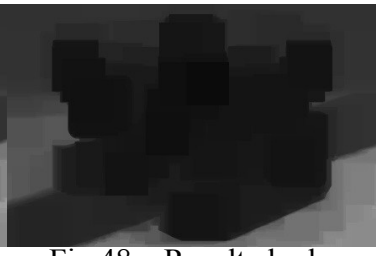


Fig.48 – Resultado da imagem 1 processada por cvErode com 25 iterações.

2.9 – Dilate

Esta função dilata a imagem usando elemento estruturado arbitrário

```
void cvDilate( const CvArr* src, CvArr* dst, IplConvKernel*
element=NULL, int iterations=1 );
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem destino.

`element`

Estrutura usada para fazer a operação dilatar. Se for NULL, é utilizado uma estrutura retangular 3x3.

`iterations`

O número de iterações que a operação é realizada.

A função cvDilate dilata a *src* usando o kernel, elemento que determina a forma de uma vizinhança de pixel sobre a qual o máximo é usado para compor o novo pixel em questão; ou seja

dst=dilate(src,element): $dst(x,y)=max_{((x',y') \text{ no elemento})}src(x+x',y+y')$

O algoritmo funciona de forma análoga à função *cvErode,* utilizando um kernel, com uma forma e tamanho específico. O propósito da variável *element* é estabelecer um kernel. O kernel, que pode ter qualquer formato ou tamanho, possui um único ponto definido como âncora. Na maioria das vezes, o kernel é um pequeno quadrado sólido ou disco com o ponto de ancoragem no centro. Podemos pensar que o kernel é como um modelo ou máscara, e seu efeito de dilatação é a de um operador que retorna o máximo local.

As imagens fonte (*src*) e destino (*dst*) possuem o mesmo tamanho. Para preencher determinado pixel da *imagem destino,* selecionamos o maior valor dentre os pixels da *imagem fonte*, que estão dentro da máscara do kernel. Este kernel está ancorado no pixel de *src* com as mesmas coordenadas que o pixel de *dst* que está sendo calculado.

Esta função permite repetir n vezes este algoritmo, onde n é o número total de iterações. Para imagens coloridas, devido ao fato da função só receber imagens com um canal, cada canal de cor deve ser processado separadamente.

A principal utilidade da função Dilate é juntar pixels desconexos em áreas próximas com o mesmo tom. A função aumenta o brilho da imagem, fundindo os pontos adjacentes em uma única forma do mesmo tom.

Nas Figs. 49 a 57, vemos o efeito da dilatação de tamanho do cano devido ao "derretimento" das bordas, ampliando o brilho para as regiões mais escuras da imagem (cujo kernel sobrepõe regiões com tons mais claros).

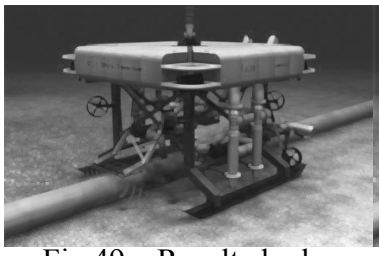


Fig.49 – Resultado da imagem 1 processada por cvDilate com 1 iteração.

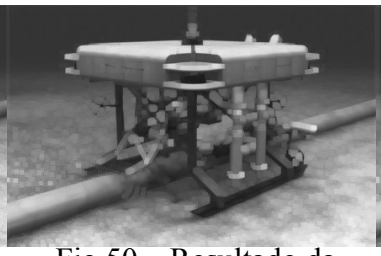


Fig.50 – Resultado da imagem 1 processada por cvDilate com 2 iterações.

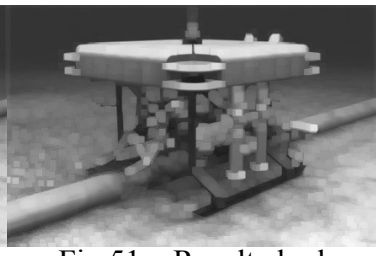


Fig.51 – Resultado da imagem 1 processada por cvDilate com 3 iterações.

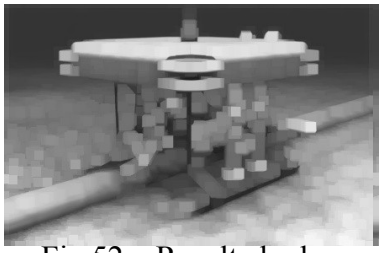


Fig.52 – Resultado da imagem 1 processada por cvDilate com 5 iterações.

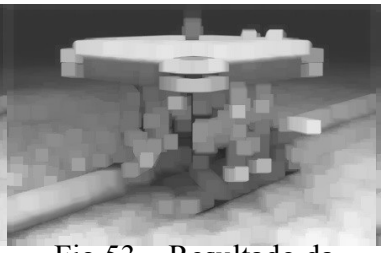


Fig.53 – Resultado da imagem 1 processada por cvDilate com 7 iterações.

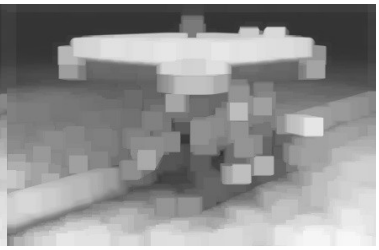


Fig.54 – Resultado da imagem 1 processada por cvDilate com 10 iterações.

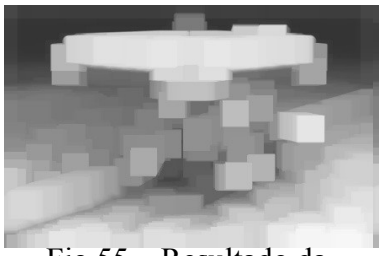


Fig.55 – Resultado da imagem 1 processada por cvDilate com 15 iterações.

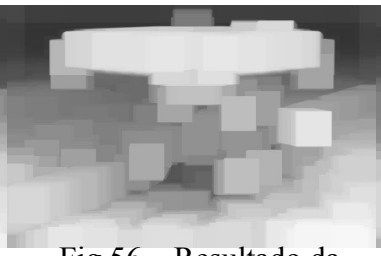


Fig.56 – Resultado da imagem 1 processada por cvDilate com 20 iterações.

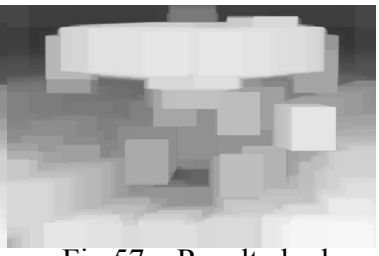


Fig.57 – Resultado da imagem 1 processada por cvDilate com 25 iterações.

2.10 – DrawContours

Esta função desenha o contorno ou preenche a imagem.

