DIEGO CAMILO FERNANDES

Sistema para detecção de desalinhamento de viga-guia em regiões de mudança de via dos Sistemas Monotrilho - Estudo e proposição do algoritmo de visão computacional

São Paulo
2016

DIEGO CAMILO FERNANDES

Sistema para detecção de desalinhamento de viga-guia em regiões de mudança de via dos Sistemas Monotrilho - Estudo e proposição do algoritmo de visão computacional

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Especialista em Tecnologia Metroferroviária.

São Paulo
2016

FOLHA DE APROVAÇÃO

DIEGO CAMILO FERNANDES

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Especialista em Tecnologia Metro-Ferroviária.

Área de Concentração:
Engenharia de Computação

Aprovado em:

Banca Examinadora

Prof. Dra. Anna Helena Reali Costa

Universidade de São Paulo **Assinatura:**

Prof. Dr. Jorge Rady de Almeida Junior

Universidade de São Paulo **Assinatura:**

Prof. Dr. Paulo Sérgio Cugnasca

Universidade de São Paulo **Assinatura:**

DIEGO CAMILO FERNANDES

Sistema para detecção de desalinhamento de viga-guia em regiões de mudança de via dos Sistemas Monotrilho - Estudo e proposição do algoritmo de visão computacional

Monografia apresentada à Escola Politécnica da Universidade de São Paulo para obtenção do título de Especialista em Tecnologia Metroferroviária.

Área de Concentração:
Tecnologia Metroferroviária

Orientador: Prof. Dra.
Anna Helena Reali Costa

São Paulo
2016

Catalogação-na-publicação

Fernandes, Diego Camilo
Sistema para detecção de desalinhamento de viga-guia em regiões de mudança de via dos Sistemas Monotrilho - Estudo e Proposição do Algoritmo de Visão Computacional / D. C. Fernandes -- São Paulo, 2016.
74 p.

Monografia (Especialização em Tecnologia Metroferroviária) - Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia.

1.Visão Computacional 2.Algoritmo 3.Open CV I.Universidade de São Paulo. Escola Politécnica. PECE – Programa de Educação Continuada em Engenharia II.t.

AGRADECIMENTOS

À Gerência de Projeto e Concepção de Sistemas – GCS – e ao Engenheiro Rubens Borloni, por demonstrarem confiança em minha competência profissional e ao apoio incondicional concedido para o cumprimento desta trajetória com longanimidade.

À minha orientadora, professora Livre-Docente Anna Reali, a quem Deus reservou a arte do conhecimento com a abundância de ensinar.

Aos colegas desta jornada no PECE: Fabio Bernardes e Ricardo Santos.

Aos Engenheiros da minha equipe de trabalho no Metrô de São Paulo: Fabio, Lucas, Mário G., Mário M., Natanael e Ruiz.

E a todos que, direta ou indiretamente, colaboraram na execução deste trabalho.

Simplicidade é a maior sofisticação.

(Leonardo da Vinci)

RESUMO

O objetivo deste trabalho é o estudo e a proposição do algoritmo de visão computacional capaz de mitigar o perigo do desalinhamento da viga-guia em regiões de mudança de via dos Sistemas Monotrilho em cenários envolvendo movimentações de trem em modalidade de condução manual nestas regiões. Serão apresentadas as premissas que conduziram à definição do código de computador e as funções da biblioteca para visão computacional OpenCV aplicadas durante o desenvolvimento do algoritmo de reconhecimento de desalinhamento da via, parte central do Sistema para Detecção de Desalinhamento de Viga-guia – SDDV – a qual se baseia no processamento das imagens obtidas por meio das câmeras de bordo dos trens das frotas do Monotrilho das Linhas 15 e 17 do Metrô de São Paulo.

ABSTRACT

The objective of this work is the study and proposal of computer vision algorithm can mitigate the risk of misalignment of the beam guide in the process of changing regions of monorail systems in scenarios involving train drives in manual driving mode in these regions.The premises will be presented that led to the computer code definition and functions of the library for computer vision OpenCV applied during the development of the track misalignment recognition algorithm, central part of the Sistema para Detecção de Desalinhamento de Viga-guia - SDDV - the which is based on processing of images taken by the onboard cameras of the trains Monorail fleet of lines 15 and 17 of the São Paulo Metro.

LISTAS DE FIGURAS

LISTAS DE TABELAS

LISTA DE SIGLAS E ABREVIATURAS

3D	Três Dimensões
AMV	Aparelho de Mudança de Via
API	*Application Programming Interface*
ASP	*Active Server Pages*
CPU	*Central Processing Unit*
DLL	*Dynamic Link Library*
FPS	*Frames per Second*
GUI	*Graphics User Interface*
GPU	*Graphics Processing Unit*
JPEG	*Joint Photographic Experts Group*
MATLAB	*Matrix Laboratory*
MPEG	*Moving Picture Experts Group*
OPENCV	*Open Computer Vision*
PIXEL	*Picture Element*
RANSAC	*Random Sample Consensus*
RGB	*Red, Green and Blue*
ROI	*Region of Interest*
SDDV	Sistema de Detecção de Desalinhamento de Via

SUMÁRIO

1. INTRODUÇÃO

Um dos principais riscos operacionais durante a operação de um sistema metroferroviário é a condução de um trem em modalidade manual através de uma região de mudança de via em condição de desalinhamento de rota, situação favorável para a ocorrência do descarrilamento da composição. Este risco é potencializado no modal de transporte Monotrilho, onde as regiões de mudança de via são elevadas – 15 metros de altura em média – e as consequências de um acidente desta natureza podem representar grandes prejuízos ambientais, patrimoniais e civis.

Para a mitigação deste risco foi proposto o Sistema de Detecção de Desalinhamento de Via – SDDV, cujo objetivo é detectar a viga do AMV – Aparelho de Mudança de Via – desalinhada (fora de posição) quando o trem estiver operando em modalidade manual, utilizando a câmera interna do trem para a obtenção de imagens que processadas por um algoritmo específico permitirá reconhecer e indicar a existência de desalinhamento na viga afim de, antecipadamente, disparar o processo que aplicará o freio de emergência a composição em movimento antes de adentrar na região de perigo.

Com o objetivo de analisar o desempenho do algoritmo de reconhecimento do sistema SDDV, em uma situação real de movimentação do trem, foi proposta a configuração de um ambiente computacional para verificação do seu funcionamento durante a reprodução de um vídeo real, obtido pela câmera de bordo do trem. Neste ambiente as ferramentas necessárias foram instaladas e a partir delas criada uma simulação da operação do processo real ao longo do tempo. Deste modo, a utilização de um simulador tornou-se a opção mais adequada para propiciar a verificação rápida de alternativas e soluções de *software* sem a necessidade da utilização do trem, exceto quando fosse necessária a obtenção de novos dados de entrada, vídeos capturados pela câmera de bordo, que salvos no registrador interno de eventos do trem, poderiam ser descarregados por meio de *pen-drive* quando da sua parada em uma Estação da linha principal ou via de estacionamento do pátio de manutenção, evitando distúrbios durante a operação comercial da linha. A Figura 1.1 apresenta o diagrama do simulador criado para testes do algoritmo do SDDV.

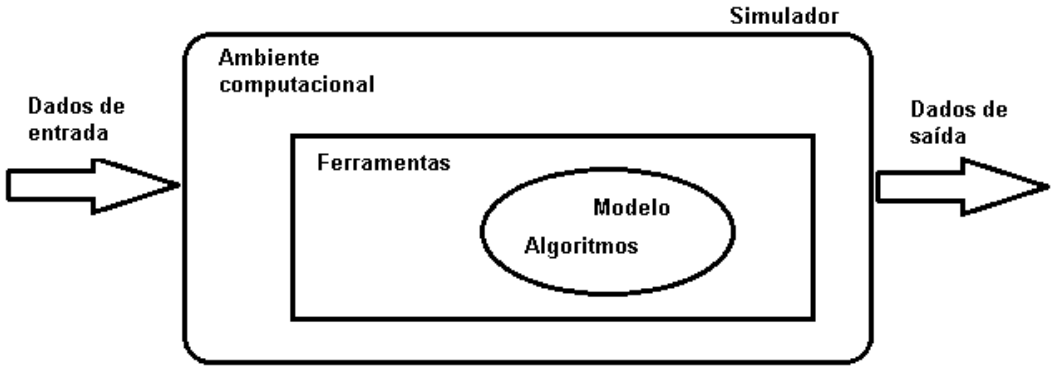


Figura 1.1 – Organização do simulador proposto para análise dos resultados. Os dados de entrada representam os vídeos obtidos por meio da câmera de bordo do trem e a execução dos algoritmos do SDDV geram dados de saída.

Os critérios definidos neste trabalho para a pesquisa e seleção do conjunto de ferramentas para composição do simulador foram:

- gratuidade: ausência de custos para aquisição e instalação dos softwares;
- suporte: oferta de documentação completa e de qualidade para instalação e utilização;
- diversidade de funções: disponibilidade de funções específicas que não fazem parte do escopo principal do estudo mas fornecem o suporte necessário para acelerar o desenvolvimento dos algoritmos e os seus meios para avaliação;
- interface gráfica: possibilitar a criação de uma interface gráfica suficiente para o manuseio dos dados, configuração dos parâmetros e visualização dos resultados da simulação dos algoritmos.
- portabilidade: migrar entre sistemas operacionais e plataformas de hardware embarcado sem impactar significativamente em mudanças no código desenvolvido, foco no desenvolvimento de um produto final;

Após extensa pesquisa, segundo os critérios apresentados, foi definido como biblioteca de *software* a ferramenta OpenCV. O OpenCV – *Open Source Computer Vision* é uma biblioteca de *software* para aplicações em visão computacional e aprendizado de máquina *open source*, ou seja, biblioteca cujo modelo de desenvolvimento promove o acesso universal por meio de código-fonte aberto permitindo projetar ou modelar um produto e redistribuí-lo, incorporando as melhorias

feitas por qualquer indivíduo. Lançado oficialmente em 1999, o projeto OpenCV era uma iniciativa da Intel® Research para o avanço em aplicações de uso intensivo da sua CPU. A partir da sua popularização, o OpenCV foi aperfeiçoado para fornecer uma infra-estrutura comum para aplicações de visão computacional e acelerar o uso da percepção de máquina em produtos comerciais. Esta biblioteca possuiu em sua versão atual mais de 2500 algoritmos otimizados, que inclui um conjunto abrangente de algoritmos clássicos e do estado da arte para visão computacional e aprendizado de máquina além de funções para criação e manipulação de objetos gráficos para construção de uma GUI – *Graphic Interface Unit* – completa. Os algoritmos fornecidos na biblioteca possibilitam a criação de aplicações como detectar e reconhecer rostos, identificar objetos, classificar as ações humanas em vídeos, interpretar movimentos de câmera, extrair modelos 3D de objetos, etc. Além da característica *open source* outras características como opções para o acesso á sua API – *Application Program Interface* – por meio de várias linguagens de programação como C++, C, Python, Java e MATLAB® *Toolbox* e compatibilidade de funcionamento com vários sistemas operacionais como Windows®, Linux, Android e MacOS®, contribuíram para a definição desta ferramenta como biblioteca básica de funções de visão computacional para o suporte ao desenvolvimento deste trabalho.

1.1. O desafio da visão computacional

A visão computacional compreende na transformação de dados de entrada, originários de uma câmara fotográfica ou de vídeo, em alguma decisão ou uma nova representação. Todas essas transformações são feitas para alcançar algum objetivo em particular. Os dados de entrada, por exemplo, podem incluir algumas informações contextuais tais como " a câmera está montada no trem " ou " o trem está a um metro de distância do obstácula à frente". A decisão por ourto lado pode ser " há chuva na via “ou " há viga está desalinhada ". A nova representação pode significar transformar uma imagem colorida em uma imagem em tons de cinza ou a remoção de movimento de câmera de uma sequência de imagens.

O ser humano foi concebido para ser capaz de realizar tarefas que envolvam a percepção, interpretação e tomada de decisões a partir de estímulos visuais de forma simples. Se entretanto, tentarmos criar uma analogia ao pensar que um computador

pode ser capaz também de realizar as mesmas tarefas da mesma forma simples, nos precipitaremos.

O cérebro humano divide o sinal de visão em muitos canais que reproduzem diferentes tipos de informação. Nosso cérebro possui um sistema de alerta, construído de uma forma dependente de tarefas, onde partes importantes de imagem são examinidas enquanto outras são suprimidas. Há realimentação maciça na corrente visual que é, por enquanto, pouco compreendido. Existem entradas associativas a partir de sensores de controle muscular e todos os outros sentidos, que permitem que o cérebro estabeleça associações cruzadas, possíveis de serem feitas a partir dos primeiros anos de vida de uma criança. Todos estes laços são realimentados dentro do cerébro e voltam para todas as fases de transformação, incluindo os próprios olhos, que controlam mecanicamente a iluminação externa captada pela íris, sintonizando a recepção de feixes por meio da superfície da retina.

Em um sistema de visão de máquina, no entanto, um computador recebe uma matriz de números provenientes de uma câmera ou de uma memória não volátil, e nada mais. Na grande maioria das vezes, não há reconhecimento de padrões embutidos, nem controle automático de foco e abertura e tampouco há associações cruzadas com base no conhecimento, anos de vida. O que o computador "enxerga" é apenas uma matriz de números. Qualquer número dentro dessa matriz pode possuir uma grande componente de ruído e por assim só nos conferir pouca precisão na informação. As ações ou decisões que o sistema de visão computacional tenta fazer com base em dados da câmera são executadas no contexto de um propósito ou tarefa específica. Nós podemos querer remover ruídos ou danos de uma imagem para que o nosso sistema de segurança, por exemplo, emita um alerta se o trem se movimentar em direção a uma região de desalinhamento na via ou porque precisamos de um sistema de monitoramento que conta quantas pessoas atravessam uma área em um parque de diversões. Independentemente da aplicação, vale como regra geral: quanto mais restrito for o contexto do sistema de visão computacional maior serão as chances de podemos contar com essas restrições para simplificar o problema e desta forma aumentar a confiabilidade do resultado final.

1.2. Origens do OpenCV

O OpenCV cresceu a partir de uma iniciativa da *Intel Research* para avançar em aplicações para uso intensivo da sua CPU. Um dos líderes desta pesquisa trabalhando pela Intel na época, visitou universidades e notou que alguns grupos universitários, como o *MIT Media Lab*, tinham desenvolvido internamente estruturas de código aberto para visão computacional que eram passadas de aluno para aluno e que isto conferia a cada novo aluno uma vantagem valiosa para o desenvolvimento da sua nova aplicação. Em vez de reinventar as funções básicas do zero, um novo estudante poderia começar seu desenvolvimento a partir da estrutura que havia sido criada anteriormente.

Desta ideia inicial o OpenCV foi concebido, nas linguagens de programação C e C++, como um caminho para tornar a infra-estrutura de visão computacional universal. Com a ajuda do *Intel's Performance Library Team*, o OpenCV começou com um núcleo de código implementado e especificações de algoritmos foram enviados para os membros desta equipe na Rússia. Foi lá onde o OpenCV começou a ganhar suas primeiras marcas de implementação: a experiência em otimização de códigos dos russos.

A partir de então o OpenCV recebeu muitas contribuições dos usuários, e o desenvolvimento do *core* da biblioteca deslocou-se em grande parte para fora da *Intel*. Ao longo de sua trajetória o OpenCV foi afetado pelo *boom* das empresas chamadas *dot-com* fazendo com que muitos dos seus pesquisadores interrompessem as contribuições e voltassem seus esforços para este mercado promissor. Durante estas flutuações, houve momentos em que o OpenCV não tinha ninguém da Intel dedicado à continuidade do seu desenvolvimento. No entanto, com o advento dos processadores *multicore* muitas novas aplicações para visão computacional tornaram-se viáveis e o desenvolvimento do OpenCV passou a ser valorizado novamente. Hoje, o OpenCV é área ativa do desenvolvimento em diversas instituições o que torna possível em curtos espaços de tempo muitas atualizações referentes a algoritmos para calibração multicamera, percepção de profundidade, métodos para a mistura de visão com *lasers* de alcance e localização, e melhores bibliotecas para reconhecimento de padrões, bem como um grande apoio para necessidades envolvendo o ramo da visão robótica.

1.3. Bibliotecas

De acordo com Bradski e Kaehler (2008), as funções do OpenCV estão distribuídas em bibliotecas que foram classificadas conforme suas características de aplicação. A Tabela 1.1 relaciona todas as bibliotecas e apresenta as descrições das aplicações.

Tabela 1.1 – Descrição das bibliotecas do OpenCV.

Biblioteca	Descrição da aplicação
Calib3d	Calibrar camera e reconstruir objetos a partir de múltiplas imagens planas
Core	Criar estruturas de dados, tipos e acessos para manipulação de memória
Features2d	Localizar e extrair características, descrições e objetos por semelhança em imagens
Flann	Utilizar funções de pesquisa rápida em espaços de grandes dimensões
GPU	Utilizar paralelização de algoritmos em GPU`s
Highgui	Criar e manipular GUI`s, imagens e vídeos
Imgproc	Processar imagens com filtragem, transformações geométricas, estruturas e análise de formas
ML	Aplicar modelos estatísticos e classificações
Nonfree	Utilizar algoritmos que estão patenteados em alguns países
Objdetect	Detectar objetos por meio de classificadores cascata e histogramas
Stitching	Aplicar mistura, distorção, costura e empacotamento em imagens
Video	Analisar movimento e objetos em vídeos

2. IDENTIFICAÇÃO DO DESALINHAMENTO DE VIGA

Uma vez que as etapas de modelagem do problema, conforme apresentado por Bernardes, Fernandes e Santos (2016), e configuração do ambiente computacional foram concluídas deu-se início a etapa de desenvolvimento do algoritmo para identificação do desalinhamento da viga. Com intuito de compreender melhor o problema a ser resolvido primeiramente foi selecionada a imagem de um vídeo, capturado durante a movimentação do trem na via, onde fosse possível encontrar o maior número de situações consideradas obstáculos para o processo de identificação de maneira a provocar uma discussão sobre quais abordagens poderiam ser aplicadas a fim de resolver o problema de forma otimizada.

Figura 2.1 – Imagem da via selecionada para avalição dos meios de solução do problema de localização da viga desalinhada.

A partir das informações visuais contidas na Figura 2.1 foi possível observar os seguintes fatos:

1. A maior concentração de informação a ser processada encontra-se numa região equivalente a 1/3 da altura da imagem, partindo de sua base;
2. Na altura da linha central da imagem se concentra a maior quantidade de informações, em sua maior parte, sem utilidade para o problema;

3. As passarelas de serviço, formas curvas de tonalidade cinza escuro, são estruturas que acompanham a viga durante o percurso da via e apresentam características geométricas muito semelhantes;
4. A viga apresenta situações de traçado curvo;
5. Existe um aumento gradativo de distorção na imagem à medida que deslocamos a visão partindo do ponto central, em direção radial, no sentido de qualquer ponto periférico. Uma consequência é o efeito abaulado da lateral do prédio branco à direita da imagem;
6. Presença do efeito de sombras sobre a viga;
7. A viga não necessariamente está posicionada no centro da linha de base horizontal da imagem. Em situações de curva como a da Figura 2.1 este centro pode estar deslocado para os lados.

Após analisar os sete pontos apresentados acima, criou-se a primeira proposta de fluxograma do algoritmo para solução do problema cuja validação seria feita incialmente com imagens estáticas, para posteriormente ser verificado em condição dinâmica onde o vídeo completo seria reproduzido e processado continuamente. O fluxograma proposto inicialmente está apresentado na Figura 2.2.

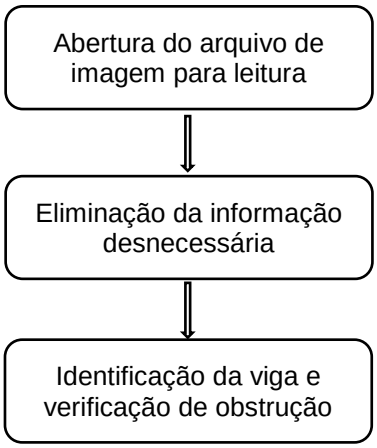


Figura 2.2 – Fluxograma inicial.

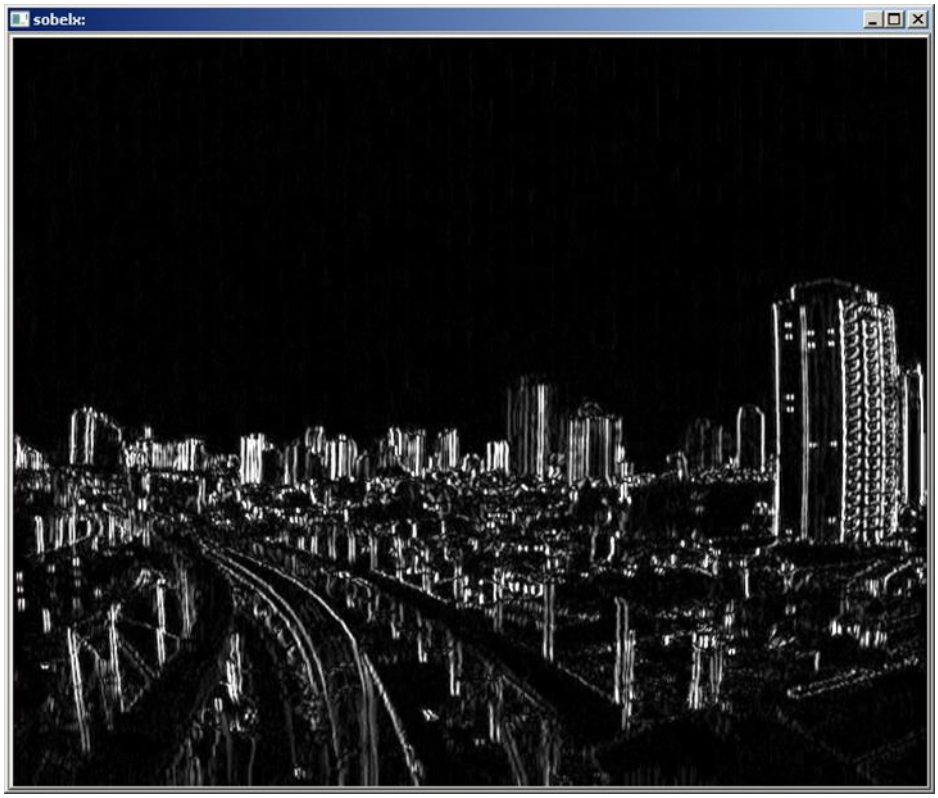
A solução para o primeiro bloco do fluxograma, “Abertura do arquivo de imagem para leitura” foi quase imediata, exigindo apenas o conhecimento da forma de utilização da função de manipulação de arquivos externos de imagem e as características da estrutura do tipo `IplImage` no OpenCV. Uma vez aberto e

acessado os dados do arquivo de imagem externa em formato JPEG, o próximo passo foi desenvolver os meios de eliminar da imagem todas as informações consideradas desnecessárias para o problema, portanto, todo o tipo de informação visual que não fosse a viga deveria ser descartado. Um paradigma foi criado, pois como seria possível eliminar toda a informação que não seja referente a viga se ainda sequer a reconhecemos na imagem. Deste modo, a ideia seguinte foi aplicar alguma técnica capaz de extrair os contornos da imagem, para então classificarmos o que seria a viga.

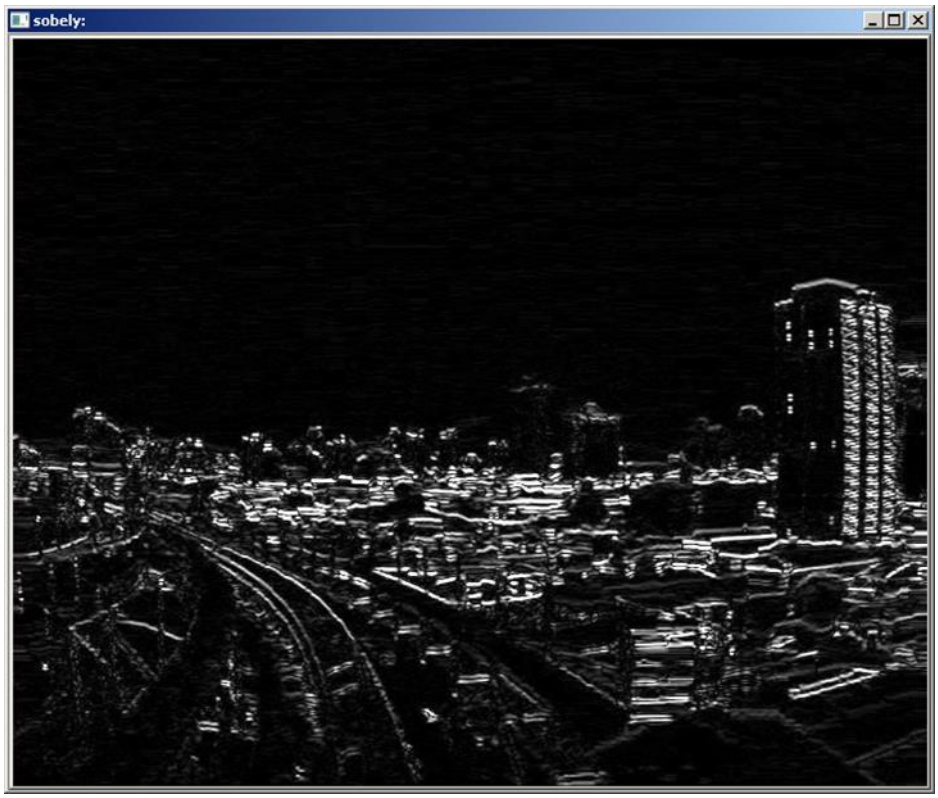
Bovik (2000) afirma que a detecção de bordas é uma importante ferramenta para análise de imagens em aplicações de visão computacional nas quais objetos devem ser reconhecidos por meio dos seus pontos periféricos. Normalmente, um objeto pode ser reconhecido a partir somente dos seus pontos periféricos (ATTNEAVE, 1954) sendo que experimentos com o sistema visual humano demonstram que os limites dos objetos em imagens são extremamente importantes e tendem a mostrar a intensidade das descontinuidades em uma imagem (BALLARD, 1982).

A biblioteca OpenCV oferece suporte à aplicação da técnica de detecção de bordas sob a forma de filtragem, onde o foco é enfatizar as bordas de um objeto por meio da análise do espectro da imagem sob o fato de na região das bordas naturalmente existir uma brusca variação de intensidade, o que caracteriza presença de componentes de alta frequência no espectro, ou sob a forma de operadores matemáticos, onde o foco é determinar bordas através da varredura de pixels em todas as direções da imagem aplicando um derivada para avaliar o grau de descontinuidade, ou “salto”, em relação ao valor da escala das cores nas vizinhanças do ponto de referência.

Neste trabalho optou-se por trabalhar com a detecção de bordas por meio dos operadores matemáticos e, com o suporte das funções `Sobel` e `Canny` do OpenCV, foram avaliadas as respostas obtidas após aplicação da técnica em duas imagens da via, sendo uma delas a mesma imagem da Figura 2.1. Os resultados da aplicação das duas funções nas duas imagens selecionadas são apresentados nas Figuras Figura 2.3 e Figura 2.4, para a função `Sobel` e Figura Figura 2.5 para a função `Canny`.

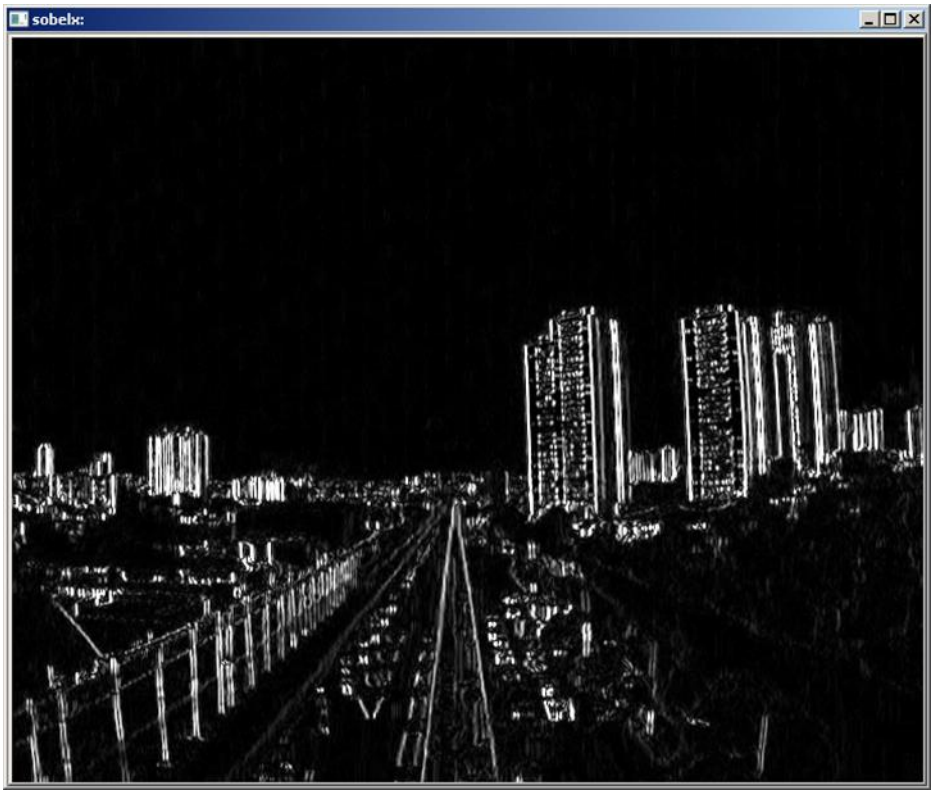


(a)

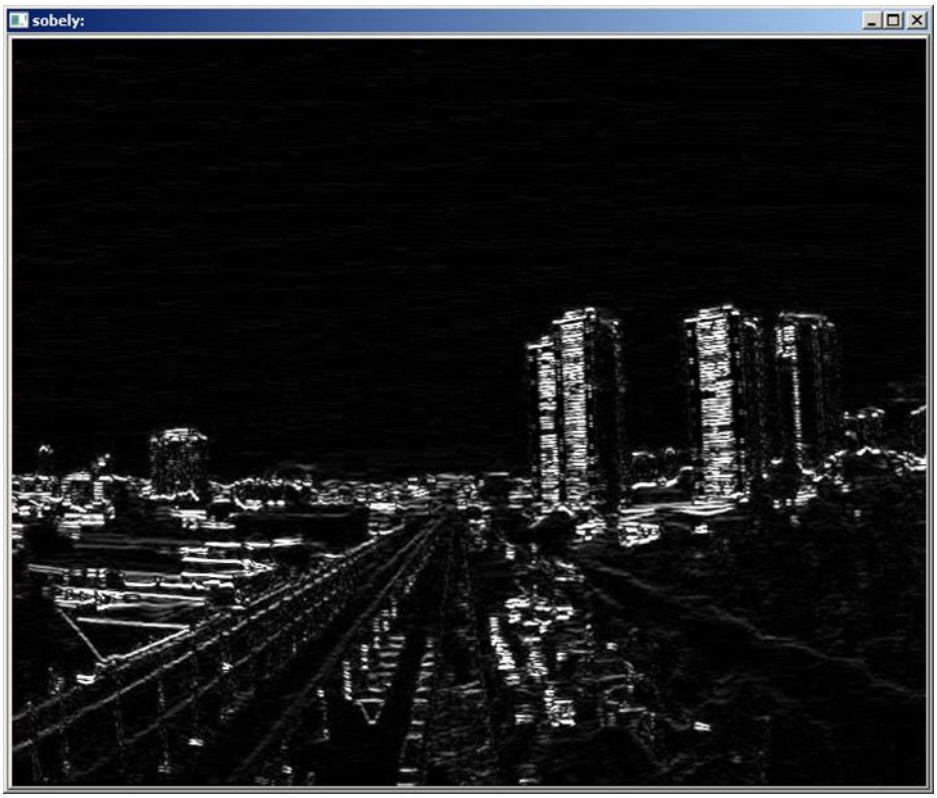


(b)

Figura 2.3 – Resultado da função `Sobel` na primeira imagem de avaliação, com gradiente (a) horizontal e (b) vertical.