```
void cvDrawContours( CvArr *img, CvSeq* contour, CvScalar
external_color, CvScalar hole_color, int max_level, int
thickness=1,            int      line_type=8,      CvPoint
offset=cvPoint(0,0) );
```

sendo:

`img`

Imagens de onde os contornos serão desenhados.

`contour`

Ponteiro para o primeiro contorno. Este ponteiro é a raiz de uma variável ramificada.

`external_color`

Cor do contorno externo.

`hole_color`

Cor do interior do contorno (preenche apenas curvas fechadas).

`max_level`

Nível máximo para contornos desenhados. Se 0, somente o primeiro contorno é desenhado. Se 1, o primeiro contorno e todos os contornos depois deste no mesmo nível são traçados. Se 2, todos os contornos depois deste nível e seus filhos serão traçados; analogamente, a cada unidade adicionada, mais um nível abaixo é traçado nas ramificações da variável, cujo centro é *contour*. Se o valor de *max_level* for -1, a função desenha o contorno a partir de *contour*. E caso *max_level* seja -2, apenas os descendentes diretos de *contour* serão desenhados.

`Thickness`

Espessura das linhas desenhadas com os contornos. Se for negativo (por exemplo, igual a CV_FILLED), o interior dos contornos são preenchidos.

`line_type`

Tipo de segmentos do contorno.

A função *cvDrawContours* desenha o contorno na imagem se *thickness*>=0 ou preenche a área delimitada pelos contornos se *thickness*<0.

Nas Figs. 58 a 65, as imagens foram processadas por *cvCanny* e depois por *cvDrawContours,* com *thickness*<0.

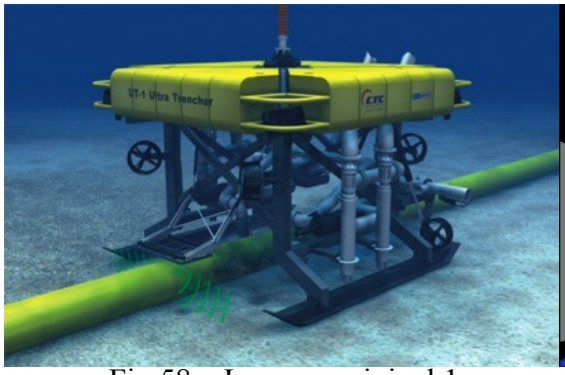


Fig.58 – Imagem original 1.

Fig.59– Resultado da imagem 1 processada por cvDrawContours.

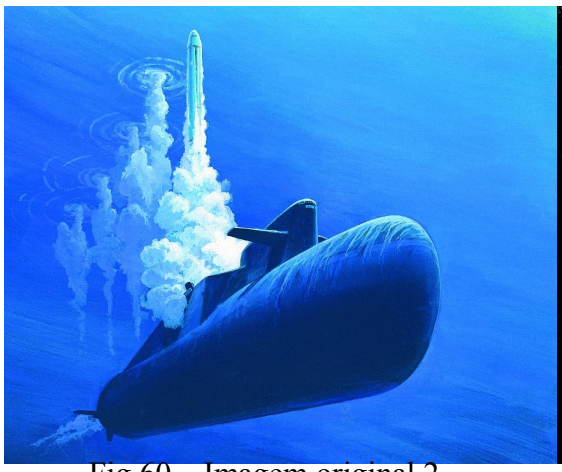


Fig.60 – Imagem original 2.

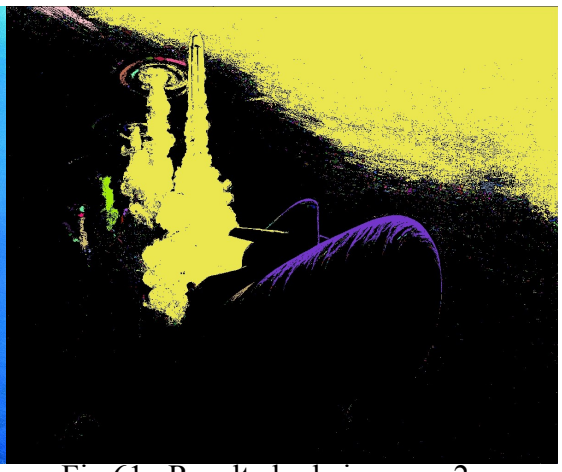


Fig.61– Resultado da imagem 2 processada por cvDrawContours.

Fig.62 – Imagem original 3.

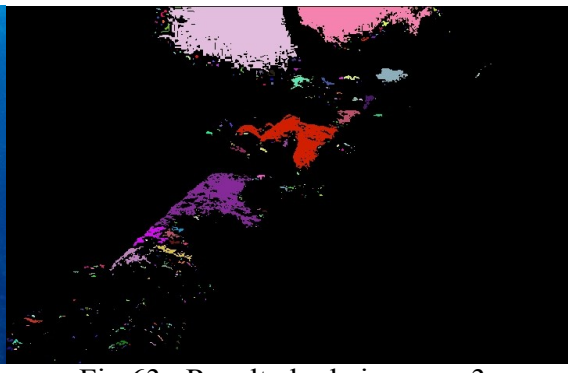


Fig.63– Resultado da imagem 3 processada por cvDrawContours.

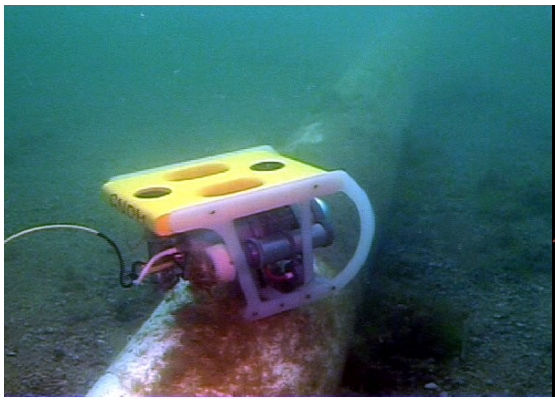


Fig.64 – Imagem original 4.

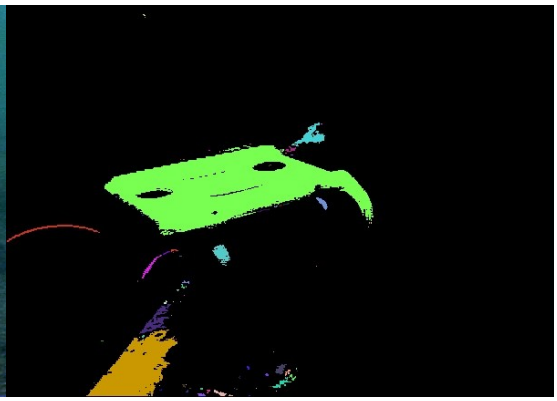


Fig.65– Resultado da imagem 4 processada por cvDrawContours.

2.11 – Smooth

Esta função suaviza a imagem de diversas formas.