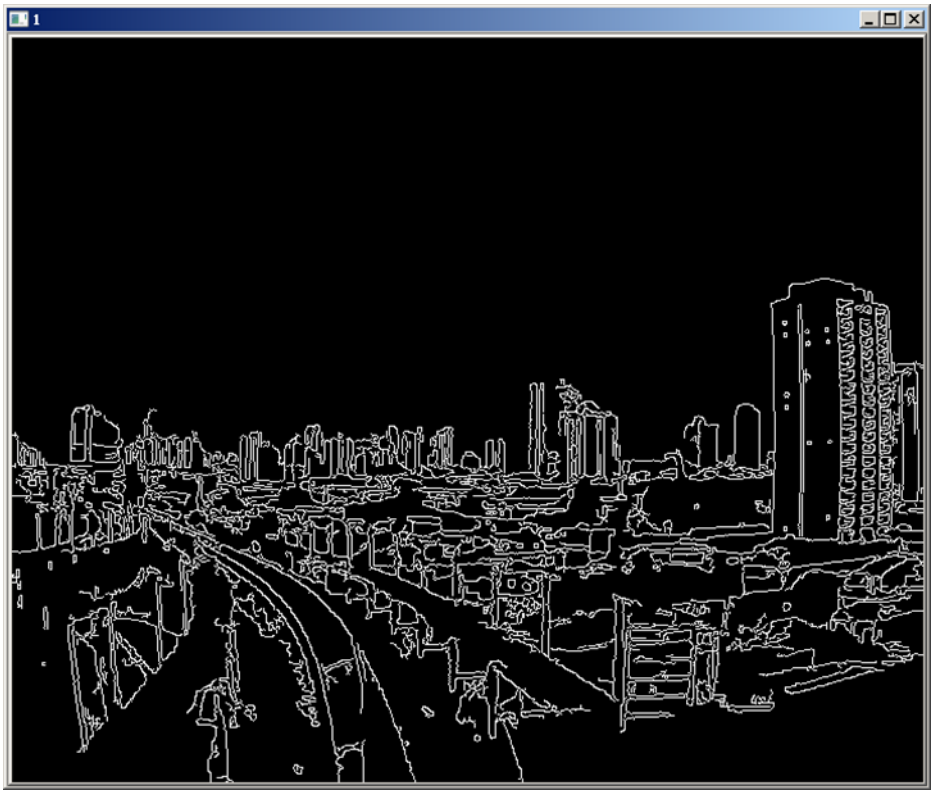


(a)

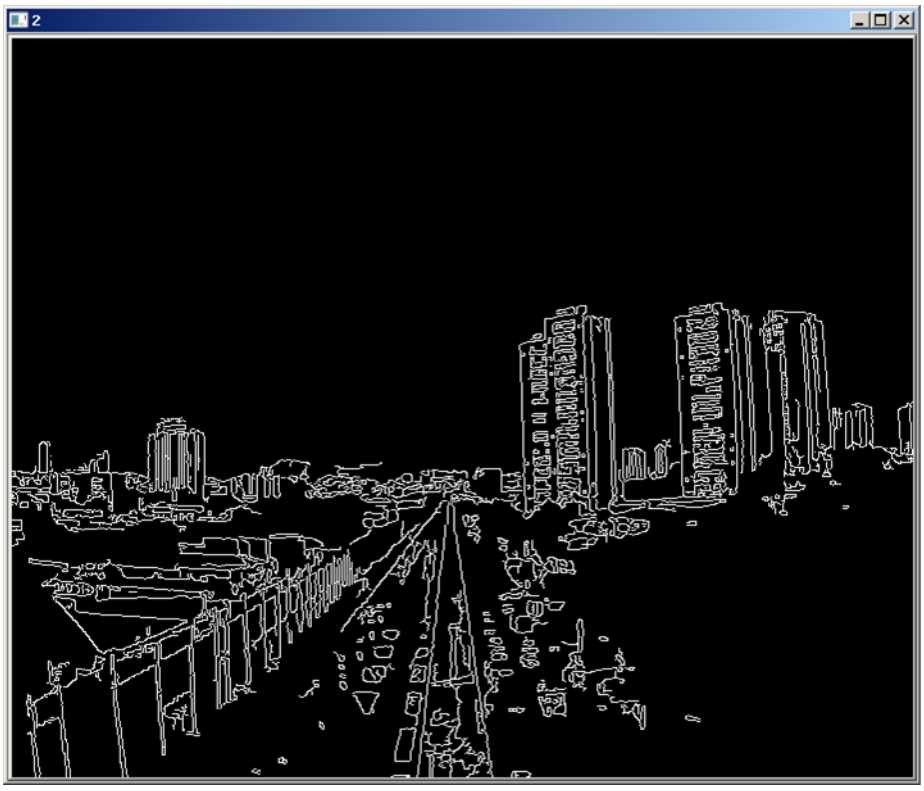


(b)

Figura 2.4 – Resultado da função `Sobel` na segunda imagem de avaliação, com gradiente (a) horizontal e (b) vertical.



(a)



(b)

Figura 2.5 – Resultado da função `Canny` na (a) primeira e (b) segunda imagem de avaliação.

Analisando os resultados é possível observar que a função `Sobel`, na condição de gradiente horizontal, foi mais eficiente que a função `Canny` na proposta de extrair as bordas da via, haja vista a quantidade de informação em excesso detectada no centro inferior da Figura 2.5 (b) se comparada à Figura 2.4 (a). Apesar desta característica, a função `Canny` foi a escolhida para extrair as bordas da via, pelo fato de apresentar um desempenho superior na detecção do trecho em curva, conforme demonstra a comparação entre as Figura 2.3 (a) e (b) em relação a Figura 2.5 (a). Foram realizados testes posteriores com diversas imagens submetidas à função `Canny` com o propósito de determinar o parâmetro de limiar de comparação mais otimizado, um dos parâmetros desta função. O valor 65 para o parâmetro de limiar se mostrou o mais equilibrado entre o compromisso de detectar a borda da viga e minimizar as demais transições de borda, ou ruídos, indesejados na imagem. Outro ponto importante a se observar é que o processo de detecção de bordas gera como resultado uma imagem "binarizada", ou preto e branco, onde os pixels resultantes do processo assumem apenas dois valores, 0, equivalente aos pixels representados na cor preta, e 255, equivalente aos pixels representados na cor branca. É preciso mencionar que as funções `Sobel` e `Canny` exigem como informação de entrada uma imagem em escala de cinza de 8 bits.

Levando-se em conta as observações apresentadas no início deste capítulo, outra providência foi incorporada ao algoritmo no sentido de otimizar a quantidade de informação a ser processada e ignorar aquilo que não fosse útil para a identificação da viga. Com isto foi criada uma ROI - região de interesse – com as dimensões de altura e largura configuradas da melhor forma para o problema. A Figura 2.6 apresenta a aplicação do conceito da ROI em relação à imagem da Figura 2.1.

A altura da ROI foi determinada por simples inspeção, observando-se que durante a execução dos vídeos de movimentação do trem cerca de 2/3 da altura de cada imagem, tomando como referência o seu topo, poderiam ser consideradas informações desprezíveis para o processo de detecção. Em relação à largura da ROI o processo de determinação desta dimensão exigiu mais critérios para sua definição: as possíveis posições da viga na imagem.

Observando o comportamento da sua posição durante a reprodução de alguns vídeos, foi possível determinar a largura média da ROI que fosse suficiente para considerar a curvatura máxima à direita, à esquerda e sua posição no centro da

imagem. O valor mais adequado para a janela foi definido como 3/5 da largura da imagem, posicionada a 2/5 do eixo *x* da imagem.

(a)

(b)

Figura 2.6 – Imagem detalhada (a) da ROI, com dimensão equivalente à 1/3 da altura e 3/5 da largura da imagem original, de 800 x 600 pixels; e (b) da sobreposição da ROI na imagem original, com as demarcações de suas subdivisões.
As subdivisões marcadas com 2, 3 e 5, somadas, equivalem a 3/5 da largura da imagem original. A altura das subdivisões marcadas com 2, 3 e 5 é 1/3 da altura da imagem original.

A ROI dentro no algoritmo é implementada por meio de um ponteiro para a região de memória que contém todos os valores da imagem original, executando o

endereçamento acrescentado um valor de *offset* compatível aos valores de largura e altura da região de interesse.

O próximo passo foi identificar uma técnica para análise de pontos das bordas a fim de reconhecer se os pontos identificados estão contidos em alguma forma geométrica conhecida. A Transformada de *Hough* é uma ferramenta que permite detectar relações geométrica entre pontos, originalmente desenvolvida para a detecção de retas. Outras variações desta transformada também permitem detectar formas cônicas como círculos, elipses e parábolas. O algoritmo proposto para implementação da transformada de *Hough* está transcrito na Figura 2.7.

```
transformada_hough()
      Inicializar matriz M(ρ,θ) com zero
      Para cada x,y pertencente à imagem com I(x,y) =1
            Para θ= -π/2+1 até θ=π/2 com incremento de Δθ
            ρ = x cosθ + y sinθ
            Incrementar de uma unidade a célula M(ρ,θ) da matriz acumuladora
            Faça laço até θ não pertencer mais ao intervalo (−π/2,π/2]
      Faça laço até o final da imagem.
```

Figura 2.7 – Algoritmo proposto para implementação da transformada de *Hough*.

A ideia da transformada de *Hough* consiste em representar uma reta por meio dos parâmetros polares *rho* e *theta* e então, substituir os valores de posição, x e y, de um ponto da borda nesta função polar, variando o ângulo *theta* desta reta em passos discretos, de modo a calcular todos os parâmetros das retas que cruzam este ponto. A cada reta que passar pelo ponto incrementa-se uma matriz acumuladora cujos índices são os parâmetros da reta, repetindo-se esse procedimento para os demais pontos. Ao final do procedimento a posição da matriz que apresentar a maior contagem será aquela cujos índices representam os parâmetros da reta que passa pela maior quantidade de pontos da imagem. Sucessivamente verifica-se a segunda posição da matriz com maior contagem, a terceira posição da matriz e assim por diante até o número desejado de retas a serem consideradas no processo de identificação. A Figura 2.8 apresenta a interpretação geométrica deste procedimento.

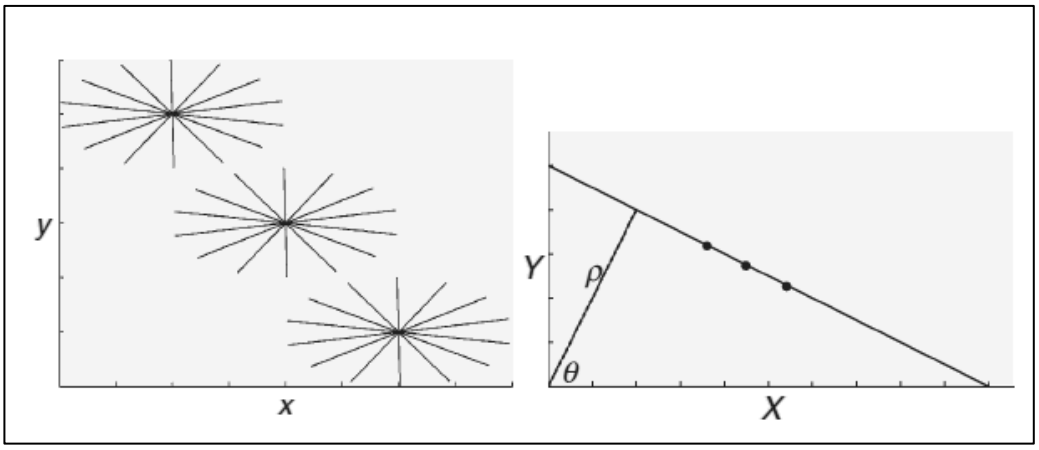
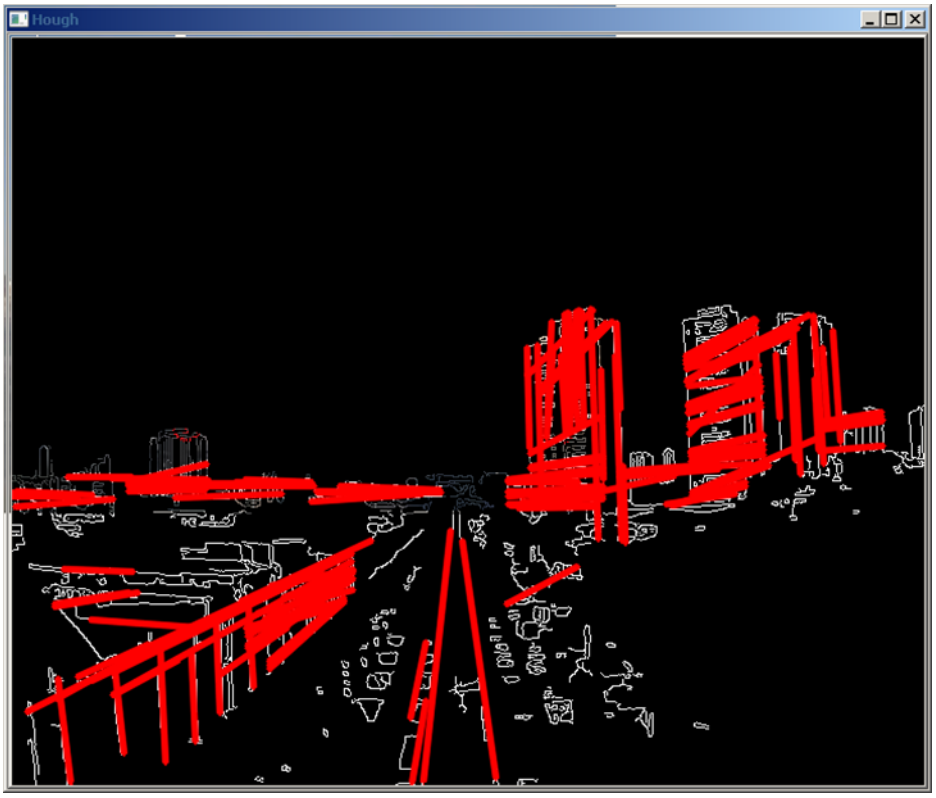


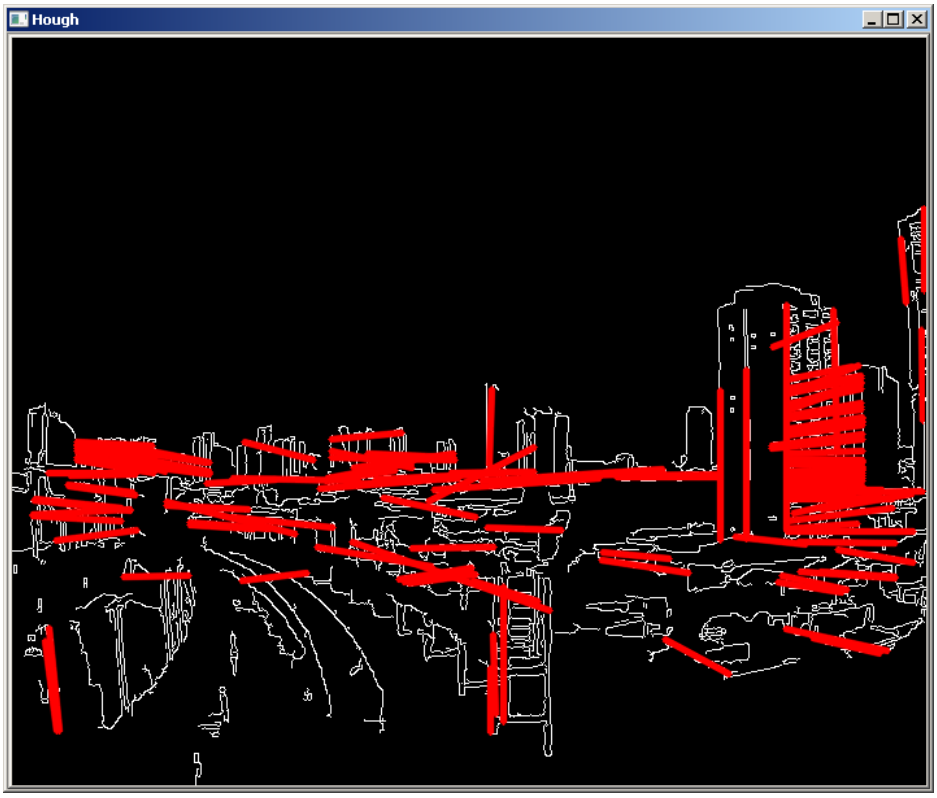
Figura 2.8 – Interpretação geométrica da Transformada de *Hough*.
Retas que passam por cada ponto, à esquerda, e a reta que torna todos os pontos colineares, à direita. A reta comum aos pontos será aquela cujo índice da matriz acumuladora apresentar o maior valor.

A implementação da transformada de *Hough* não possui a função de imprimir as retas identificadas na imagem. Com o objetivo de avaliar os resultados de forma gráfica na imagem foi necessário desenvolver um trecho de código adicional com a função de eleger os candidatos a reta a partir da matriz acumuladora $M(\rho,\theta)$, escrever as equações em funções dos parâmetros, aplicar pontos às funções e com o resultado na forma de coordenadas x e y da imagem, alterar a cor do pixel correspondente para a cor vermelha. Antes deste processo a imagem “binarizada” foi convertida para o espaço de cores RGB a fim de permitir a sobreposição das retas de forma colorida.

Os resultados apresentados na Figura 2.9 demonstram a capacidade da transformada em identificar retas na imagem. É possível concluir que para a primeira imagem de avaliação o fato da característica da forma da viga neste cenário ser equivalente a duas retas inclinadas tornou a eficiência do algoritmo alta do ponto de vista de detecção da sua forma, conforme apresentado na Figura 2.9 (a). Porém, à medida que o movimento do trem se aproxima de uma região da via cuja viga encontra o limite do seu raio de curvatura, como representado na segunda imagem de avaliação, a dificuldade de aproximar sua forma por uma reta aumenta drasticamente ao ponto de não ser possível identificá-la por meio da transformação implementada, conforme observado pela a ausência de retas sob a borda da viga na Figura 2.9 (b).



(a)



(b)

Figura 2.9 – Retas identificadas após aplicação da Transformada de *Hough* (a) na primeira e (b) na segunda imagem de avaliação.

A primeira tentativa para se contornar este problema foi criar uma variação da implementação da transformada de *Hough* que somente fosse aplicada sobre a imagem a partir do momento em que a transformada padrão, baseada em retas, não retornasse parâmetros de retas para a região da viga na imagem, sob a hipótese que neste cenário a representação da viga na imagem deixou de ser reta e passou a ser curva. Para a seleção da nova forma geométrica a ser incorporada na transformada de *Hough* foram tomados, a partir de imagens binarizadas da viga em curva, diversos valores de coordenadas x e y dos pontos sob suas bordas, repetindo-se este procedimento para outras imagens onde houvesse a variação do seu raio de curvatura. Toda esta massa de dados foi então submetida a procedimentos de ajustes de curvas e os ajustes obtidos foram então comparados. Após as comparações foi concluído que o melhor lugar geométrico para descrever os pontos da borda da curva seria uma parábola. Por definição a transformada de *Hough* opera em um espaço onde uma parábola necessariamente deve ser representada em função de seus parâmetros polares e, portanto, os passos a seguir apresentam a dedução para obtenção desta relação e a sua incorporação ao algoritmo da transformada.

A parábola, Figura 2.10, é uma curva plana, lugar geométrico dos pontos de um plano que são equidistantes de um ponto fixo F e de uma reta fixa k. No plano cartesiano, é definida como a curva plana formada pelos pontos $P(x, y)$, tais que:

$$p = PP' = PF,$$

onde p representa a distância entre os pontos P e F ou à distância entre o ponto P e a reta k (diretriz).

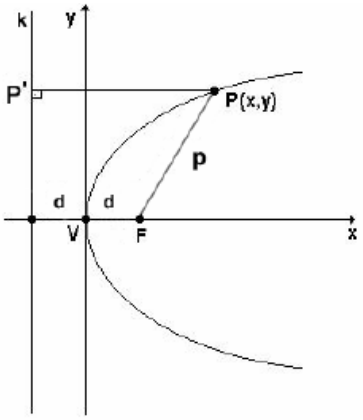


Figura 2.10 – Representação geométrica da parábola e seus parâmetros.

Outro parâmetro, $2d$, representa a distância do foco F à reta diretriz k. Esta forma cônica também pode ser definida como uma função polinomial do segundo grau do tipo $y = g.x2 + h.x + i$ *ou* $x = g.y2 + hy + i$, com $g \neq 0$. Sempre que o parâmetro $g > 0$, a parábola terá concavidade voltada para cima ou para direita, e quando $g < 0$ a parábola terá concavidade voltada para baixo ou para esquerda. No caso de uma equação reduzida da parábola de eixo horizontal e vértice na origem, consideremos os pontos: $F(d, 0)$ - foco da parábola, e $P(x, y)$ - um ponto qualquer da parábola. Usando a fórmula da distância entre pontos no plano cartesiano, obtemos:

$$[\,(x-d)^2 + (y-0)^2\,]^{1/2} = [\,(x+d)^2 + (y-y)^2\,]^{1/2}$$

Desenvolvendo e simplificando a expressão acima para parábolas de eixo horizontal e vértice em um ponto qualquer, do mesmo modo que se comporta a borda da viga, se o vértice da parábola não estiver na origem e, sim num ponto (x_0, y_0), a equação pode ser reescrita na forma:

$$(y - y_0)^2 = 4d\,(x - x_0).$$

Agora, observando a Figura 2.22 e efetuando projeções no eixo de referência x é possível estabelecer as seguintes relações:

$$\rho = \frac{2d}{1-\cos\beta}$$

$$x = \rho\cos\beta\,,\quad x = \rho sen\beta\,,$$

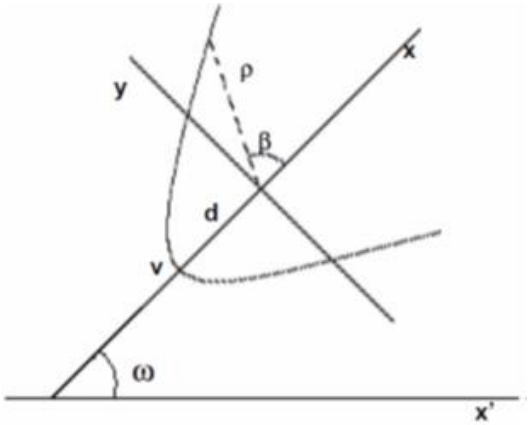


Figura 2.11 – Representação geométrica de uma parábola em função dos parâmetros polares ρ, β e ω.

onde d representa a distância do foco ao vértice, ρ representa a distância do foco a cada ponto da parábola, equivalente ao p da Figura 2.10, β representa o ângulo de ρ em relação ao eixo horizontal, e x e y são as coordenadas de um ponto qualquer da parábola.

Entretanto para a descrição completa, é necessário considerar as inclinações dos demais eixos e suas projeções:

$$x' = x_f + \rho\cos\beta\cos\omega - \rho\sin\beta\sin\omega$$
$$y' = y_f + \rho\cos\beta\sin\omega + \rho\sin\beta\cos\omega,$$

onde x' e y' são os pontos dos pixels da parábola em relação ao eixo da imagem e x_f e y_f os pontos de localização do vértice da parábola.

De posse das expressões de x', y' e ρ, o algoritmo para implementação da transformada de *Hough* foi adaptado para identificação de uma parábola, com a sua implementação final transcrita na Figura 2.12.

```
parabola_hough()
      Guardar todos os pixels num array de uma dimensão
      Limpar acumulador M(x0,y0,d, ω)
      Para pixel I(x,y), se este pixel I(x,y) estiver aceso na imagem
            Para d>mínimo até d<máximo
                  Para β=0 até β<2π
                  ρ = 2d 1- cosβ
                        Para ω=0 até ω<2π
                        x = x -ρ cosβ cosω + ρ sinβ sinω f
                        y = y -ρ cosβ sinω -ρ sinβ cosω f
                        M(x0, y0,d, ω)=M(x0, y0,d, ω)+1
                        Faça para os ângulos de β=0 à 2π
                  Faça para os ângulos de ω=0 à 2π
            Faça para todos os d´s de mínimo à máximo
      Faça para todos os pixels da imagem
```

Figura 2.12 – Algoritmo para implementação da Transformada de *Hough,* adaptado para identificação de uma parábola.

O funcionamento do algoritmo é idêntico àquele implementado para reta, porém a diferença aqui é que a matriz acumuladora possui um índice maior devido à quantidade de parâmetros da parábola, isto é, o tamanho necessário de memória para armazenar os parâmetros calculados aumentou oito vezes em relação ao que era necessário anteriormente.

Os resultados da Figura 2.24 indicam que, para a imagem contendo a região curva, foi possível aproximar as bordas da viga por duas parábolas. A parábola à esquerda não apresentou um ajuste muito adequado à borda da viga; por outro lado, a parábola à direita foi capaz de acompanhar com razoável aproximação o contorno.

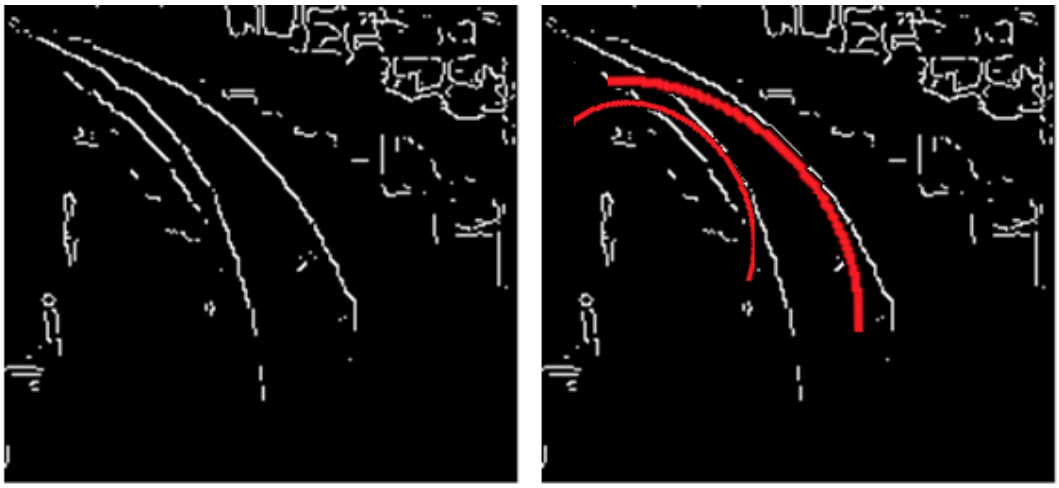


Figura 2.13 – Parábolas identificadas na segunda imagem de avaliação após aplicação da Transformada de *Hough* modificada. Em detalhe à esquerda a ROI da imagem original.

Em termos do tempo de processamento, esta implementação foi cerca de dez vezes mais lenta do que em relação à transformada utilizando a reta como forma de identificação, mesmo levando-se em conta a utilização da ROI para minimizar o número de cálculos com pontos da imagem sem representação para este problema. O tempo total despendido para estimar as parábolas neste cenário de distribuição de pontos na imagem foi cerca de 110 ms, aproximadamente 3 vezes mais que o intervalo de tempo entre quadros do vídeo de movimentação do trem, reproduzido a 30 fps ou 33 ms de intervalo entre quadros de imagens. Estes fatos motivaram a busca por outra forma alternativa para identificar a viga na imagem.

Outra linha de abordagem para a solução deste problema seria interpretar os pontos de borda da imagem como medidas diretas da forma que se deseja estimar, levando o problema para o campo da estimação. Estimar parâmetros significa, como o próprio nome diz, estimar coisas que não variam, são constantes ao longo do processo de estimação. Então, para começar a estimar algo, necessita-se de um conjunto de medidas, no contexto do problema pontos, que esteja relacionada a esse algo. O próximo passo é modelar como essas medidas se relacionam aos parâmetros a serem estimados.

Um dos estimadores de parâmetros utilizado pela comunidade científica é o algoritmo de mínimos quadrados (CHAPRA, 2008). Este procedimento, tão antigo quanto Gauss, que primeiro o formulou para processar observações astronômicas de

corpos celestes, formalmente, trata de minimizar uma função custo do quadrado dos resíduos na forma:

$$L = (y - Hx)^t (y - Hx) ,$$

onde y representa o vetor contendo m medidas, x representa o vetor de n parâmetros a serem estimados, e H representa uma matriz m x n que relaciona as medidas aos parâmetros.

Seja, por exemplo, o caso de ajustar uma reta aos dados através do método de mínimos quadrados, conforme ilustrado na Figura 2.14, como a situação onde os pontas da borda na viga, primeira figura de avaliação, são modeladas por duas retas inclinadas.

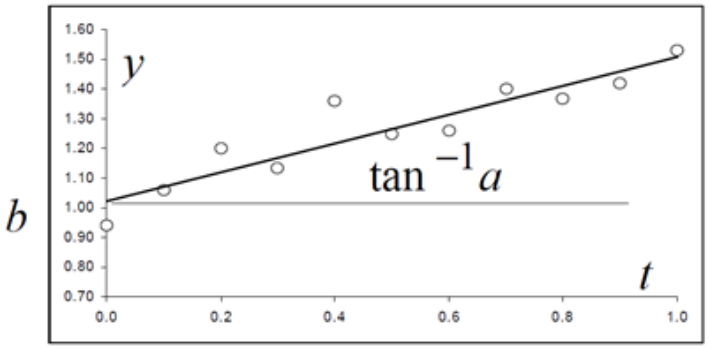


Figura 2.14 – Ajuste linear. Adaptado de Kuga (2005).

A equação genérica da reta é dada por $y_i = at_i + b$. Logo, a equação que relaciona as medidas aos parâmetros é formulada como:

$$y = Hx ,$$

ou, de modo explícito:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m1} \end{bmatrix} = \begin{bmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \\ t_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} ,$$

onde $x = (a, b)$ é o vetor que contém os parâmetros a serem estimados.

A princípio, a forma mais direta de processar essas medidas é o chamado processamento em lotes ou *batch*. A expressão geral desta implementação pode ser escrita na forma:

$$\hat{x} = (H^t H)^{-1} H^t y ,$$

onde $\hat{x}$ representa o vetor que contém os parâmetros estimados.

No processamento em lote é necessário que todas as medidas estejam disponíveis para que os parâmetros sejam estimados, o que apesar de possível no contexto deste trabalho não é interessante do ponto de vista computacional, uma vez que o algoritmo prenderá o fluxo de execução principal no ponto da rotina que executa este cálculo até que todas as medidas sejam processadas, podendo deixar de lado outras tarefas que, no âmbito global do *software,* exijam seu cumprimento dentro deste intervalo de tempo.

Em busca desta melhoria foi implementado o algoritmo de mínimos quadrados recursivo, que nada mais é do que uma forma algebricamente equivalente de processar as medidas. Outra vantagem desse algoritmo, aplicado à estimação de parâmetros, reside no fato de evitar inversões de matrizes. Diz-se recursivo por ter características de recursividade, portanto, bastante adequado para programação em computador. Outra vantagem é a de necessitar de matrizes de menor dimensão, traduzindo-se em menos memória de armazenamento. Basicamente, o algoritmo usa a forma de *Kalman* para o processamento (GREWAL; ANDREWS, 2015).

Inicialmente, particionam-se as matrizes envolvidas:

$$\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_{m1} \end{bmatrix} = \begin{bmatrix} t_1 & 1 \\ t_2 & 1 \\ \vdots & \\ t_m & 1 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} H_1 \\ H_2 \\ \vdots \\ H_m \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix},$$

onde os H_i são os vetores linha que compõem a matriz H. Em seguida, calcula-se:

$$P_2 = \left(\begin{bmatrix} H_1{}^t & H_2{}^t \end{bmatrix} \begin{bmatrix} H_1{}^t \\ H_2{}^t \end{bmatrix} \right)^{-1}$$

$$\hat{x}_2 = P_2 \begin{bmatrix} H_1{}^t & H_2{}^t \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \end{bmatrix}$$

Então, o algoritmo torna-se recursivo para $i = 3, \ldots, m$:

$$\begin{aligned} K_i &= P_{i-1} H_i{}^t \left(H_i P_{i-1} H_i{}^t + 1 \right)^{-1} \\ P_i &= \left(I - K_i H_i \right) P_{i-1} \\ \hat{x}_i &= \hat{x}_{i-1} + K_i \left(y_i - H_i \hat{x}_{i-1} \right) \end{aligned},$$

onde I representa a matriz identidade. O parâmetro K é conhecido como ganho de *Kalman* e P é a matriz de covariância. O novo fluxograma do algoritmo incorporando a aplicação do método de mínimos quadrados recursivos está apresentado na Figura 2.15.

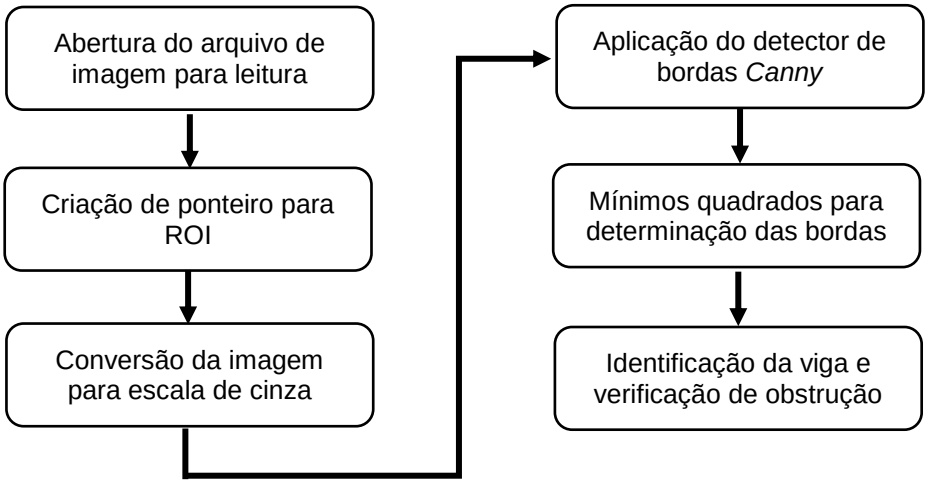
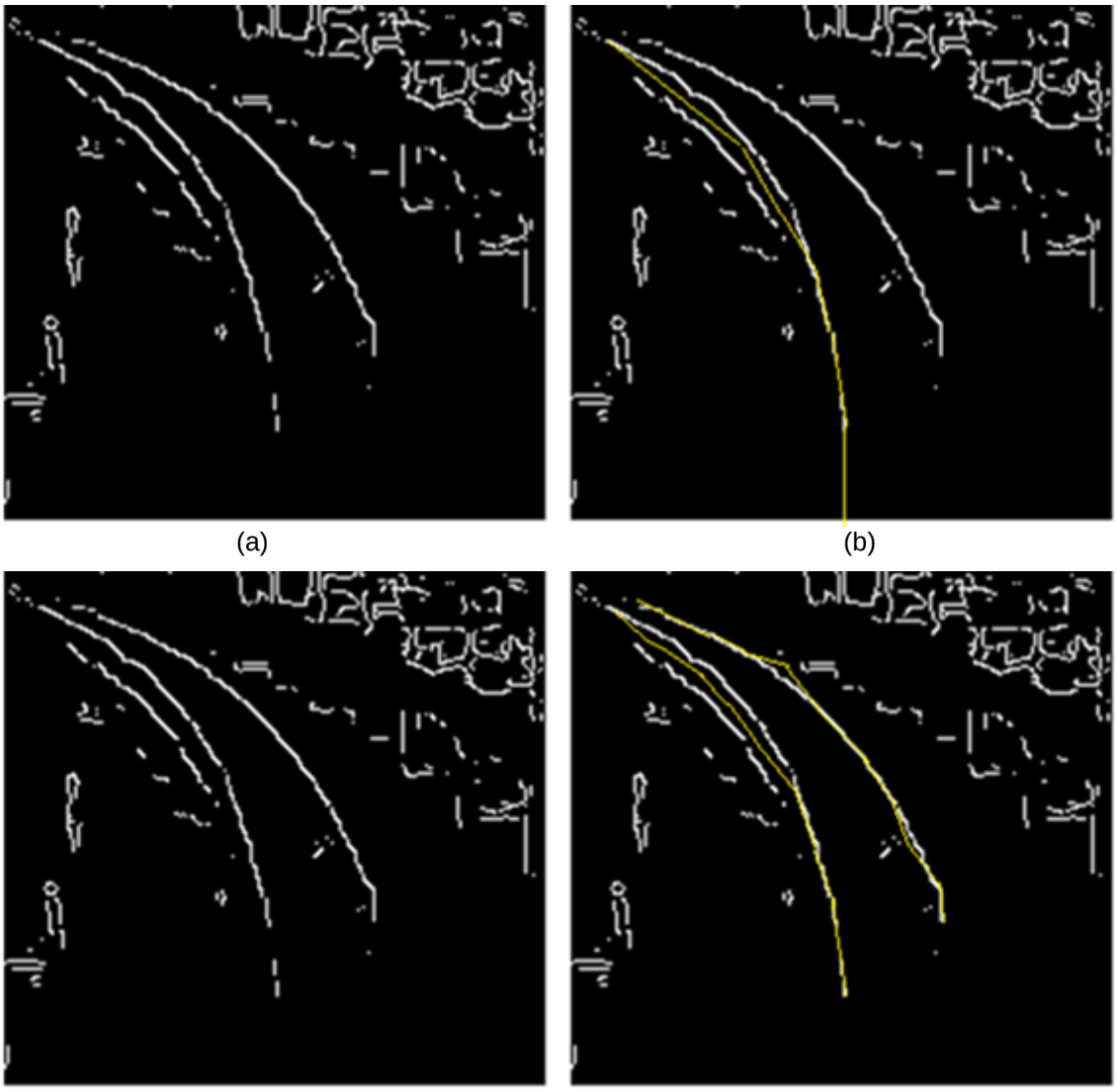


Figura 2.15 – Fluxograma atualizado para aplicação do algoritmo de mínimos quadrados recursivo.