```
void cvSmooth( const CvArr* src, CvArr* dst,int
smoothtype=CV_GAUSSIAN, int param1=3, int param2=0, double
param3=0, double param4=0 );
```

sendo:

`src`

Imagem fonte.

`dst`

Imagem destino.

`smoothtype`

Tipo de suavização.

A descrição dos demais parâmetros será incluída após detalharemos cada um dos cinco tipos de suavização. O uso dos parâmetros: `param1, param2, param3, param4` será explicado para cada caso.

CV_BLUR

Simple blur (borrão simples) - É a média aritmética dos pixels adjacentes. A quantidade de pixels que fazem parte da média é determinada pelo tamanho da "janela", estabelecido através dos parâmetros da função.

Os parâmetros `param1` e `param2` são responsáveis pelo tamanho da janelas (`param1×param2`). Estes *pixels* são somados e escalonados por 1/(`param1×param2`). Os parâmetros `param3` e `param4` não são utilizados por esta função.

Em relação às imagens fonte e destino, estas podem ter 8 bits sem sinal ou 32 bits com ponto flutuante. As imagens podem ter 1 ou 3 canais, ou seja, podem ser coloridas. As Figs. 66 e 67 exibem a imagem 1 processada com o tipo de suavização CV_BLUR.

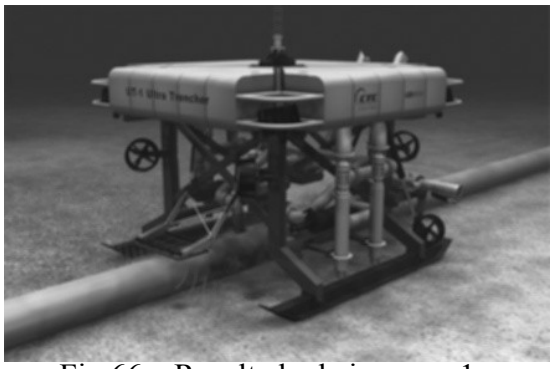


Fig.66 – Resultado da imagem 1 processada por cvSmooth tipo CV_BLUR com parâmetros 3, 3, 0 e 0.

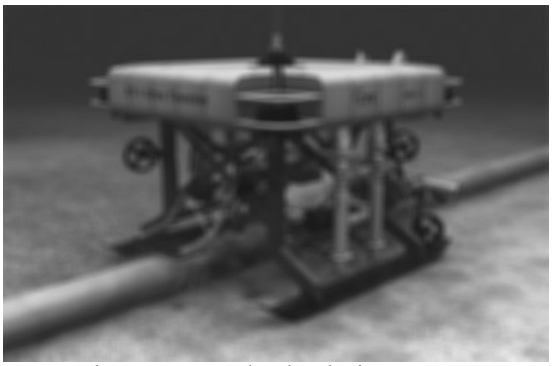


Fig.67– Resultado da imagem 1 processada por cvSmooth tipo CV_BLUR com parâmetros 7, 7, 0 e 0.

CV_BLUR_NO_SCALE

Simple blur without scaling (borrão simples sem escalonamento) – A diferença em relação ao CV_BLUR é que é realizada apenas no somatório dos pixels da janela sem escalonar por 1/(`param1×param2`). A vantagem é um processamento um pouco mais rápido comparado com a anterior.

Analogamente, os parâmetros `param1` e `param2` são responsáveis pelo tamanho das janelas (`param1×param2`) e os parâmetros `param3` e `param4` não são utilizados.

Nota-se que ao não escalonar a imagem, pode-se gerar um *overflow*. Desta forma, ao usar imagens fonte de 8 bits, a imagem destino deverá ter 16 ou 32 bits com sinal. Para imagens fonte com 32 bits com ponto flutuante, podem-se usar imagens destino do mesmo tipo, porque grandezas diferentes são representadas através da variação do ponto. Aumenta-se a ordem de grandeza e perde-se precisão. Em geral, para imagens, a perda de precisão é desprezível. Neste caso, as imagens só podem ter um canal. Para processar imagens em cor, deve-se processar cada canal separadamente e depois juntá-los.

CV_GAUSSIAN

Guassian Blur (borrão gaussiano) – é a convolução da imagem com o kernel gaussiano. O processo funciona da seguinte forma: a soma de uma janela de tamanho `param1×param2` constituída por pixels da imagem fonte é convoluída com o kernel Gaussiano.

O tamanho deste kernel pode ser definido de 3 maneiras. A primeira, através dos parâmetros `param3` e `param4`, formando um kernel `param3×param4`. A segunda, é no caso do último parâmetro ser nulo. O kernel terá tamanho `param3×param3`. Por último, caso os dois parâmetros sejam nulos, o tamanho do kernel é definido pelas equações abaixo:

$$\sigma_x = \left(\frac{n_x}{2} - 1\right) \cdot 0.30 + 0.80, \quad n_x = \text{param1}$$

$$\sigma_y = \left(\frac{n_y}{2} - 1\right) \cdot 0.30 + 0.80, \quad n_y = \text{param2}$$

O tamanho do kernel será $\sigma_x \times \sigma_y$.

Em relação às imagens fonte e destino estas podem ter 8 bits sem sinal ou 32 bits com ponto flutuante. As imagens podem ter um ou três canais, ou seja, podem ser coloridas. Este filtro, embora útil, tem menor velocidade de processamento. As Figs. 68 e 69 exibem a imagem 1 processada com tipo de suavização CV_GAUSSIAN.

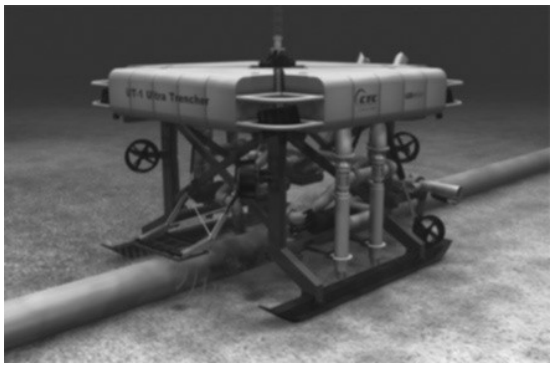


Fig.68 – Resultado da imagem 1 processada por cvSmooth tipo CV_GAUSSIAN com parâmetros 3, 0, 0 e 0.

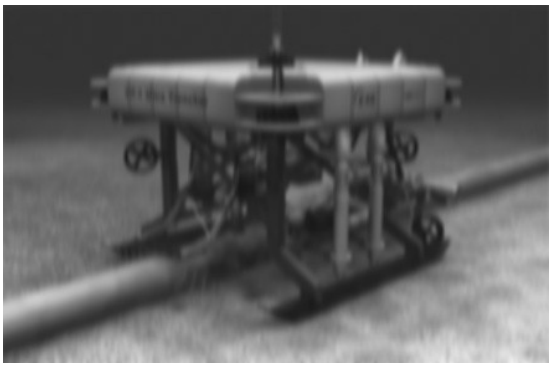


Fig.69– Resultado da imagem 1 processada por cvSmooth tipo CV_GAUSSIAN com parâmetros 3, 0, 3 e 0.

CV_MEDIAN

Median filter (filtro mediano) – O filtro seleciona o valor mediano dos pixels de uma janela quadrada e atribui este valor ao pixel da posição correspondente na imagem destino.

Diferente das demais funções, a janela do CV_MEDIAN é quadrada e formada apenas pelo primeiro parâmetro depois de `smoothtype`, cujo tamanho é `param1×param1`.

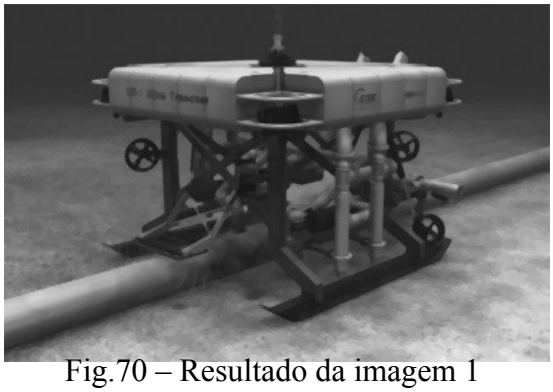


Fig.70 – Resultado da imagem 1 processada por cvSmooth tipo CV_MEDIAN com parâmetros 3, 0, 0 e 0.

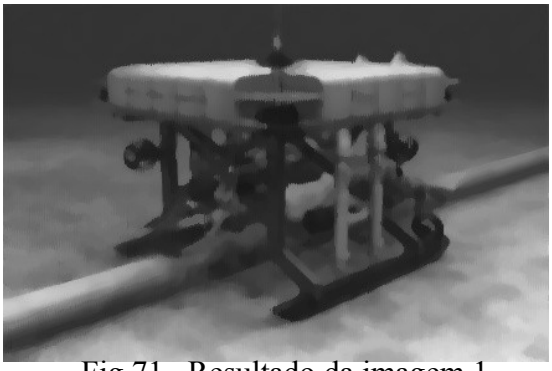


Fig.71– Resultado da imagem 1 processada por cvSmooth tipo CV_MEDIAN com parâmetros 7, 0, 0 e 0.

A imagem fonte e destino devem ser do tipo 8 bits sem sinal. Entretanto é possível usar imagens coloridas, de três ou quatro canais, além das com apenas um canal. Este tipo de filtro é sensível a ruídos, pois estes podem ser utilizados como centro das

janelas, ampliando seu tamanho à área da janela toda. As Figs. 70 e 71 exibem a imagem 1 processada com tipo de suavização CV_MEDIAN.

CV_BILATERAL

Bilateral filtering (filtro bilateral) - aplicando filtro 3x3 bilateral com a cor sigma =`param1` e espaço sigma=`param2`. O filtro Bilateral é análogo ao filtro gaussiano; entretanto, o filtro gaussiano tem melhor desempenho em imagens com pequenas variações entre os pixels, de forma que um ruído seja convoluído com o Kernel eliminando-o. Porém ao aplicá-lo em imagens com bordas, estas sofrem a suavização correlacionado-as com os pixels adjacentes. Com um processamento mais lento, o Filtro Bilateral suaviza a imagem, mas o algoritmo não é aplicado próximo a bordas.

O primeiro parâmetro está associado à largura da borda, já o segundo está associado à largura de cores, ou seja, o quão maior é o segundo parâmetro, maior é a quantidade de tons de cores envolvidas na suavização. Nota-se que a diferença entre os pixels de uma borda precisa ser grande o suficiente para não ser suavizada; os gradientes próximos são uniformizados, mas a bordas e os tons são mantidos.

Quanto ao uso de imagens, estas podem ter um ou três canais de cores, e no caso de três, cada *pixel* é calculado pela media dos correspondentes ao mesmo nos três canais. O tipo da imagem deve ser 8 bits sem sinal. As Figs. 72 e 73 exibem a imagem 1 processada com tipo de suavização CV_BLUR.

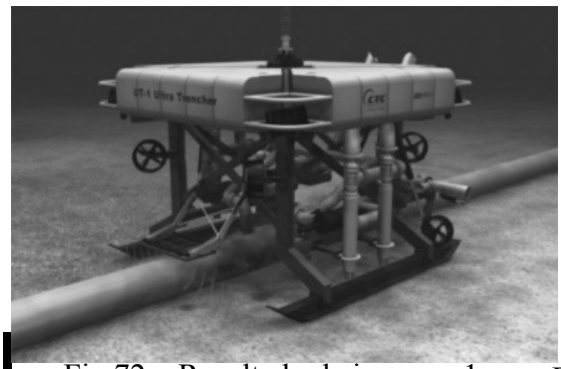


Fig.72 – Resultado da imagem 1 processada por cvSmooth tipo CV_BILATERAL com parâmetros 3, 3, 100 e 100.

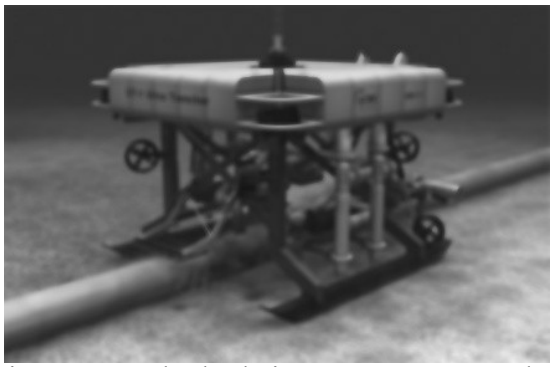


Fig.73– Resultado da imagem 1 processada por cvSmooth tipo CV_BILATERAL com parâmetros 7, 7, 100 e 100.

Na primeira imagem, ocorre a suavização sem comprometimento das bordas. Isto é diferente do caso da segunda imagem devido aos valores altos dos parâmetros `param1` e `param2`.

2.12 – Threshold

Esta função aplica um threshold com parâmetros pré-fixados em uma imagem.