O algoritmo precisou ser reescrito para incorporar as expressões dos mínimos quadrados recursivos. A estratégia para detectar retas de *Hough* foi substituída por esta estratégia que também passou a ser aplicada nas situações de detecção da viga em curva. Incialmente são criados cinco estimadores cada qual responsável por uma subdivisão da imagem, conforme apresentado na Figura 2.19. Cada estimador processa 20 pixels verticais numa faixa de 10 pixels à direita e a esquerda do ponto de início da busca, crescendo no sentido de baixo para cima da imagem, eixo y negativo. Ao término da busca os parâmetros são avaliados para verificar a característica de inclinação da reta e também a matriz de covariância resultante com o intuito de avaliar a qualidade do ajuste. Se as condições forem satisfeitas, valores baixos na matriz e coeficientes de retas com inclinações acima de 30º, a janela de medidas equivalente a 20 pixels escorrega para cima na vertical tendo como centro da base da nova janela o último ponto da reta estimada no processo anterior. O objetivo é que ao final, somente as janelas que possuírem pontos que possam ser estimados pela pequena reta de aproximação alcançarão o limite vertical da ROI. Neste instante, os segmentos de retas unidos pelos seus pontos extremos representarão o contorno da forma da viga.



(a) (b)

(c) (d)

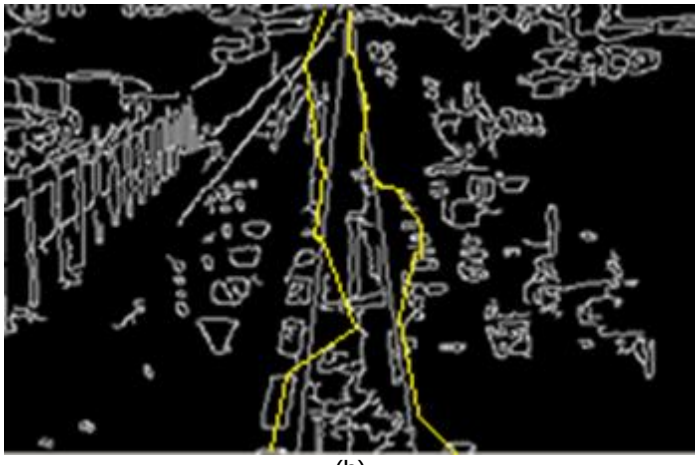
Figura 2.16 – Resultado da aplicação do algoritmo para identificação da curva por mínimos quadrados recursivo na segunda imagem de avaliação: curvas em amarelo em (b) e (d).

Na Figura 2.16 (b) uma avaliação da resposta do algoritmo, pixels em amarelo, mostra algumas limitações da sua implementação. Pelo fato da imagem ser submetida a detecção de bordas, muitos dos pontos da viga que deveriam existir na sua base não foram representados, devido ao efeito de sombras que atenuaram as variações de intensidade nesta região, deste modo, o algoritmo não foi capaz de receber os 20 primeiros pontos que forneceriam a orientação da reta que deveria acompanhar a curva à direita da imagem. Outro fator importante é que por se tratar de um estimador, caso existam muitos pontos na periferia do traçado real, existe uma tendência natural de o processo tentar encontrar um ajuste médio entre as curvas,

criando um efeito de descolamento em relação a curva da viga, como se observa na extremidade superior da curva à esquerda da imagem.

Para a obtenção da resposta da Figura 2.16 (d), forçou-se o algoritmo a começar com um offset de 35 pixels na vertical permitindo desta forma que a curva à direita da imagem fosse identificada. É possível observar durante o acompanhamento desta curva sobre a borda da viga que existem alguns picos, estes picos são causados pelos pontos periféricos que, uma vez observados pela janela de varredura do algoritmo, são considerados nas estimativas e acabam por assim causar o mesmo efeito ocorrido na Figura 2.16 (b).

(a)



(b)

Figura 2.17 – Em (a) a primeira imagem de avaliação, detalhe da ROI; em (b) o resultado da aplicação do algoritmo para identificação da curva por mínimos quadrados recursivo na primeira imagem de avaliação.

Na Figura 2.17 (Figura 2.17b), as influências de ruídos demonstram claramente a dificuldade do algoritmo em acompanhar a trajetória esperada, chegando ao ponto das curvas em amarelo quase se cruzarem na base da imagem.

Apesar dos resultados não serem satisfatórios, em termos de tempo de processamento esta solução se mostrou muito eficiente. Nos três casos estudados o tempo médio para finalizar o processo não ultrapassou 5 ms.

Após os testes com as transformadas de *Hough* e o método dos mínimos quadrados, optamos por avaliar a resposta das técnicas do tipo *match template* (BRADSKI; KAEHLER, 2008) para a execução da tarefa de identificar a viga na imagem.

Match template é uma técnica de processamento digital de imagens utilizada para encontrar em pequenas partes de uma imagem uma correspondência para uma imagem modelo. Para modelos sem características fortes, ou para quando a maior parte da imagem do modelo constitui a imagem correspondente, uma abordagem baseada em *match template* pode ser eficaz. O método básico de *match template* consiste em utilizar uma máscara de convolução (o modelo), adaptado para uma característica específica de pesquisa de imagem. Esta técnica pode ser facilmente empregada com as imagens em escala de cinza ou imagens binarizadas, contendo bordas. O resultado do processo de convolução será maior nas regiões onde a estrutura de imagem corresponde à estrutura da máscara, ou seja, onde os valores da imagem serão multiplicados pelos valores da máscara e por serem semelhantes, resultarão em grandes valores.

Este método é normalmente implementado primeiramente escolhendo uma parte da imagem para ser pesquisada com o modelo. Seja a imagem pesquisada $I(x, y)$, onde (x, y) representam as coordenadas de cada pixel da imagem. Vamos chamar o modelo de $T(x_t, y_t)$, onde (x_t, y_t) representam as coordenadas de cada pixel no modelo. Em seguida, simplesmente movimentamos o centro (ou a origem) do modelo de $T(x_t, y_t)$ sobre cada ponto (x, y) da imagem pesquisada e calculamos a soma dos produtos entre os coeficientes de $I(x, y)$ e $T(x_t, y_t)$ ao longo de toda a área do modelo. Como todas as posições possíveis do modelo em relação a imagem pesquisada são consideradas, a posição com a pontuação mais alta será a melhor posição. De acordo com Desai; Pandya e Potdar (2013) existem variantes para as técnicas do tipo *match template*, a saber:

1. Método da diferença do quadrado, calculado pela equação:

$$R_{sq\_diff}(x, y) = \sum_{x', y'} \left[T(x', y') - I(x + x', y + y') \right]^2 ,$$

2. Método da correlação, calculado pela equação:

$$R_{ccor}(x,y) = \sum_{x',y'} \left[T(x',y') \cdot I(x+x',y+y') \right]^2 ,$$

3. Método do coeficiente de correlação, calculado pela equação:

$$R_{ccoeff}(x,y) = \sum_{x',y'} \left[\left(T(x',y') - \frac{1}{(w \cdot h) \sum_{x'',y''} T(x'',y'')} \right) \cdot \left(I(x+x',y+y') - \frac{1}{(w \cdot h) \sum_{x'',y''} I(x+x'',y+y'')} \right) \cdot \right]^2$$

4. Métodos normalizados, calculado pelas equações:

$$Z(x,y) = \sqrt{\sum_{x',y'} \left[T(x',y') \cdot I(x+x',y+y') \right]^2} ,$$

$$R_{sq\_diff\_normed}(x,y) = R_{sq\_diff}(x,y) \Big/ Z(x,y) ,$$

$$R_{ccor\_normed}(x,y) = R_{ccor}(x,y) \Big/ Z(x,y) ,$$

$$R_{ccoeff\_normed}(x,y) = R_{ccoeff}(x,y) \Big/ Z(x,y) ,$$

onde R representa o coeficiente de correlação, w representa a largura e h representa a altura da imagem.

Por questões de simplicidade para implementação e de desempenho, este método possui 98% de precisão quando comparado aos demais até aqui apresentados (DESAI; PANDYA e POTDAR, 2013), a escolha para ser incorporado ao algoritmo foi o método da diferença do quadrado, cuja implementação em linguagem C está transcrita na Figura 2.18.

```
p0 = (double*)sum->data.ptr;
p1 = p0 + templ->cols*cn;
p2 = (double*)(sum->data.ptr + templ->rows*sum->step);
p3 = p2 + templ->cols*cn;
sum_step = sum ? sum->step / sizeof(double) : 0;
sqsum_step = sqsum ? sqsum->step / sizeof(double) : 0;
for( i = 0; i < result->rows; i++ )
{
    float* rrow = (float*)(result->data.ptr + i*result->step);
    idx = i * sum_step;
    idx2 = i * sqsum_step;
    for( j = 0; j < result->cols; j++, idx += cn, idx2 += cn )
    {
        double num = rrow[j], t;
        double wnd_mean2 = 0, wnd_sum2 = 0;
        if( num_type == 1 )
        {
            for( k = 0; k < cn; k++ )
            {
                t = p0[idx+k] - p1[idx+k] - p2[idx+k] + p3[idx+k];
                wnd_mean2 += CV_SQR(t);
                num -= t*templ_mean.val[k];
            }
            wnd_mean2 *= inv_area;
        }
        if( is_normed || num_type == 2 )
        {
            for( k = 0; k < cn; k++ )
            {
                t = q0[idx2+k] - q1[idx2+k] - q2[idx2+k] + q3[idx2+k];
                wnd_sum2 += t;
            }

            if( num_type == 2 )
                num = wnd_sum2 - 2*num + templ_sum2;
        }
        if( is_normed )
        {
            t = sqrt(MAX(wnd_sum2 - wnd_mean2,0))*templ_norm;
            if( t > eps )
            {
                num /= t;
                if( fabs(num) > 1. )
                    num = num > 0 ? 1 : -1;
            }
            else
                num = method != diff || num < DBL_EPSILON ? 0 : 1;
        }
        rrow[j] = (float)num;
    }
}
```

Figura 2.18 – Implementação em linguagem C do algoritmo método da diferença do quadrado.

Neste esquema de identificação é necessário definir uma região de modelo ou *template* da imagem que seja de conhecimento *a priori*, ou seja, é necessário a princípio que este *template* seja conhecido e esteja armazenado em memória para

ser endereçado durante a inicialização do processo de busca. Para a sequência de testes iniciais dois *templates*, conforme a Figura 2.19, foram extraídos das imagens do vídeo. Duas janelas adicionais denominadas ROI e Espectro foram criadas adicionalmente com o intuito de avaliar os passos internos do processo. Nesta captura de imagem, com a viga sem curvas, é possível verificar que a sua localização dentro da ROI está situada na região central.

(a) (b)

Figura 2.19 – *Templates* para utilização no algoritmo de convolução (*match template*), aumentados.
Em (a) o detalhe do modelo da viga em linha reta com a resolução do quadro selecionado equivalente a 62 x 79 pixels. Em (b) o detalhe do modelo da viga em curva com a resolução do quadro selecionado equivalente a 106 x 85 pixels. Ambos *templates* estão representados no espaço de cores RGB.

A imagem formada dentro da janela Espectro permite observar a distribuição pixel a pixel do resultado pós processo de convolução. Os pontos mais brancos indicam a área que possui a maior probabilidade de estar semelhante ao *template*. Para esse teste o *template* (a) da Figura 2.19 foi utilizado. O pixel central desta distribuição, com o maior valor calculado, fornecerá as coordenadas da janela alvo impressas na janela ROI e na imagem em teste. Deve-se ressaltar que os valores obtidos pela expressão da correlação passam por um processo de normalização, podendo assumir somente valores compreendidos entre 0 e 255, daí reside o fato pela qual a janela Espectro apresenta uma coloração de seus pixels numa escala de cinza.

O fluxograma do código atualizado incorporando o método de *match template* está apresentado na Figura 2.20 e na Figura 2.21 observamos o resultado da aplicação da técnica.

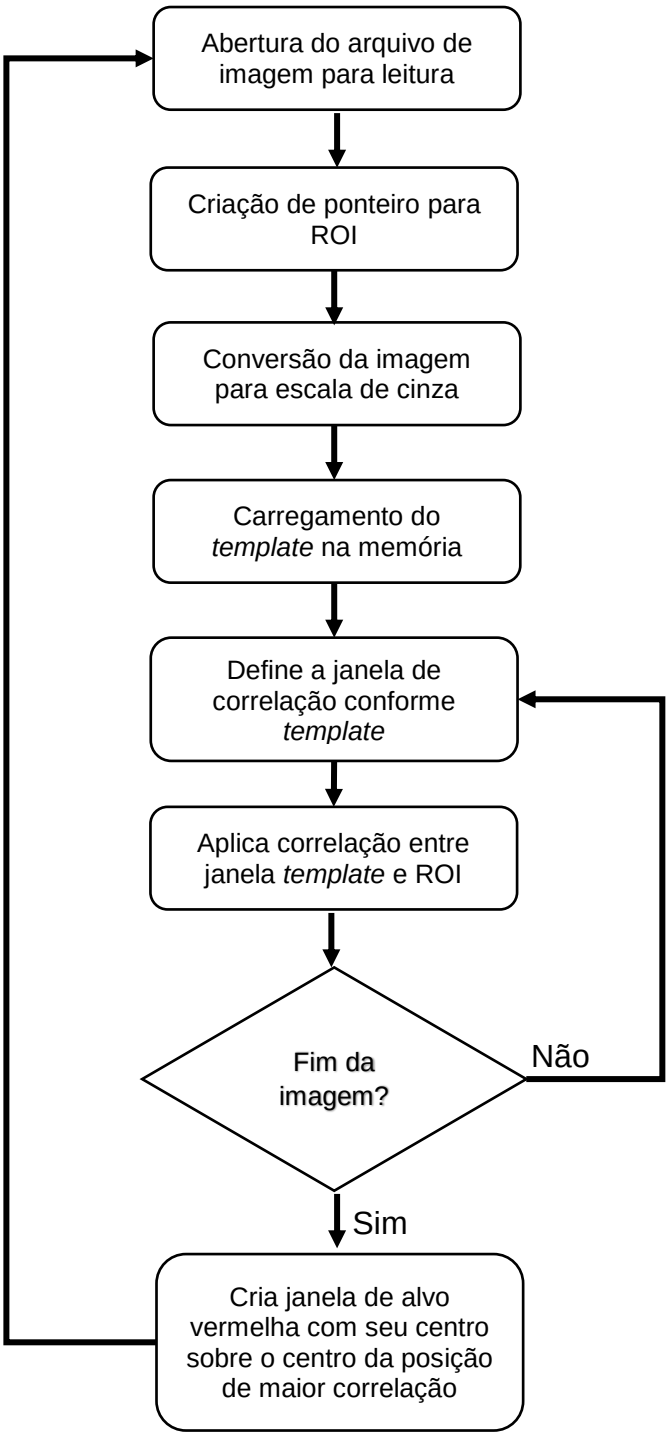


Figura 2.20 – O fluxograma do código atualizado incorporando o método de *match template*.

Figura 2.21 – Resultado da aplicação da técnica de match *template* em uma imagem da via extraída de vídeo com resolução 800 x 600 pixels.

Para a mesma janela foi aplicado o *template* (b) da Figura 2.19, apresentando uma pequena mudança na distribuição dos pixels brancos, que representam alta correlação, mas sem grandes mudanças no resultado final, a posição de máxima correlação da imagem de entrada. O próximo passo foi avaliar o desempenho do algoritmo em imagens contendo diversos cenários, representados da Figura 2.22 até a Figura 2.29.

Todos os testes executados foram repetidos com o *template* apresentado na Figura 2.19 (b). Como não houve alterações significativas durante sua utilização o *template* da Figura 2.19 (a) foi mantido para o restante dos testes. Com a melhoria alcançada na detecção da viga foi implementada a última etapa do algoritmo que consiste na etapa para determinação se a viga está alinhada ou não.

Figura 2.22 – Resposta do algoritmo em curva acentuada à esquerda.
No detalhe da ROI é possível verificar a grande presença de sombra na região da janela. A janela de convolução “Espectro” não apresenta variações acentuadas por conta deste efeito.

Figura 2.23 – Cenário com sombra de trem circulando na outra via.
Ponto luminoso verde no centro do alvo da se refere a luz de guia de alinhamento do AMV. Aumento da área de valor máximo na janela de convolução.

Figura 2.24 – Cenário onde existe transição de uma região clara, para outra sombreada.
Na janela de convolução observa-se o surgimento de regiões onde o cálculo retornou valores maiores (branco) mas que permanecem na região da viga.

Figura 2.25 – Sombra na viga proveniente dos prédios.
Nas áreas periféricas da janela de convolução a resposta é similar à Figura 2.23 porém com a dispersão dos valores mais altos na região central.

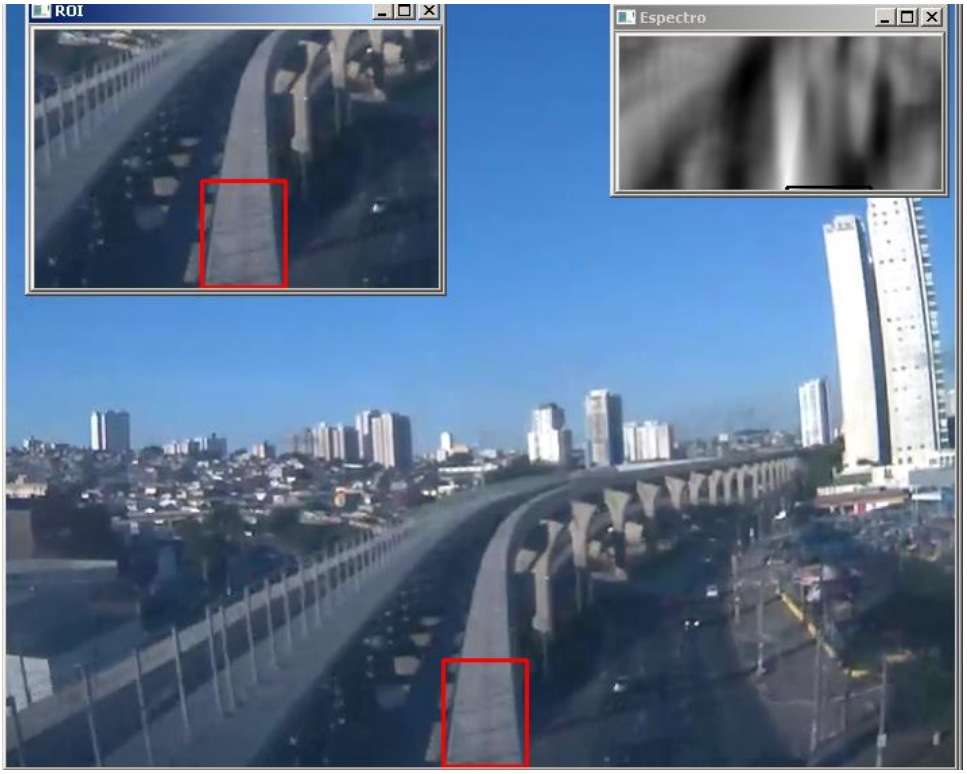


Figura 2.26 – Curva longa à direita.

Na janela de convolução a área correspondente aos maiores valores calculados da convolução concentram-se e alinham-se de maneira bem definida no sentido vertical.

Figura 2.27 – Curva longa à esquerda.

A janela de convolução similar à Figura 2.26. A particularidade reside na inclinação do “feixe” referente aos maiores valores calculados da convolução acompanhar à inclinação da viga.

Figura 2.28 – Aproximação na região de plataforma.

Presença de sombras e outros elementos que se assemelham com o *template*. Janela de convolução apresenta valores maiores no canto inferior esquerdo da ROI, a convolução na região da viga retorna valores praticamente nulos (mancha preta).

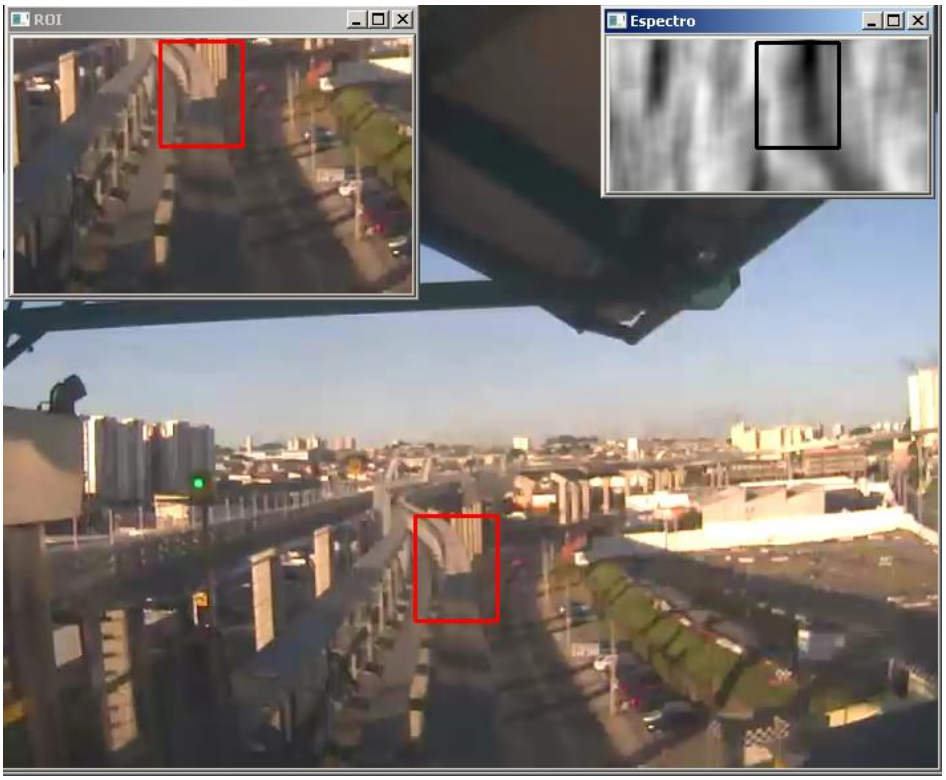


Figura 2.29 – Final de plataforma.

Neste cenário a detecção é instável e a janela de alvo oscila entre locais da ROI. Captura realizada no instante onde a convolução apresentou maiores valores na região da viga.

Para este objetivo duas técnicas foram empregadas: a técnica de preenchimento de pixels e a técnica de contagem de área. A ideia é que a janela de alvo esteja sobre a viga e que a sua dimensão vertical esteja próxima daquela equivalente ao comprimento em pixels da distância a ser detectada, aproximadamente 40 metros; a técnica de preenchimento de pixels simplesmente é disparada no ponto central da janela alvo, a própria viga, e uma área aproximadamente correspondente aos limites da viga seja preenchida. Se esta área preenchida for inferior à área equivalente à mínima necessária, caracteriza-se o desalinhamento da viga. Para a determinação da área utiliza-se a segunda técnica mencionada, a técnica de contagem de pixels para o cálculo de área.

Também conhecida como *flood fill* ou *seed growing*, a técnica de preenchimento de pixels é muito útil e frequentemente utilizada em cenários onde é necessário marcar ou isolar uma determinada região de imagem para aplicar em seguida um processamento específico. Um ponto da região, denominado semente, é selecionado na imagem e todos os outros pontos similares da sua vizinhança são coloridos com a mesma cor do ponto semente. Caso o contorno da região de interesse esteja com contraste bem definido, o preenchimento de uma cor no ponto central praticamente garante o preenchimento de toda região.

O algoritmo *flood fill* – preenchimento de região – necessita para sua inicialização três parâmetros: o ponto de início, uma cor alvo e a cor de substituição. Após sua inicialização o algoritmo varre toda a matriz de cores da imagem observando quais são as posições da imagem que estabelecem um caminho de conexão com o ponto de início desde que mantenha a mesma cor alvo. Após a varredura todas as posições localizadas sofrem modificação da sua cor original para a cor definida por meio do parâmetro nova cor. É possível estabelecer uma analogia do funcionamento desta técnica com as ferramentas de preenchimento de cores, normalmente disponibilizada em softwares de tratamento de imagens e coloquialmente denominada como “balde de preenchimento”. As Figuras 2.41 e 2.42 apresentam a implementação da estrutura de dados e da função *flood fill*, respectivamente, utilizada neste trabalho.

```
struct node
{
    int x, y;
    struct node *next;
};

int push(struct node **top, int x, int y)
{
    struct node *newNode;
    newNode = (struct node *)malloc(sizeof(struct node));
    if(newNode == NULL) //If there is no more memory
        return 0;
    newNode->x = x;
    newNode->y = y;
    newNode->next = *top;
    *top = newNode;
    return 1; //If we push the element correctly
}

int pop(struct node **top, int &x, int &y)
{
    if(*top == NULL) //If the stack is empty
        return 0;
    struct node *temporal;
    temporal = *top;
    x = (*top)->x;
    y = (*top)->y;
    *top = (*top)->next;
    free(temporal);
    return 1; //If we pop an element
}
```

Figura 2.30 – Estrutura de dados e funções auxiliares para o algoritmo de *flood fill.*

```
void floodFill(int x, int y, int color_to_replace, int color_to_fill)
{
    if(color_to_replace == color_to_fill) return;
    struct node *stack = NULL;
    if(push(&stack, x, y) == 0)  return;
   while(pop(&stack, x, y) == 1)
   {
       pixel(x, y, color_to_fill);
       if(x+1 < 640 && read_pixel(x+1, y) == color_to_replace)
           if(push(&stack, x+1, y) == 0) return;
       if(x-1 >= 0 && read_pixel(x-1, y) == color_to_replace)
           if(push(&stack, x-1, y) == 0) return;
       if(y+1 < 480 && read_pixel(x, y+1) == color_to_replace)
           if(push(&stack, x, y+1) == 0) return;
       if(y-1 >= 0 && read_pixel(x, y-1) == color_to_replace)
           if(push(&stack, x, y-1) == 0) return;
   }
}
```

Figura 2.31 – Algoritmo *flood fill.*

A Figura 2.43 apresenta o exemplo de aplicação da técnica de preenchimento na primeira imagem de avaliação.

Figura 2.32 – Aplicação da técnica de preenchimento de região na primeira imagem de avaliação. A descontinuidade nas cores dos pixels, sobre a superfície da viga, segrega naturalmente a área.

Observando os resultados da aplicação da técnica de preenchimento foi definido o critério para detecção de alinhamento baseado no fato de que a ausência de viga cria naturalmente na imagem uma descontinuidade sobre a superfície da viga, em função desta característica é analisada dentro da janela alvo a área útil limitada pela superfície cujo preenchimento foi completado. Um exemplo de descontinuidade na viga e o efeito do preenchimento sob a região de sua área é apresentado na Figura 2.44.

Figura 2.33 – Aplicação da técnica de preenchimento de região numa região de mudança de via no pátio Oratório – Linha 15.

Na região central da imagem a viga está alinhada. À esquerda e à direita da viga central não existe alinhamento estabelecido, condição de interrupção da viga guia.

Na Figura 2.44 observa-se que o resultado da aplicação da técnica de preenchimento sobre os caminhos não alinhados estabeleceu uma estimativa da área sobre a viga, representada pelas cores vermelho escuro, verde claro e verde escuro. Nesta condição os limites na região de borda das vigas foram bem delimitados pela aplicação da técnica. Para estimar o cálculo de área da região preenchida contabilizam-se todos os pixels na cor vermelha dentro da área de alvo. Com o critério de 70% da área de alvo, o que equivale a 200 pixels (altura estimada para 40 metros) vezes 20 pixels (largura do alvo), realiza-se o julgamento se existe ou não o alinhamento. As Figuras 2.45 e 2.46 apresentam duas imagens do resultado final do algoritmo de reconhecimento nas condições de operação durante o dia e à noite. O fluxograma representando a estrutura do algoritmo final é apresentado na Figura 2.47.

Figura 2.34 – Resultado final do algoritmo – movimentação diurna do trem.

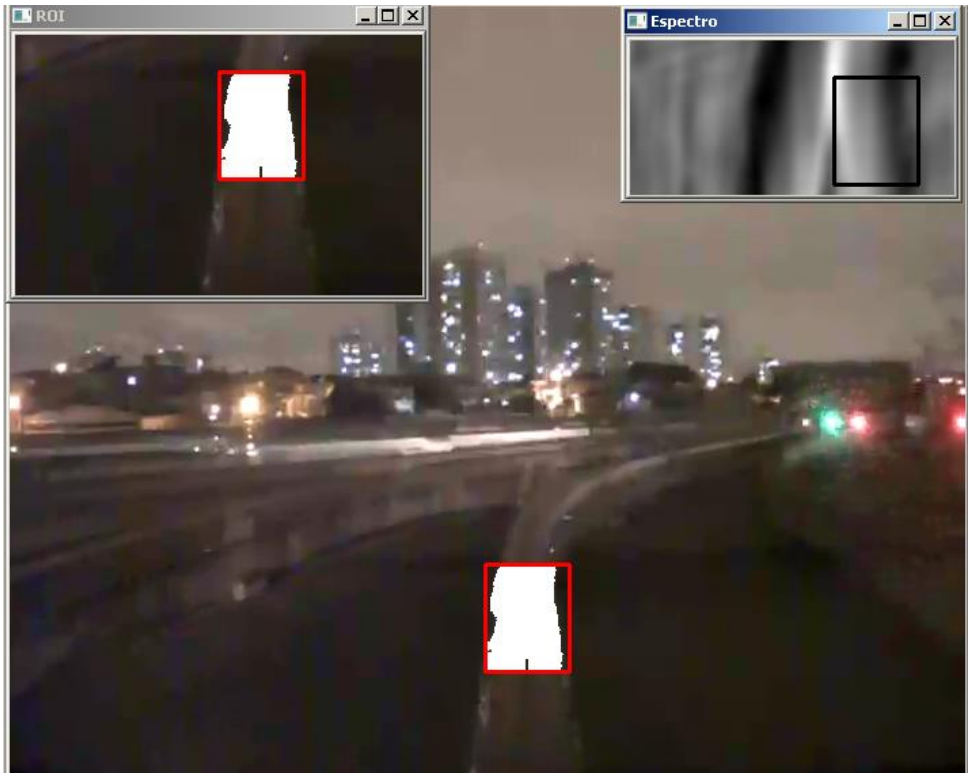


Figura 2.35 – Resultado final do algoritmo – movimentação noturna do trem.

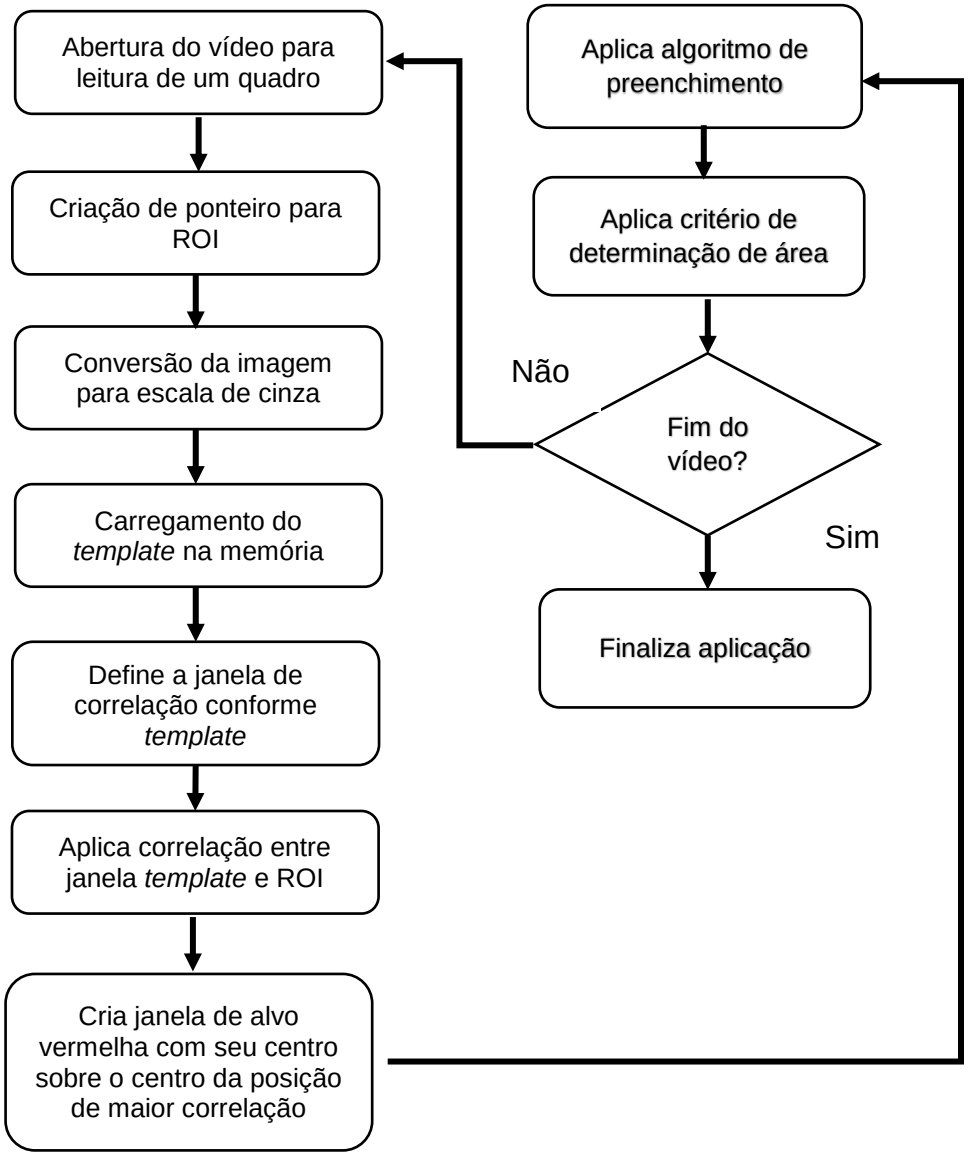


Figura 2.36 – O fluxograma da implementação final do algoritmo de detecção de desalinhamento de viga.