```
void cvThreshold( const CvArr* src, CvArr* dst, double
threshold,double max_value, int threshold_type );
```

sendo:

`src`

Imagem fonte

`Dst`

Imagem destino

`Threshold`

Limiar de corte.

`max_value`

Limiar máximo.

`threshold_type`

Tipo de threshold.

A função threshold, a partir de um limiar, filtra uma imagem executando um de seus cinco operadores:

CV_THRESH_BINARY

Se $(src_i > Threshold)$ então $dst_i = max\_value$, senão $dst_i = 0$.

O modelo da Fig. 74 exemplifica o comportamento da função.

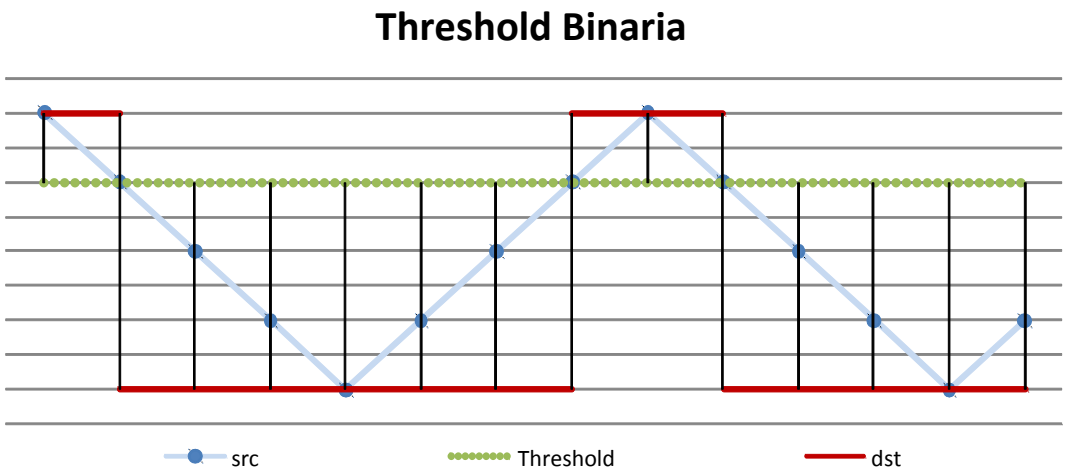


Fig. 74 - Modelo CV_THRESHOLD_BINARY

CV_THRESH_BINARY_INV

Se $(src_i > Threshold)$ então $dst_i = 0$, senão $dst_i = max\_value$

O modelo da Fig. 75 exemplifica o comportamento da função.

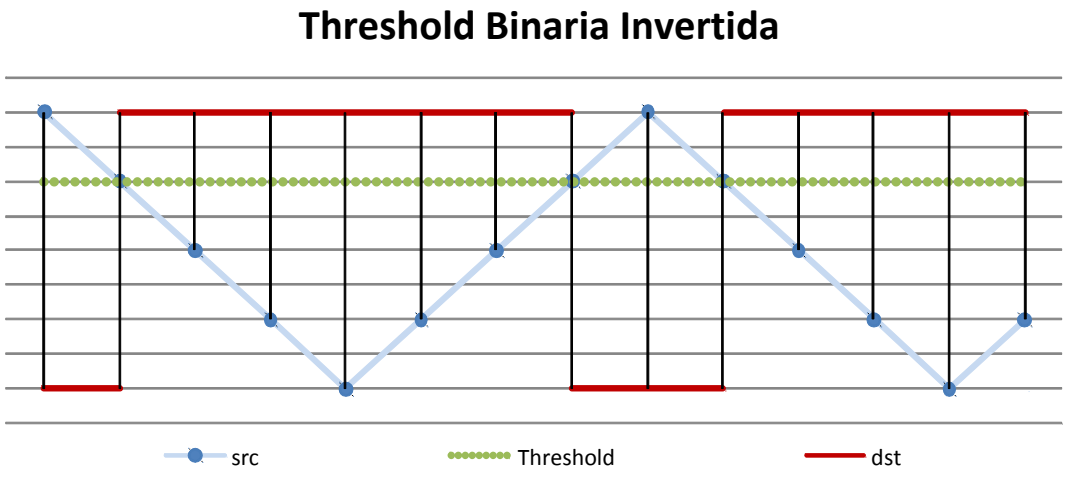


Fig. 75 - Modelo CV_THRESHOLD_BINARY_INV

CV_THRESH_TRUNC

Se $(src_i > Threshold)$ então $dst_i = Threshold$, senão $dst_i = src_i$

O modelo da Fig. 76 exemplifica o comportamento da função.

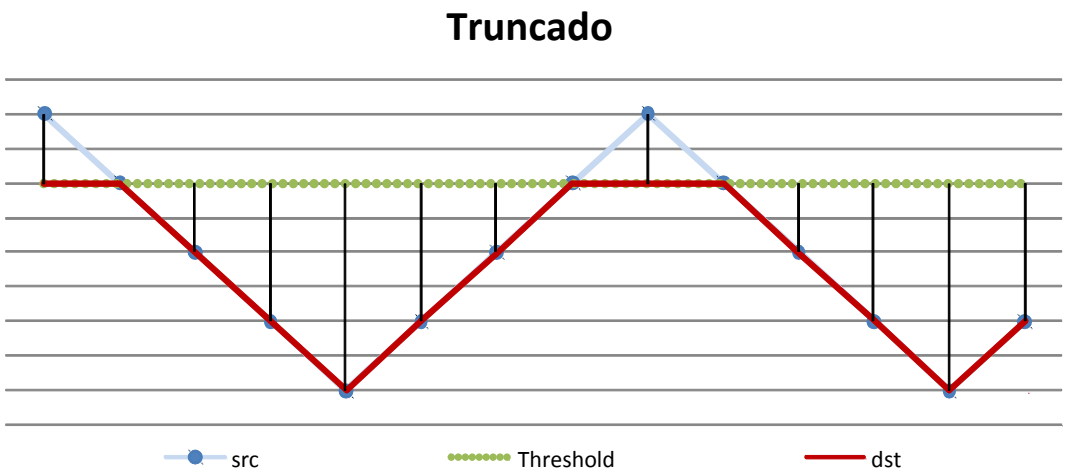


Fig. 76 - Modelo CV_THRESHOLD_TRUNC

CV_THRESH_TOZERO

Se $(src_i > Threshold)$ então $dst_i = src_i$, senão $dst_i = 0$

O modelo da Fig. 77 exemplifica o comportamento da função.

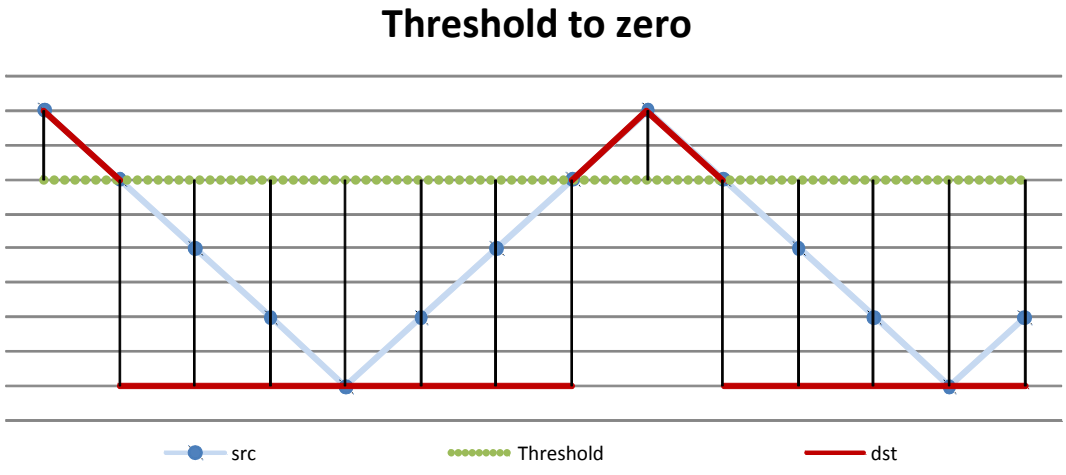


Fig. 77- Modelo CV_THRESHOLD_TOZERO

CV_THRESH_TOZERO_INV

Se $(src_i > Threshold)$ então $dst_i = 0$, senão $dst_i = src_i$

O modelo da Fig. 78 exemplifica o comportamento da função.

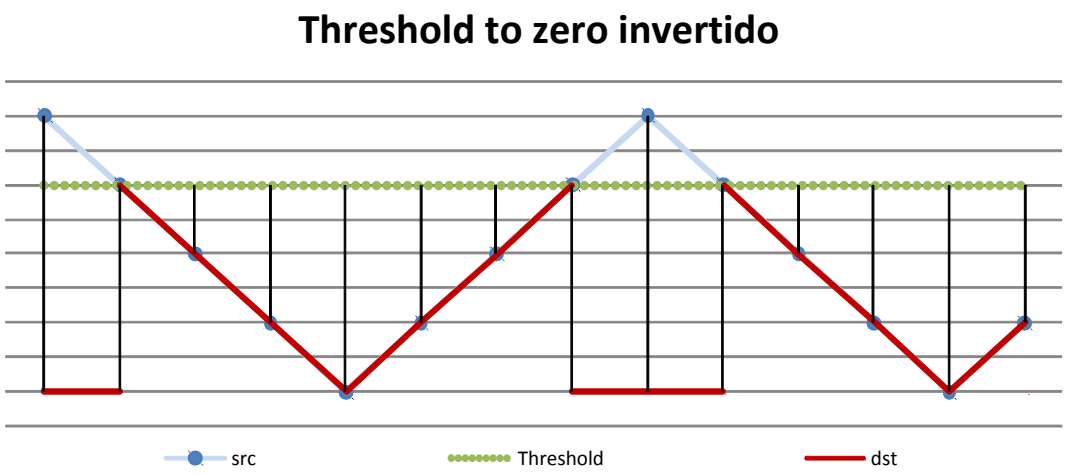


Fig. 78 Modelo CV_THRESHOLD_TOZERO_INV

De acordo com cada tipo de threshold, é possível selecionar parte da imagem e decidir por descartar ou substituir por um valor. O algoritmo é simples, e no caso das imagens testadas, podemos perceber um intervalo útil para delimitar as bordas dos objetos. Os valores de threshold entre **60** e **85**, demonstrados nas Figs. 79 a 93 (especificados na legenda), exemplificam o desempenho próximo ao valor mínimo e máximo de um intervalo ótimo. No primeiro caso algumas bordas são perdidas e, ao utilizar um valor muito superior, aumentam significativamente a presença de ruídos, mas evidenciando melhor as bordas. Este é o problema a ser resolvido para detecção de bordas.

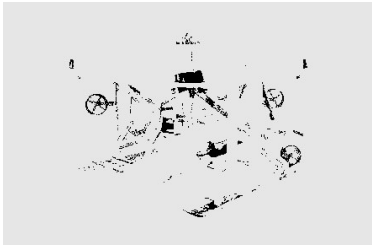


Fig.79 – Resultado da imagem 1 processada por Threshold Binary com Threshold = 30 e Max value = 230.

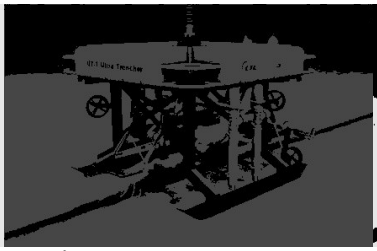


Fig.80 – Resultado da imagem 1 processada por Threshold Binary com Threshold = 60 e Max value = 70.

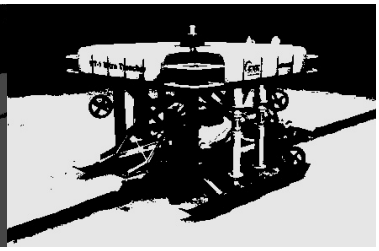


Fig.81 – Resultado da imagem 1 processada por Threshold Binary com Threshold = 85 e Max value = 230.

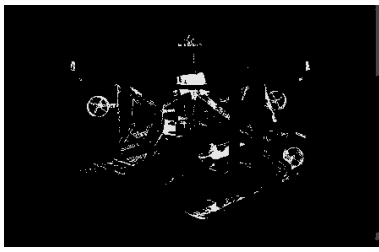


Fig.82 – Resultado da imagem 1 processada por Threshold Binary Inv com Thres = 30 e Max value = 230.

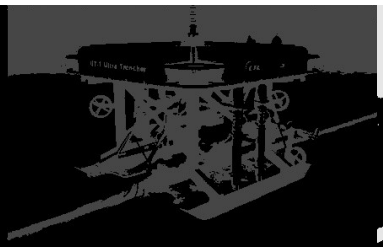


Fig.83 – Resultado da imagem 1 processada por Threshold Binary Inv com Thres = 60 e Max value = 70.

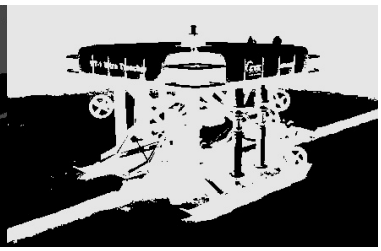


Fig.84 – Resultado da imagem 1 processada por Threshold Binary Inv com Thres = 85 e Max value = 230.

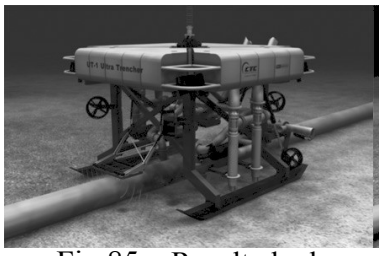


Fig.85 – Resultado da imagem 1 processada por Threshold to Zero com Thres = 30 e Max value = 230.

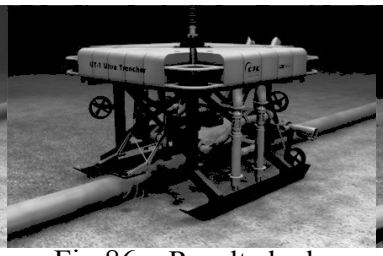


Fig.86 – Resultado da imagem 1 processada por Threshold to Zero com Thres = 60 e Max value = 70.

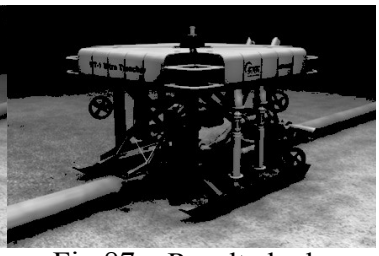


Fig.87 – Resultado da imagem 1 processada por Threshold to Zero com Thres = 85 e Max value = 230.

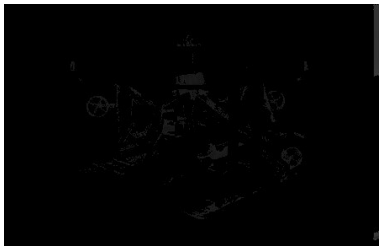


Fig.88 – Resultado da imagem 1 processada por Threshold to Zero Inv com Thres = 30 e Max value = 230.

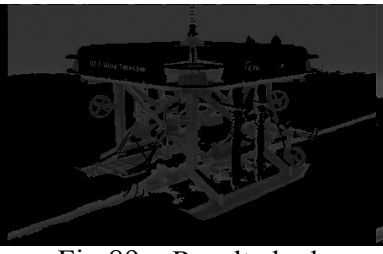


Fig.89 – Resultado da imagem 1 processada por Threshold to Zero Inv comThres = 60 e Max value = 70.

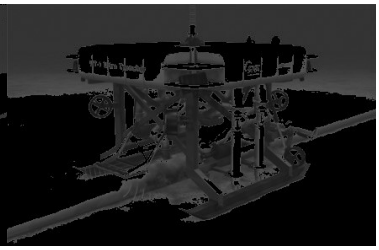


Fig.90 – Resultado da imagem 1 processada por Threshold to Zero Inv com Thres = 85 e Max value = 230.

Fig.91 – Resultado da imagem 1 processada por Threshold Trunc com Thres = 30 e Max value = 230.

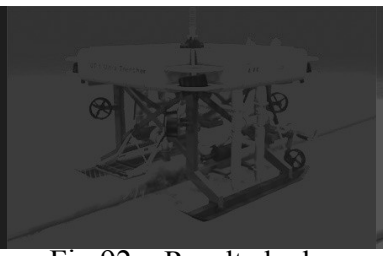


Fig.92 – Resultado da imagem 1 processada por Threshold Trunc comThres = 60 e Max value = 70.

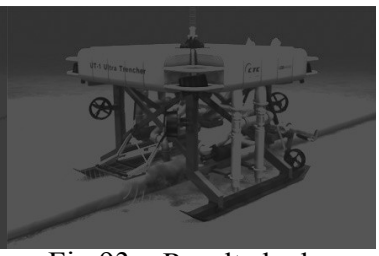


Fig.93 – Resultado da imagem 1 processada por Threshold Trunc com Thres = 85 e Max value = 230.

No que diz respeito às imagens, estas podem ser de 8 bits sem sinal ou 8 bits com ponto flutuante. No primeiro tipo, estas podem gerar *overflow*, mas não é o caso de imagens analisadas neste estudo. Para imagens coloridas, cada canal dever ser processado separadamente e concatenado em seguida.

2.13 – HoughLines2

A função encontra linhas em uma imagem binária usando a transformada de Hough.