3. TRABALHANDO COM O OPENVC NO SDDV

3.1. Instalação

Para realizar a configuração do ambiente conforme a organização proposta na Figura 1.1, foi iniciada a sequência de instalação das ferramentas a partir do *software* Visual Studio, seguida pela instalação da biblioteca OpenCV e finalizada com a instalação da ferramenta CMake. As instalações de todos os *softwares* foram realizadas de forma completa, ou seja, foram instalados todos os recursos de cada ferramenta, conforme as opções disponíveis em cada aplicativo assistente de instalação acordo com o ilustrado nas Figuras Figura 3.1, Figura 3.2 e Figura 3.3. Após a conclusão do processo de instalação das ferramentas, o próximo passo foi extrair da Internet, por meio da seção *downloads* da página www.opencv.org, o arquivo de configuração do CMake escrito para compilação da biblioteca OpenCV no ambiente Visual Studio. Uma vez aberto este arquivo dentro da ferramenta CMake e iniciada e concluída a compilação da biblioteca nativa, segundo as configurações determinadas em seu script, foi criada uma pasta denominda “VS2008” dentro da pasta de instalação gerada originalmente pelo assistente de instalação do OpenCV. Dentro da pasta, “VS2008”, foram criados automaticamente pelo CMake um projeto denominado “OpenCV” e todos os códigos fonte da biblioteca escritos em linguagem C. Deste ponto da configuração foi então iniciada a execução do *software* Visual Studio e solicitado a abertura do projeto “OpenCV”. Uma vez aberto este projeto, a etapa seguinte foi iniciar o processo de compilação dentro do ambiente, por meio do comando “Compile” disponibilizado no menu superior principal do Visual Studio, de maneira que ao final deste processo de compilação fossem criadas todas as dll`s – *dinamic link libraries* – do OpenCV para utilização em linguagem C dentro do ambiente Visual Studio, finalizando portanto toda a configuração do ambiente. Com as dll`s geradas é necessário criar um novo projeto dentro do ambiente Visual Studio e referenciar os arquivos dll`s que contenham as funções de visão computacional do OpenCV mais adequadas à proposta dos algoritmos desenvolvidos.

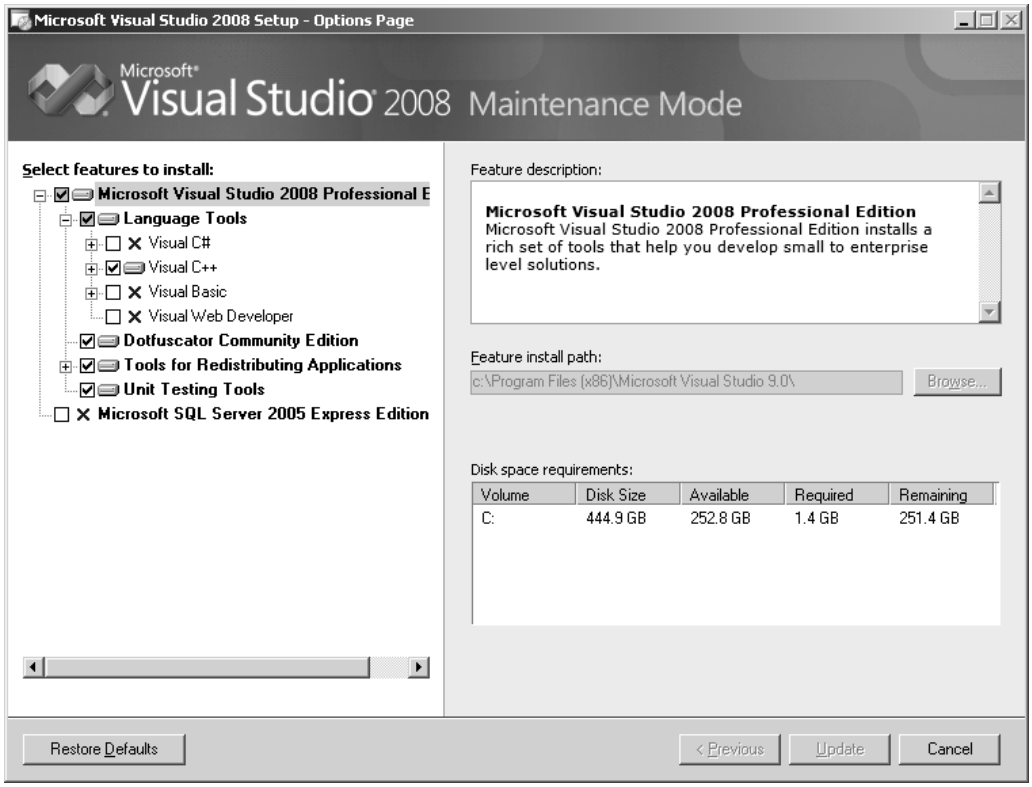


Figura 3.1 – Tela principal das opções para instalação do Microsoft Visual Studio®.

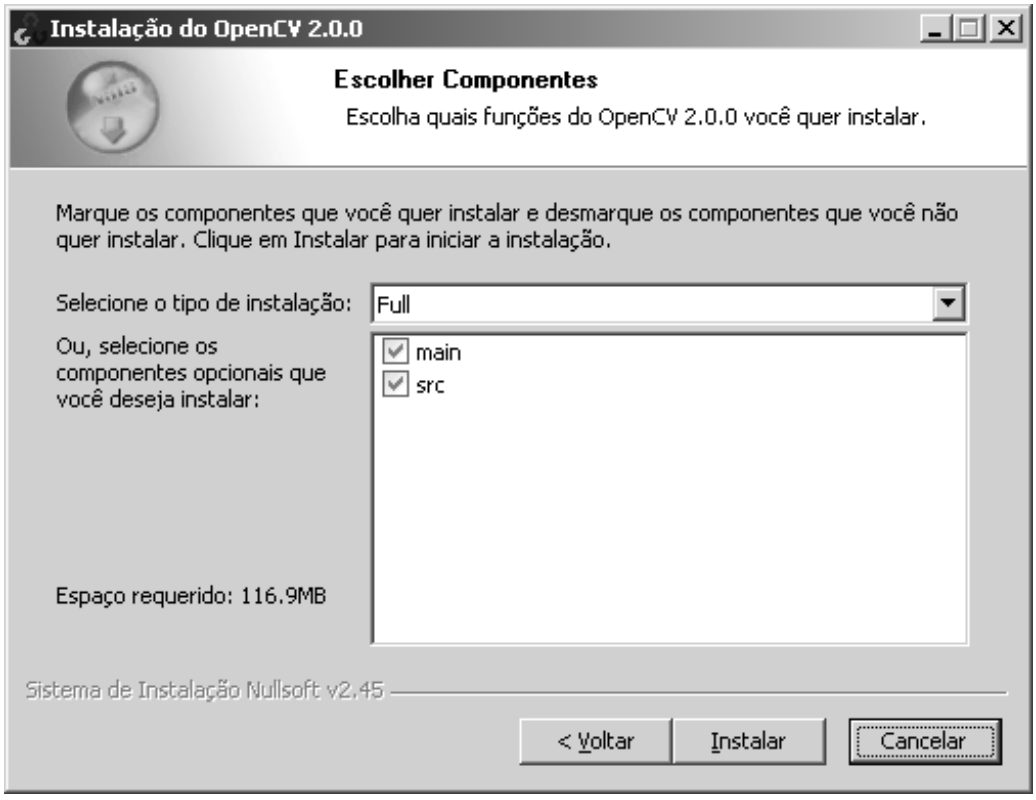


Figura 3.2 – Tela principal das opções para instalação do OpenCV.

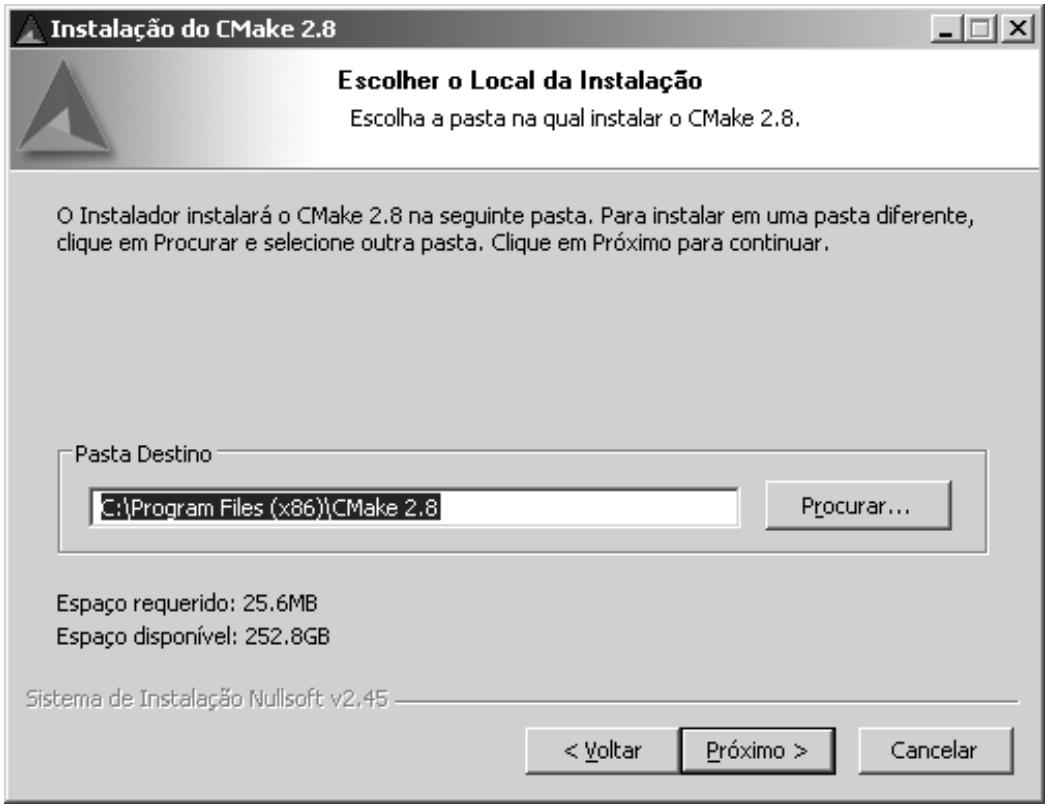


Figura 3.3 – Tela principal das opções para instalação do *software* CMake.

3.2. Descrição das funções

A partir da observação da Figura 3.4, que representa o algoritmo final desenvolvido para o sistema SDDV, foram relacionadas todas as funções e estruturas da biblioteca OpenCV utilizadas no trabalho, transcritas na Tabela 3.1.

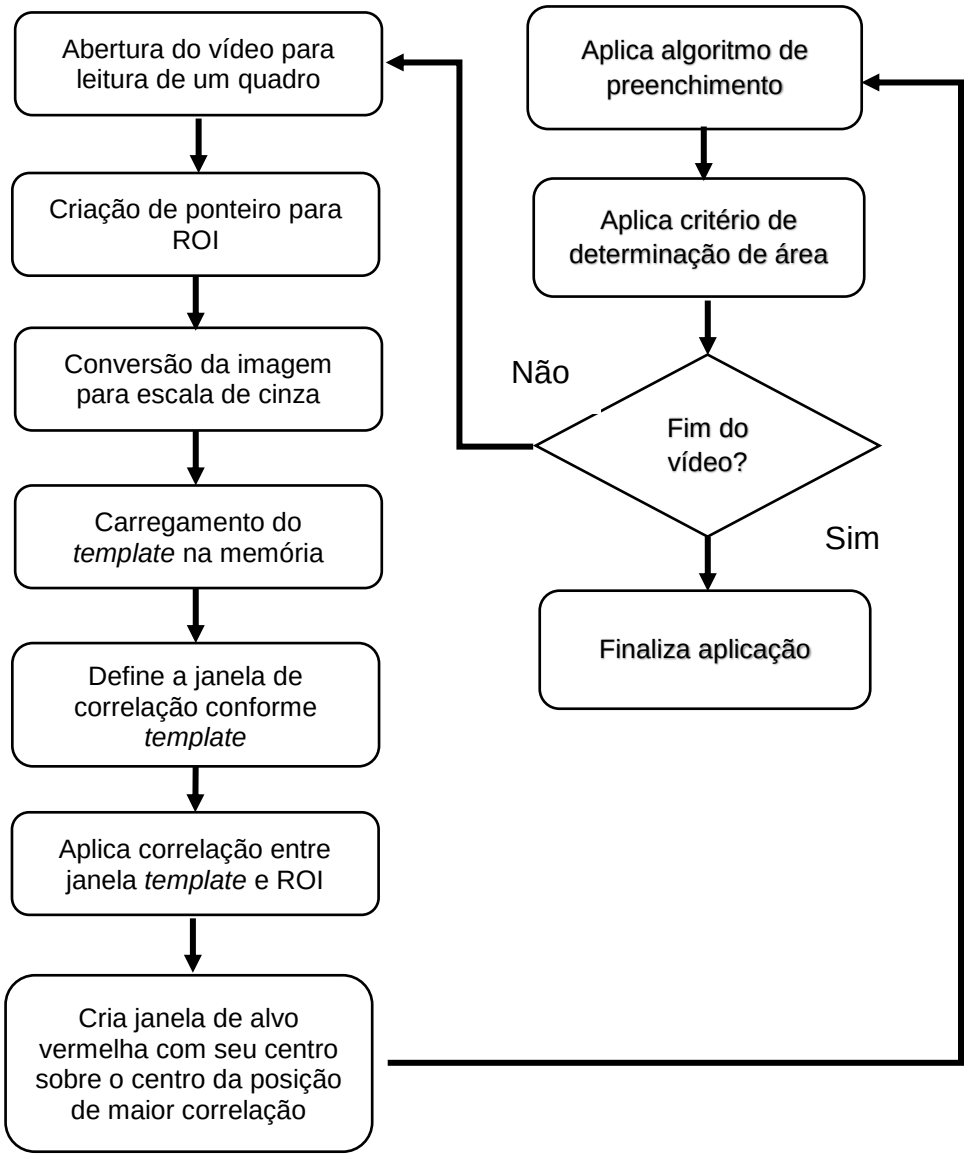


Figura 3.4 – O fluxograma da implementação final do algoritmo de detecção de desalinhamento de viga.

Tabela 3.1 – Funções do OpenCV utilizadas no SDDV.

Função	Tipo	Biblioteca
CvCapture	Estrutura de dados	Highgui
IplImage	Estrutura de dados	*Cxtypes
cvMat	Estrutura de dados	*Cxcore
Imread	Função	Highgui
cvCreateFileCapture	Função	Highgui
cvQueryFrame	Função	Highgui
cvSetCaptureProperty	Função	Highgui
cvCreateImage	Função	*Cxcore
cvSetImageROI	Função	*Cxcore
cvCvtColor	Função	*Cv
cvCanny	Função	*Cv
rectangle	Função	*Cxcore
namedWindow	Função	Highgui
cvReleaseCapture	Função	Highgui
cvDestroyWindow	Função	Highgui

*Integram a biblioteca *Core*, apresentada na tabela 1.1.

As estruturas CvCapture, IplImage e cvMat foram utilizadas para manipular as informações em formato JPEG contidas nas cenas de vídeo em formato MPEG.