```
CvSeq* cvHoughLines2( CvArr* image, void* line_storage, int
method, double rho, double theta, int threshold,double
param1, double param2 );
```

sendo:

`image`

Imagem fonte.

`line_storage`

O armazenamento das linhas detectadas.

`Method`

Há três tipos de métodos para executar Hough Lines 2:

- `CV_HOUGH_STANDARD` – Método padrão. Cada linha é representada por um plano polar (ρ, θ), onde cada coordenada é um número em ponto flutuante.
- `CV_HOUGH_PROBABILISTIC` – Transformada Probabilística de Hough. Retorna segmentos de linhas e não linhas inteiras. Todo segmento tem pontos inicial e final definidos.
- `CV_HOUGH_MULTI_SCALE` – é uma variação do `CV_HOUGH_STANDARD`. Nesse caso é usada uma ponderação utilizando os parâmetros `param1` e `param2` para melhorar a resolução.

rho

Distância em unidades de pixel.

Theta

Ângulo medido em radiano.

threshold

Uma linha retorna para função se o valor acumulado é superior ao threshold.

param1

É apenas utilizada na CV_HOUGH_PROBABILISTIC e CV_HOUGH_MULTI_SCALE. Seu valor no método CV_HOUGH_STANDARD é 0. O parâmetro funciona como o menor comprimento de linha em CV_HOUGH_PROBABILISTIC – nota-se que neste método não são geradas linhas, mas segmentos com pontos iniciais e finais da linha definidos. Sendo assim justifica-se apenas para este caso um valor mínimo a ser atingido para que se exiba a linha.

O último método, CV_HOUGH_MULTI_SCALE, utiliza o parâmetro como divisor da resolução da distância rho, formando uma resolução da distância acurada (rho/param1).

param2

Analogamente, é apenas utilizada na CV_HOUGH_PROBABILISTIC e CV_HOUGH_MULTI_SCALE com valor 0 no método CV_HOUGH_STANDARD. O parâmetro funciona como o máximo espaçamento entre pontos de uma linha em CV_HOUGH_PROBABILISTIC.

Em CV_HOUGH_MULTI_SCALE, utiliza-se o parâmetro como divisor da resolução do ângulo theta, formando uma resolução de ângulo acurada (theta/param2).

As imagens podem ser do tipo 8 bits sem sinal. Para um desempenho satisfatório, seu uso é associado à função *cvCanny* antes de ser processada para que o algoritmo defina os pontos com precisão. O Método CV_HOUGH_STANDARD, assim como sua variação CV_HOUGH_MULTI_SCALE, identifica linhas inteiras através de pontos gerando um número maior de linhas comparado com CV_HOUGH_PROBABILISTIC, usada principalmente quando há pontos formando poucas linhas longas, principalmente porque há limi-

tação mínima para o comprimento e a função só exibe segmentos. Podemos observar o desempenho do método `CV_HOUGH_PROBABILISTIC` através da Fig. 94.

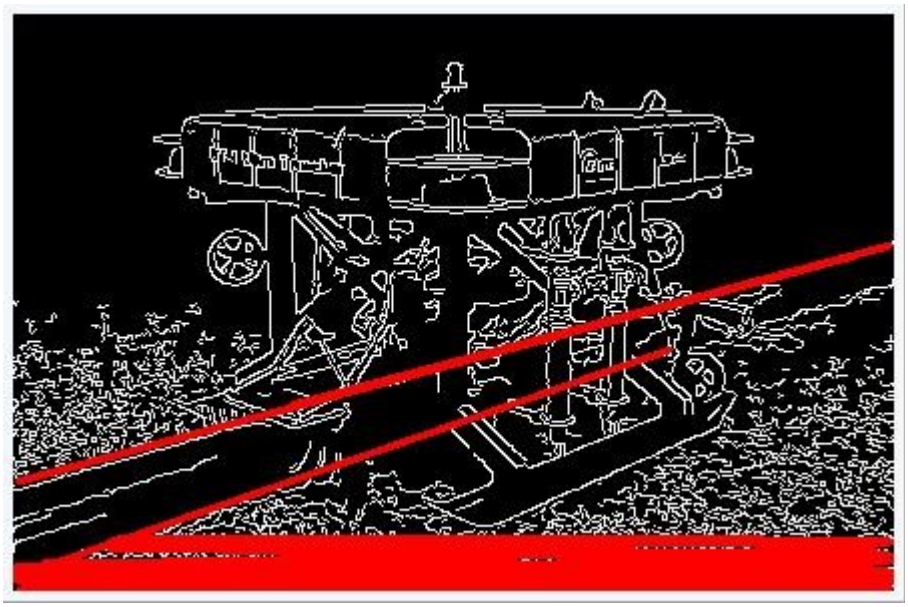


Fig. 94 - Resultado da função *cvHoughLines2* utilizando CV_HOUGH_PROBABILISTIC.

Capítulo 3

Algoritmo de Detecção de Bordas

3.1 – Rotina do programa

O objetivo do programa é detectar bordas de objetos, eliminando as letras nas imagens sem comprometer o resultado. O algoritmo é simples e foi inicialmente implementado em MATLAB. Além da rotina, foram feitas algumas alterações aproveitando as vantagens oferecidas pelo OpenCV. Ao executar o *threshold*, por exemplo, o programa permite escolher um dos cinco métodos para filtrar uma imagem, produzindo respostas diferenciadas para cada caso, como pode ser observado através de sua interface gráfica exibida na Fig.95.

Em um segundo momento exibir-se-á o código comentado. A seguir, serão mostradas algumas imagens com entradas específicas exibindo o melhor resultado possível para vários tipos de imagens.

Finalmente será mostrado o desempenho de uma função associada ao algoritmo inicial – cvHoughLines2 – nos casos falhos de detecção.

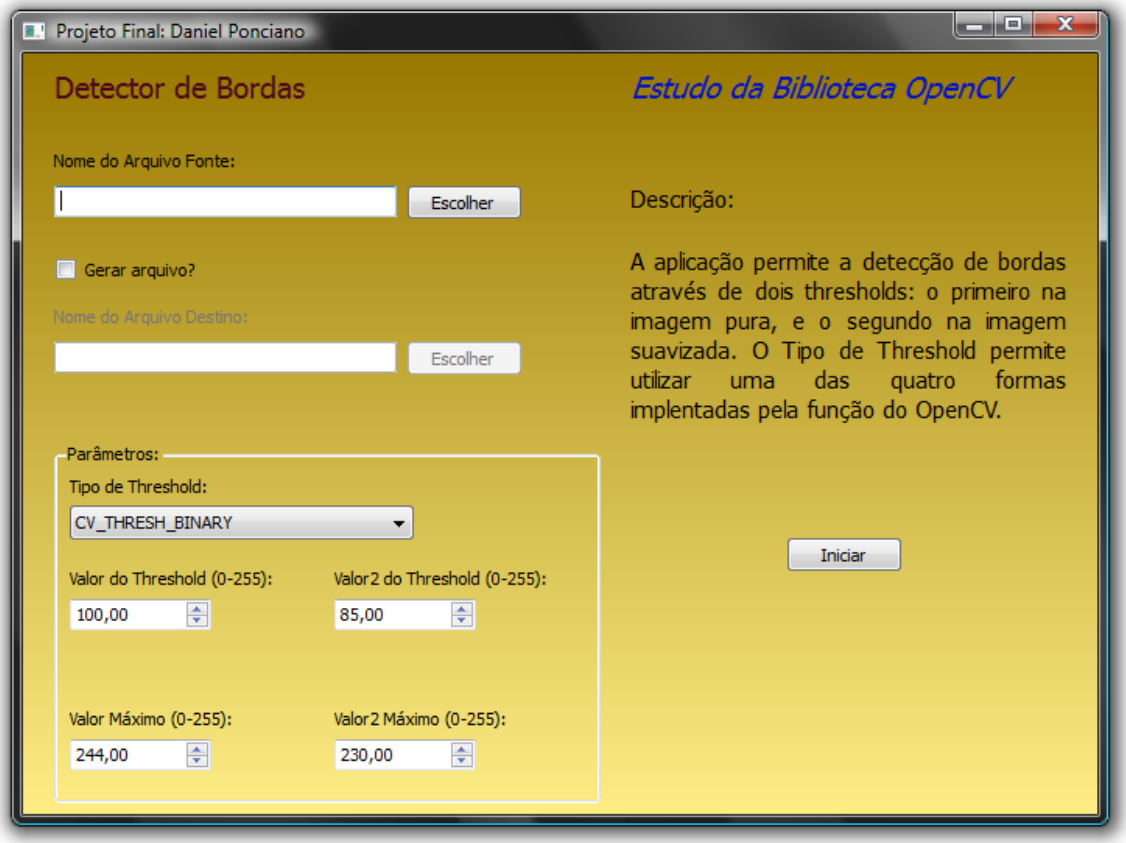


Fig. 95 - Interface Gráfica.

A seguir será detalhado o código, explicando as linhas principais para o funcionamento do algoritmo.

i) Inclusão de Bibliotecas.

```
#include <cv.h>
#include <cvaux.h>
```

```
#include <highgui.h>
#include <stdio.h>
#include <math.h>
#include "mainwindow.h"
#include "ui_mainwindow.h"
```

ii) Ocorre a construção da janela. A aplicação foi desenvolvida em QT, uma biblioteca gráfica aberta multiplataforma, atualmente pertencente a Nokia que pode ser utilizado com C e C++, entre outras linguagens, inclusive embarcada.

```
MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), ui(new Ui::MainWindowClass)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::on_checkBox_stateChanged(int state)
{
    if(state==Qt::Checked){
        ui->label_3->setEnabled(true);
        ui->lineEdit_2->setEnabled(true);
        ui->pushButton_2->setEnabled(true);
    }
    else{
        ui->label_3->setEnabled(false);
        ui->lineEdit_2->clear();
        ui->lineEdit_2->setEnabled(false);
        ui->pushButton_2->setEnabled(false);
    }
}

void MainWindow::on_pushButton_clicked()
{
```

```
    QString fileName = QFileDialog::getOpenFileName(this,
     tr("Escolher arquivo fonte"), "", tr("Imagens (*.png *.jpg
*.bmp)"));

    ui->lineEdit->setText(fileName);
}

void MainWindow::on_pushButton_2_clicked()
{
    QString fileName = QFileDialog::getSaveFileName(this,
     tr("Escolher arquivo de destino"), "", tr("Imagens (*.png *.jpg
*.bmp)"));

    ui->lineEdit_2->setText(fileName);
}
```

iii) Construção do botão "Iniciar" – seu acionamento inicia a rotina do algoritmo.

```
void MainWindow::on_pushButton_3_clicked()
{

  double threshold1;
  double max_value1;
  double threshold;
  double max_value;

  threshold1 = ui->doubleSpinBox_6->value();
  max_value1= ui->doubleSpinBox_7->value();
  threshold = ui->doubleSpinBox_5->value();
  max_value= ui->doubleSpinBox_8->value();
  /* Carregar imagem de entrada */

  const char* fonte = "zero";
  const char* destino = "zero";

  int tipo;

  fonte = ui->lineEdit->text().toAscii();
  destino = ui->lineEdit_2->text().toAscii();
  tipo = ui->comboBox->currentIndex();
```

iv) Carrega a imagem – caminho inserido no campo “Nome do Arquivo Fonte”.

```
IplImage* img_original = cvLoadImage(fonte);
```

v) Copia a imagem para outras duas variáveis auxiliares.

```
IplImage* img_erodetres = cvCreateImage( cvGetSize( img_original ),
IPL_DEPTH_8U, 1 );

IplImage* img_tres = cvCreateImage( cvGetSize( img_original ),
IPL_DEPTH_8U, 1 );
```

vi) As imagens são convertidas para tons de cinza para que possam ser processadas pela maioria das funções.

```
cvCvtColor( img_original, img_erodetres, CV_BGR2GRAY );
cvCvtColor( img_original, img_tres, CV_BGR2GRAY );
```

vii) Uma das imagens é corroída para que se eliminem as letras da figura sem que haja perda do tamanho.

```
cvErode( img_erodetres, img_erodetres,NULL, 3 );
```

viii) Ambas as imagens passam por um threshold, com parâmetros definidos pelo usuário. Deve-se escolher um dos cinco métodos para execução do cvThreshold.

```
cvThreshold( img_erodetres, img_erodetres, threshold1,
max_value1,tipo );


cvThreshold( img_tres, img_tres, threshold, max_value, tipo );
```

ix) As imagens então são multiplicadas binariamente (Xor) e a imagem final recebe mais uma corrosão antes que finalmente passe pelo algoritmo de detecção de bordas de Canny – nota-se que ocorre apenas uma homogeneização para os casos em que apresentam valores *max_value* diferentes para cada *threshold.* Apenas nos métodos *threshold to zero*, *threshold to zero invertido* e *threshold truncado* ocorre uma real

detecção de bodas por Canny, fortemente orientada pela descontinuidade gerada por cvThreshold.

```
  cvMul( img_erodetres, img_tres, img_tres,1  );

  cvErode( img_tres, img_tres, NULL, 1 );

  cvCanny( img_tres, img_tres, 0, 55, 3 );

  if(ui->checkBox->checkState())

  cvSaveImage(destino, img_tres);

  cvNamedWindow( "Imagem Original", 0 );

  cvNamedWindow( "Borda", 0 );

  cvShowImage( "Imagem Original", img_original );

  cvShowImage( "Borda", img_tres );

  int keyCode = cvWaitKey( );

  cvReleaseImage( &img_tres );
  cvReleaseImage( &img_erodetres );
  cvReleaseImage( &img_original );

  cvDestroyWindow( "Imagem Original" );
  cvDestroyWindow( "Borda" );

}
```

Fim do código.

3.2 – Desempenho do Algoritmo com Imagens Submarinas

A seguir, analisaremos o desempenho do algoritmo em cinco imagens submarinas fornecidas para este fim. As imagens 1 e 2 já foram previamente ilustradas nas Figs. 1 e 2. As imagens 3, 4 e 5 são mostradas nas Figs. 96 a 98.

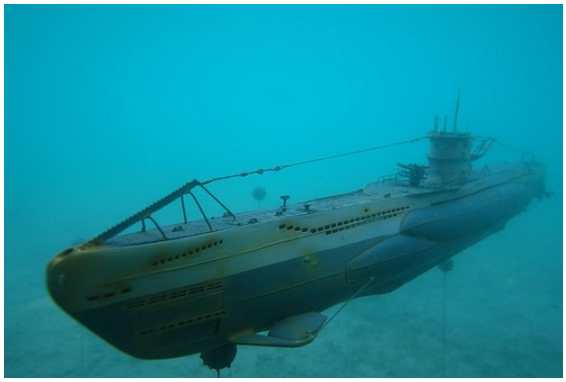


Fig.96 – Imagem 3.

Fig.97 – Imagem 4.

Fig.98 – Imagem 5.

Em cada caso serão exibidos os melhores resultados para a imagem em questão.

Na Fig. 96 o melhor desempenho foi utilizando o tipo CV_THRES_BINARY. Com *threshold1* = 85 e *threshold2* = 85, como podemos observar na Fig. 99, a imagem delimita a borda mas também seleciona outra parte do solo.

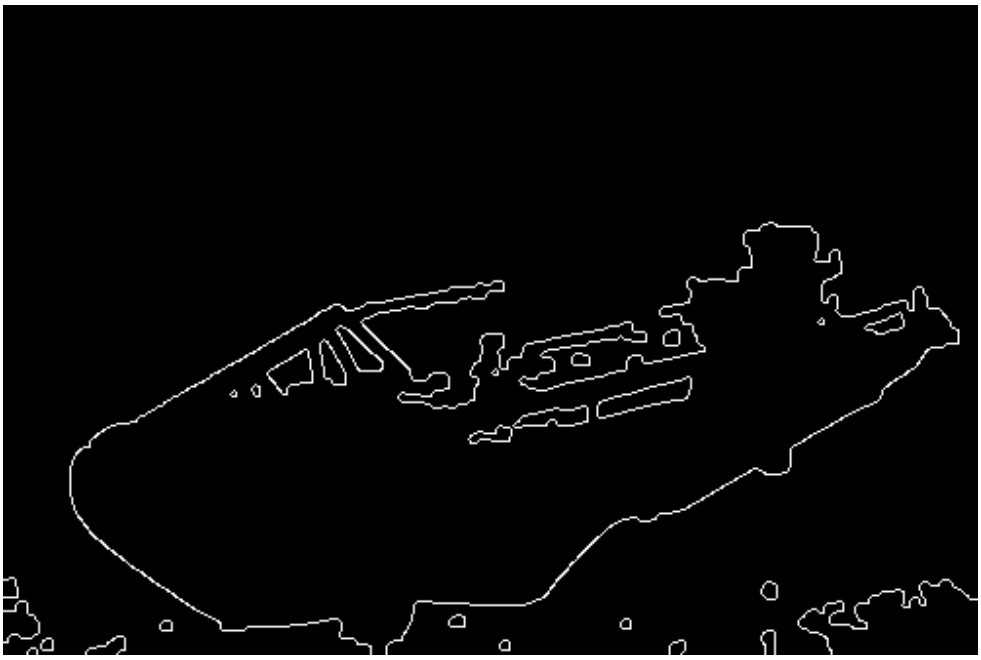


Fig.99 – Resultado da imagem 1 processada pelo programa de detecção de bordas utilizando CV_THRES_BINARY, thres1 = 85 e thres2 = 85.

Na Fig. 97, o melhor desempenho foi utilizando o tipo CV_THRES_TO_ZERO. Com *threshold1* = 65 e *threshold2* = 85, como podemos observar na Fig. 100, a imagem não consegue delimitar totalmente a borda e apresenta ruídos.

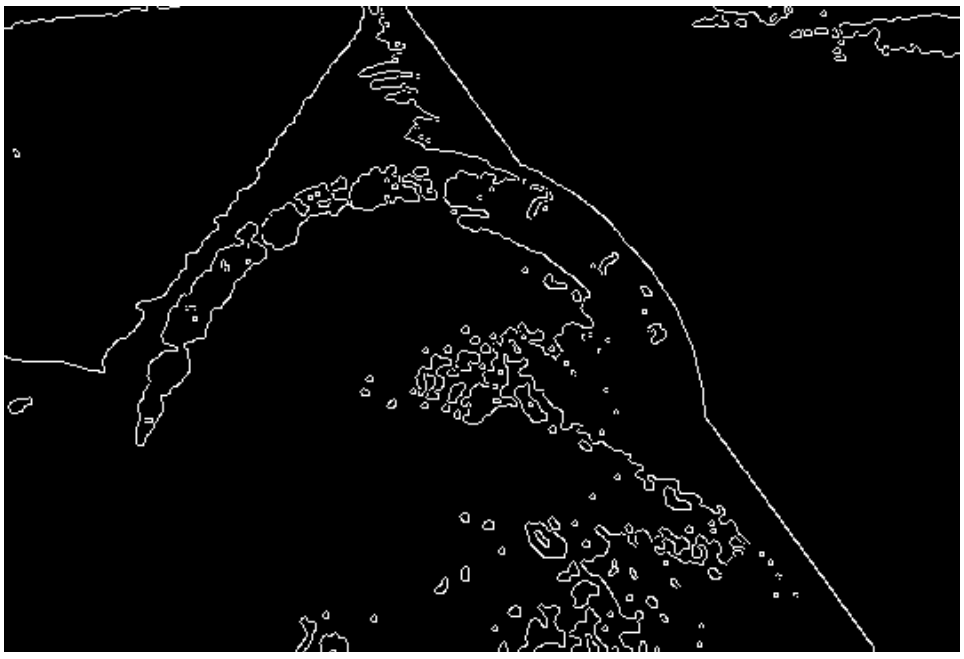


Fig.100 – Resultado da imagem 2 processada pelo programa de detecção de bordas utilizando CV_THRES_TO_ZERO, thres1 = 65 e thres2 = 85.

Com a imagem da Fig. 2, o melhor desempenho foi obtido utilizando o tipo CV_THRES_TO_ZERO_INV, com *threshold1* = 60 e *threshold2* = 65, como podemos

observar na Fig. 101. A imagem não consegue delimitar completamente as bolhas e ao usar o método *threshold to zero invertido*, fez com que as letras não fossem eliminadas.

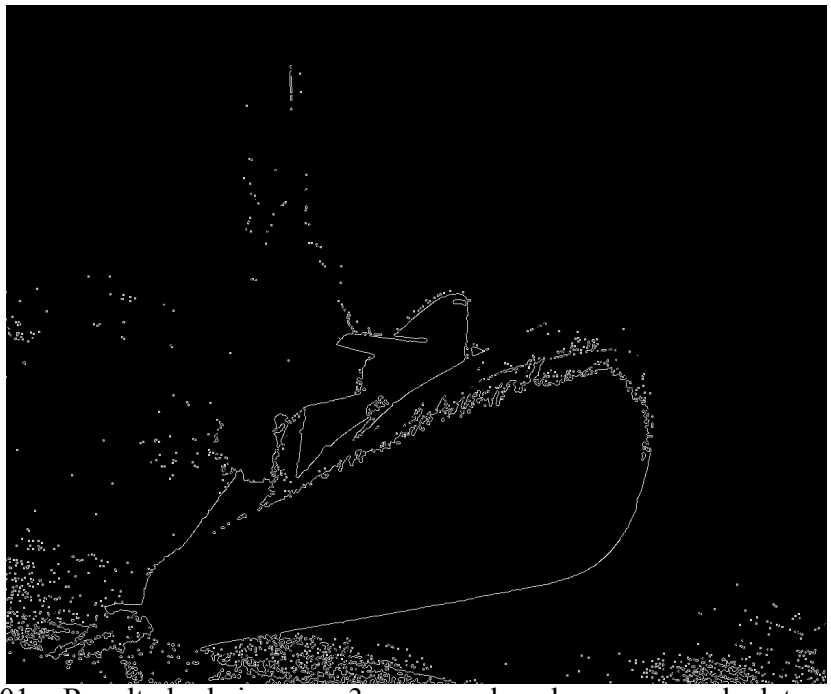


Fig.101 – Resultado da imagem 3 processada pelo programa de detecção de bordas utilizando CV_THRES_TO_ZERO_INV, thres1 = 60 e thres2 = 65.

Com a imagem da Fig. 1, o melhor desempenho foi obtido utilizando o tipo CV_THRES_BINARY_INV, com *threshold1* = 70 e *threshold2* = 65, como podemos observar na Fig. 102. A imagem delimita a borda, entretanto exibe ruídos no objeto. Ao utilizar-se de métodos invertidos de *threshold*, é normal a eliminação parcial das letras.

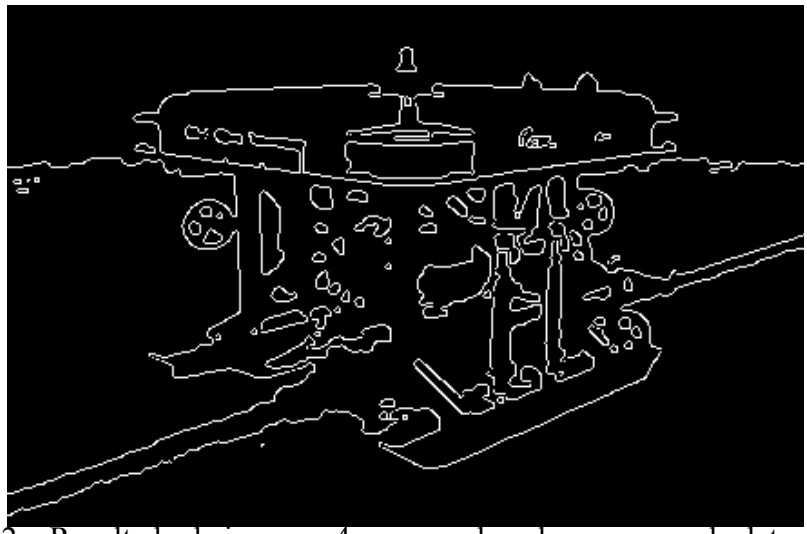


Fig.102 – Resultado da imagem 4 processada pelo programa de detecção de bordas utilizando CV_THRES_BINARY_INV, thres1 = 70 e thres2 =65

Para a imagem da Fig. 98, observamos, que a imagem resultante do método *threshold truncado*, associado à uma outra função – *cvHoughLines2*, é potencialmente

interessante para detecção de bordas. Por tratarmos de canos, a aproximação dos pontos a linhas é extremamente razoável. Nota-se que a imagem gerada não possui pontos, mas pequenos conjuntos de pixels alinhados no sentido das bordas. Sendo assim, antes de executarmos a função *cvHoughLines2*, a imagem é pré-processada por *cvDilate* duas vezes e uma vez por *cvErode*, visando a formação de pontos sólidos e posteriormente reduzidos por *erode* para um reconhecimento melhor da imagem ao traçarem as linhas de Hough. O método usado foi `CV_HOUGH_PROBABILISTIC`.

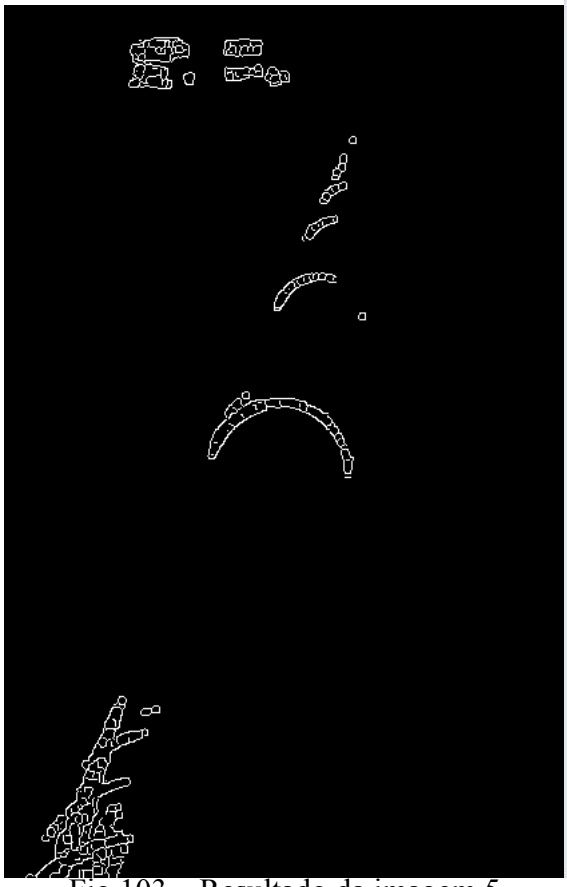


Fig.103 – Resultado da imagem 5 processada pelo programa de detecção de bordas utilizando CV_THRES_TRUNC, com thres1 = 6 e thres2 =20.

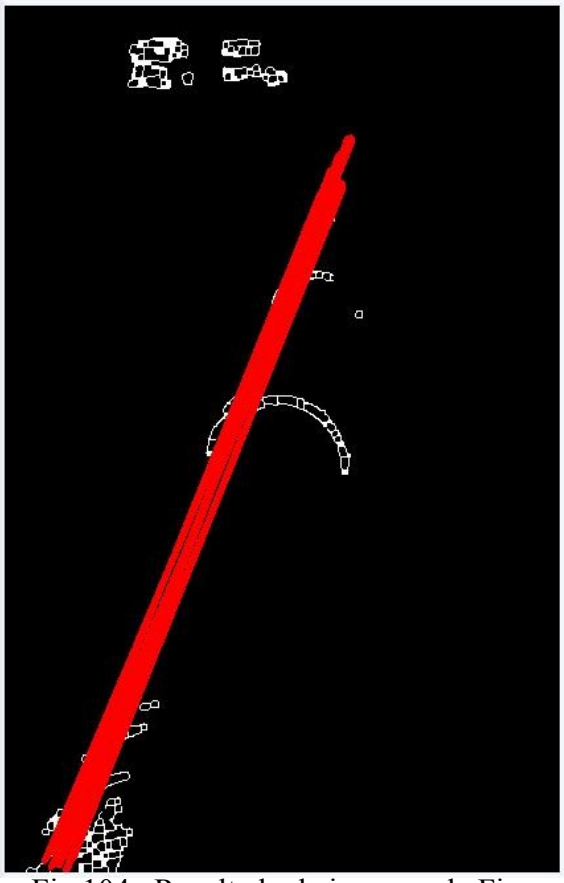


Fig.104– Resultado da imagem da Fig. 107 processada por cv HoughLines2 utilizando o método CV_HOUGH_PROBABILISTIC.

Esta associação mostra-se como uma possível alternativa às limitações em definir todos os pontos de uma ou das duas bordas de um cano.

Capítulo 4

Conclusão

O OpenCV não possui a mesma quantidade de funções implementadas que o MATLAB. Entretanto, é uma biblioteca recente, multiplataforma, e aberta – está em constante desenvolvimento e é utilizada em renomados centros científicos tanto para processamento de imagens, quanto para inteligência artificial e operações matemáticas. Entre estes centros, destacam-se o *Stanford Artificial Intelligence Lab* e o *MIT Media Lab*, a CMU, e Cambridge. No Brasil, temos um grupo de processamento de imagem do LABIEM da UTFPR e o TecGraf da PUC-RJ. Um grupo que merece destaque é o Willow Garage, formado por cientistas especializados em software para robótica, e que atualmente é responsável pelos maiores investimentos no desenvolvimento desta biblioteca, alinhando interesses acadêmicos e industriais, projetando softwares abertos.

O OpenCV possui uma vantagem em relação ao MATLAB – o processamento é mais rápido. Seu propósito é funcionar em tempo real, por isso está associado a vídeos, inteligência artificial, e reconhecimento de objetos. Desta forma, com suas funções básicas bem estruturadas, é possível reproduzir, a partir de um domínio mais profundo de algoritmos de processamento de imagens, funções complexas, utilizando-se da propriedade de processamento rápido. Na verdade, foi assim que ocorreu seu desenvolvimento em 2000, no MIT Media Lab. Os códigos abertos eram reaproveitados, aumentando o grau de complexidade das funções a partir da apropriação do conhecimento compartilhado.

De acordo com estudos, além de um conjunto de funções básicas de processamento de imagens, a biblioteca possui diversas funções matemáticas (cxcore.h), como convolução e integral, que permitiriam o desenvolvimento de um código específico, como é o caso do projeto da COPPE. Neste estudo, o foco foi a análise de funções diretamente associadas ao algoritmo proposto, devido ao grande volume de funções da biblioteca. Além disto, este software apresenta algumas funções implementadas que facilitariam o desenvolvimento e até proporcionariam uma solução alternativa. A função cvHoughLines2 possui um potencial a ser explorado em códigos diferentes, visando a detecção de bordas onde os algoritmos tradicionais não conseguem

reconhecer ou a borda não existe visivelmente na imagem original. Referente à aprendizagem de máquinas e ao reconhecimento de objetos, a função novamente se destaca, não só por funções complexas implementadas, mas por seu desempenho, permitindo aplicações em tempo real.

Finalmente, creio ser extremante proveitoso o uso desta biblioteca para o desenvolvimento dos códigos a serem embarcados no submarino não tripulado. Além de ser uma alternativa implementável na linguagem C, possui um desenvolvimento constante em grandes centros. É aberta, e pode ser uma forma de agregar a acadêmicos conhecimentos profundos desta ferramenta para que a UFRJ venha a ampliar sua atuação como centro de desenvolvimento de *softwares* de processamento de imagem e aprendizado de máquinas. A linguagem unificada pode ser um passo significativo para integração e desenvolvimento conjunto.

Bibliografia

[1] http://www.tecgraf.puc-rio.br/~malf/OpenCV/

[2] BRADSKI, G., KAEHLER, A., Learning OpenCV, Califórnia – E.U.A., O'Reilly Media, Inc., 2008.

[3] HARRIS, C., STEPHENS, M., "A combined corner and edge detector," Proceedings of the 4th Alvey Vision Conference (pp. 147–151), 1988.

[4]http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/MANDUCHI1/Bilateral_Filtering.html

[5] ALBUQUERQUE, M., ALBUQUERQUE, M., Processamento de Imagens: Métodos e Análises, Rio de Janeiro - Brasil., Centro Brasileiro de Pesquisas Físicas – CBPF/MCT - Coordenação de Atividades Técnicas - CAT

[6] http://www.willowgarage.com/

[7] INTEL CORPORATION, OpenCV Reference Manual, E.U.A., Copyright© 1999-2001 Intel Corporation All Rights Reserved Issued in U.S.A. Order Number: 123456-001

[8] J. CANNY, "A computational approach to edge detection," IEEE Transactions on Pattern Analysis and Machine Intelligence 8 (1986): 679–714.