A estrutura IplImage é definida como padrão de imagem da Intel Processing Library (IPL). A definição exata da estrutura IplImage é mostrado na Figura 3.5.

```
typedef struct _IplImage {
     int nSize;
     int ID;
     int nChannels;
     int alphaChannel;
     int depth;
     char colorModel[4];
     char channelSeq[4];
     int dataOrder;
     int origin;
     int align;
     int width;
     int height;
     struct _IplROI* roi;
     struct _IplImage* maskROI;
     void* imageId;
     struct _IplTileInfo* tileInfo;
     int imageSize;
     char* imageData;
     int widthStep;
     int BorderMode[4];
     int BorderConst[4];
     char* imageDataOrigin;
} IplImage;
```

Figura 3.5 – Estrutura IplImage

Após os membros *width* e *height*, *depth* e *channels* são as mais importantes membros desta estrutura. A variável *depth* está associada a um de um conjunto de valores definidos no arquivo ipl.h, que definem o formato de representação numérica dos valores armazenados na matriz de pontos da imagem, contidos nesta estrutura. Os próximos dois membros importantes são *origin* e *dataOrder*. A variável *origem* pode assumir os valores IPL_ORIGIN_TL ou IPL_ORIGIN_BL, correspondente à localização da origem das coordenadas da imagem, que pode estar localizado em qualquer dos cantos, superior esquerdo ou inferior esquerdo, da imagem, respectivamente. A falta de um padrão na origem das coordenadas (superior versus inferior) é uma fonte importante de erro nas rotinas de visão computacional. Dependendo de onde uma imagem foi gerada, sistema operacional, *codec*, formato de armazenamento, câmera, etc pode afetar a localização da origem das coordenadas e consequentemente gerar erros de execução dos algoritmos e estouros de ponteiros no acesso a memória.

O membro *dataOrder* pode assumir os valores IPL_DATA_ORDER_PIXEL ou IPL_DATA_ORDER_PLANE. Este valor indica se os dados devem ser empacotados com vários canais um após o outro para cada pixel (intercalados, o caso mais usual), ou todos os canais agrupados em planos de imagem com os planos dispostos um após o outro.

O parâmetro *widthStep* contém o número de bytes entre pontos na mesma coluna e linhas sucessivas. A largura variável não é suficiente para calcular a distância, porque cada linha pode ser alinhada com um determinado número de bytes para conseguir um processamento mais rápido da imagem; conseqüentemente pode haver algumas lacunas entre o fim da i-ésima linha e o início da (i + 1) linha. O parâmetro *imageData* contém o ponteiro para a primeira linha de dados de imagem. Se há várias planos separados na imagem (como quando dataOrder = IPL_DATA_ORDER_PLANE), então eles são colocadas consecutivamente como imagens separados de *height* vezes *nChannels* linhas no total, mas normalmente eles estão intercalados de modo que o número de linhas é igual à altura e com cada linha contendo os canais intercalados em ordem. E ,enfim, existe a importante região de interesse (ROI), que é na verdade um instância de outra estrutura IPL / IPP, Ipl ROI. Um IPL ROI contém um *xOffset*, um *yOffset*, um *height*, um *width* e uma *coi*, onde COI significa *channel of interest*.

A estrutura cvMat, outra importante estrutura de dados da biblioteca OpenCV, ao contrário da IplImage é uma estrutura mais simples cujo objetivo é armazenar as informações de uma imagem em formato *raw* para àquelas funções que dispensam em seus parâmetroas as informações relacionadas a cor, canais e tipo de dado. A Figura 3.6 apresenta os membros que compõem este tipo de dado.

```
typedef struct CvMat {
      int type;
      int step;
      int* refcount; // for internal use only
      union {
            uchar* ptr;
            short* s;
            int* i;
            float* fl;
            double* db;
      } data;
      union {
            int rows;
            int height;
      };
      union {
            int cols;
            int width;
      };
} CvMat;
```

Figura 3.6 – Estrutura CvMat

A partir de agora serão descritos todas as funções do OpenCv utilizadas no trabalho.

Função Imread

Carrega uma imagem a partir de um arquivo em disco e retorna um ponteiro para uma estrutura tipo Mat. Os seguintes padrões de imagem são suportados:

- Windows bitmaps - *.bmp, *.dib
- JPEG files - *.jpeg, *.jpg, *.jpe
- JPEG 2000 files - *.jp2
- Portable Network Graphics - *.png
- *Portable image* - *.pbm, *.pgm, *.ppm
- *Sun rasters* - *.sr, *.ras
- TIFF files - *.tiff, *.tif

Sintaxe:

```
Mat imread( const string& filename, int flags=1 );
```

Parâmetros:

filename Nome do arquivo a ser carregado.

flags Especifica o tipo de cor a ser carregado na imagem: se >0 a imagem carregada é forçada a ter 3 canais de cor. Se 0 a imagem a possuir escala de cinza.

Função cvCreateFileCapture

Aloca e inicializa a estrutura CvCapture para leitura de um *stream* de dados de vídeo a partir de um arquivo em disco.

Sintaxe:

```
public static IntPtr cvCreateFileCapture(string    filename);
```

Parâmetros:

filename Nome do arquivo de vídeo.

IntPtr Ponteiro para estrutura CvCapture.

Função cvQueryFrame

Apanha e retorna um quadro a partir de uma câmera ou arquivo. A função cvQueryFrame apanha um quadro de uma câmera ou arquivo de vídeo, descomprime e a retorna numa estrutura tipo IplImage.

Sintaxe:

```
IplImage* cvQueryFrame( CvCapture* capture );Parâmetros:
```

Parâmetros:

capture Estrutura do arquivo de vídeo.

Função cvSetCaptureProperty

Configura as propriedades para captura do vídeo. A função cvQueryFrame apanha um quadro de uma câmera ou arquivo de vídeo, descomprime e a retorna numa estrutura tipo IplImage.

Sintaxe:

```
double cvGetCaptureProperty( CvCapture* capture, int property
id );
```

Parâmetros:

capture Estrutura do arquivo de vídeo.

property_id Identificador da propriedade.

CV CAP PROP POS MSECF	Posição atual do filme em milisegundos
CV CAP PROP POS FRAMES	Índice para a posição do quadro a ser capturado.
CV CAP PROP POS AVI RATIO	Posição relativa do arquivo de vídeo
CV CAP PROP FRAME WIDTH	Largura dos quadros no vídeo.
CV CAP PROP FRAME HEIGHT	Altura dos quadros no vídeo.
CV CAP PROP FPS	*Frame rate.*
CV CAP PROP FOURCC	Código do *codec* em 4-caracteres.
CV CAP PROP FRAME COUNT	Número de quadros no vídeo.
CV CAP PROP BRIGHTNESS	Brilho da imagem (para câmeras).
CV CAP PROP CONTRAST	Contraste da imagem (para câmeras)
CV CAP PROP SATURATION	Saturação da imagem (para câmeras).
CV CAP PROP HUE	*Hue* da imagem (para câmeras).

Função cvCreateImage

Cria um cabeçalho e aloca memória para os dados de uma imagem.

Sintaxe:

```
cvCreateImage(size, depth, channels)->image
```

Parâmetros:

size	Estrutura do arquivo de vídeo.
depth	Número de elementos de bits da imagem.
channels	Número de canais por pixel.

Função cvSetImageROI

Configura o retângulo de uma região de interesse de uma imagem.

Sintaxe:

```
void cvSetImageROI(IplImage* image, CvRect rect );
```

Parâmetros:

image	Ponteiro para o cabeçalho da imagem.
rect	Tipo de dado que representa o retângulo da ROI.

Função cvCvtColor

Converte uma imagem de um espaço de cores para outro.

Sintaxe:

```
void cvCvtColor( const CvArr* src, CvArr* dst, int code );
```

Parâmetros:

src	A imagem de origem em 8-bit (8u), 16-bit (16u) ou ponto flutuante de precisão simples(32f).
dst	A imagem de destino.
code	Constante que representa o tipo de operação de conversão de cores.

Função cvCanny

Implementa o algoritmo de *Canny* para detecção de bordas.

Sintaxe:

```
void cvCanny( const CvArr* image, CvArr* edges, double threshold1, double threshold2, int aperture size=3 );
```

Parâmetros:

image	Imagem de entrada, em canal simples.
edges	Imagem de retorno com bordas em canal simples.
threshold1	Valor de ajuste do primeiro limiar.
threshold2	Valor de ajuste do segundo limiar.
aperture_size	Parâmetro de abertura do operador *Sobel*.

Função rectangle

Desenha um retângulo com espessura e preenchimento.

Sintaxe:

```
void cvRectangle( CvArr* img, CvPoint pt1, CvPoint pt2, CvScalar color, int thickness=1, int line type=8, int shift=0 );
```

Parâmetros:

img	Imagem
pt1	Um dos vértices do retângulo.
pt2	Vértice oposto do retângulo.
color	Cor da linha em RGB.
thickness	Espessura da linha.
line_type	Tipo da linha.
shift	Número de bits fracionais nos pontos de coordenadas.

Função namedWindow

Cria uma janela que pode ser utilizada para exibir imagens e *trackbars*.

Sintaxe:

```
int cvNamedWindow( const char* name, int flags );
```

Parâmetros:

name Nome da janela que será utilizada como identificador.

flags Se configurado em 1, o tamanho da janela é ajustado em função do conteúdo exibido.

Função cvReleaseCapture

Desaloca a estrutura CvCapture da memória.

Sintaxe:

```
void cvReleaseCapture( CvCapture** capture );
```

Parâmetros:

capture Ponteiro para a estrutura CvCapture.

Função DestroyWindow

Destroi uma janela.

Sintaxe:

```
void cvDestroyWindow( const char* name );
```

Parâmetros:

name Nome da janela que será destruída.

4. CONCLUSÃO

Este trabalho propôs uma nova abordagem para o reconhecimento das condições de desalinhamento de via por meio de um sistema baseado na captura de imagens com base na arquitetura proposta e apresentada para o SDDV.

O tempo final de processamento do algoritmo desenvolvido, considerando as condições do *hardware* utilizado, é próximo a 7ms. Este tempo é inferior ao intervalo de tempo entre dois quadros sucessivos de um vídeo gerado pela câmera, 33ms, e os tempos de reação necessários para atuação do sistema de frenagem do trem, conforme as considerações apresentadas durante a modelagem do problema. Este é um ponto positivo, pois fornece uma perspectiva para a margem de implementação de um produto final, levando em consideração que em uma arquitetura embarcada, os recursos computacionais de processamento são normalmente mais escassos do que o disponibilizado em microcomputadores pessoais.

A utilização da biblioteca OpenCV para o desenvolvimento do algoritmo base do SDDV se mostrou muito eficiente. Todas as funções e estruturas selecionadas e utilizadas trabalharam de maneira adequada e não ofereceram nenhum tipo de obstáculo durante o desenvolvimento do algoritmo. A documentação de suporte à sua utilização é muito detalhada e somado ao fato de ser uma biblioteca com código fonte aberto, auxilia no rápido aprendizado pois é permitido o acesso irrestrito para estudo à todas as funções disponibilizadas. Outro fator relevante é que o fato desta biblioteca ser concebida para permitir sua compilação sobre diversas plataformas, imaginando o produto final, a microprocessador embarcado poderá receber o código portado desenvolvido no Visual Studio sem nenhum tipo de problema, desde que o ambiente de desenvolvimento do microprocessador embarcado seja compatível com as linguagens de programação C ou C++.

No decorrer do desenvolvimento do SDDV, notou-se que o cenário de aplicação deste sistema permite que a solução seja personalizada dentro de certos padrões, como parâmetros da via, velocidade do trem, posicionamento da câmera, aspectos do aparelho de mudança de via e padrões da imagem. Assim, foi possível observar que muitas informações contidas nas imagens coletada pela câmera, podem ser consideradas fora da região de interesse do problema otimizando a massa de informações a ser processada. Para a proposta de extração das bordas da

imagem original, a função `Canny` apresentou-se melhor desempenho na detecção em trechos de curva comparada à função `Sobel`. Os parâmetros desta função foram determinados de modo a se obter melhor equilíbrio entre a informação principal da imagem e os ruídos ou distorções.

Para a identificação de formas geométricas conhecidas dentro da imagem binarizada, resultante da função `Canny`, tentou-se utilizar a Transformada de *Hough* para eleger retas e curvas próximas dos padrões da viga-guia. No entanto, devido ao demasiado tempo gasto para a estimativa de parábolas, cerca de três vezes maior que o tempo de intervalo entre os *frames* do vídeo, essa ferramenta foi descartada.

Outra proposta foi realizar estimativas dos pontos da borda através do método de mínimos quadrados recursivo, implementado na forma de *Kalman*. Sua grande vantagem foi, ao utilizar matrizes de menor dimensão e evitar uso de inversão de matrizes, exigir menos memória para o armazenamento e demandar menor tempo de processamento. No entanto, foram observadas algumas limitações através da análise das curvas de resposta. Devido ao efeito das sombras na imagem original, em alguns casos, o algoritmo não foi capaz de identificar os primeiros pontos que fornecem a orientação da reta inicial. Além disso, caso existam muitos pontos na periferia do traçado real, como há uma tendência do processo em ajustar um valor médio entre as curvas, houve um efeito de deslocamento em relação a curva real da viga-guia. As influências de ruídos demonstraram, de forma significativa, implicar na dificuldade no acompanhamento da trajetória esperada.

Por fim, a proposta que se mostrou mais eficaz para a identificação da viga foi através da técnica do *match template*, onde foram pré-definidas duas regiões de modelo para a imagem: uma para a viga em linha reta e outra para a viga em curva. Desta forma, a comparação dos padrões se deu através da análise do espectro de probabilidade da posição da viga. A estimativa da posição resultante desta análise, que apresenta 98% de precisão, se mostrou bem estável, mesmo havendo alguma divergência nas regiões escuras ou com sombras. Nas verificações com as imagens durante o período da noite e com chuva, mas com o mesmo *template* gerado da imagem durante o dia, observou-se um comportamento com maior instabilidade de detecção. Durante à noite existem mais pontos intensos de luminosidade, ou causados pela sinalização de via, ou pela iluminação pública, que resultam em

resultados que provém alto valor de correlação entre o padrão de imagem predefinido da viga e a viga real da imagem

A verificação da continuidade da viga através do cálculo da área preenchida pela ferramenta *flood fill* permitiu identificar a descontinuidade da viga. No entanto, em algumas situações, os fatores externos podem incorrer em uma conclusão equivocada, interpretando regiões de sombras como se fosse uma descontinuidade da viga. Uma proposta para melhorar essa interpretação seria conciliar processos distintos de verificação, por diferentes métodos e executá-los em paralelo, para que sejam avaliados em conjunto. Após o processamento e o reconhecimento da condição de desalinhamento da via, o dispositivo iniciará a interface com o MTC, que tomará a decisão quanto à aplicação do freio de emergência no trem. Essa comunicação pode ser realizada através da rede de comunicação disponível no trem ou através de sinal exclusivo para esta finalidade. Foi demonstrado que é possível identificar os limites das bordas da viga-guia e realizar verificações computacionais da continuidade da via.

Decorrente dos resultados apresentados a partir deste trabalho considera-se como pontos de melhoria as seguintes propostas:

1. Obter vídeos de movimentação do trem a frente de regiões de mudança de via em condições de desalinhamento. Devido as premissas de segurança implementadas pela Operação do Metrô, as obtenções destes vídeos não foram permitidas até então, mas com o decorrer do comissionamento da Linha 15, será possível obter vídeos nestas condições. Estas informações deverão ser entrada para o algoritmo a fim de verificarmos o seu desempenho num cenário real de desalinhamento.
2. Adoção de um mapa de *templates*, que represente as características individuais dos trechos de via com maior interesse. Esta melhoria além de tornar mais robusto o cálculo de correlação, pode ser entrada para um sistema de votação que poderá determinar com precisão a exata região de mudança em que se localiza o trem.
3. Verificar alternativas para os algoritmos de detecção de bordas. Entendemos que é possível aplicar um método de estimação robusta para a identificação do contorno da borda da viga. A proposta futura será implementar o algoritmo RANSAC e avaliar o seu desempenho nestas

condições. Os benefícios poderiam refletir na diminuição do tempo de detecção das bordas e o aumento de precisão da contagem de área sobre a viga, desde que um modelo de ajuste mais fiel a geometria real como um clotóide, seja definido para o processo de estimação.

4. Realizar os primeiros estudos para a montagem de um protótipo embarcado. Com o algoritmo mais consolidado, o próximo passo será iniciar a especificação de um protótipo eletrônico capaz de executar a função de processamento das imagens, comunicação com as câmeras de bordo do trem e interface com o sistema de freios do trem.

Com maior investimento de tempo e recurso neste trabalho, o SDDV poderá ser adotado como parte de uma solução alternativa às existentes, e de forma gradativa dispensar o emprego de dispositivos instalados em via para o cumprimento desta função.

5. REFERÊNCIAS BIBLIOGRÁFICAS

ATTNEAVE, F. Some information aspects of visual perception. **Psychological review**, v. 61, n. 3, p. 183-193, 1954.

BALLARD, D. H.; BROWN, C. M. **Computer vision**. New Jersey: Prentice-Hall, 1982. 523 p.

BERNARDES, F. S.; FERNANDES, D. C.; SANTOS, R. S. **Sistema para detecção de desalinhamento de viga-guia em regiões de mudança de via dos Sistemas Monotrilho**. São Paulo. 2016. 100 p. Monografia (Especialização em Tecnologia Metroferroviária) – Escola Politécnica da Universidade de São Paulo. PECE – Programa de Educação Continuada em Engenharia. Universidade de São Paulo, São Paulo, 2016.

BOVIK, A. C. **Handbook of image and video processing**. Canadá: Academic Press, 2000. 891 p.

BRADSKI, G.; KAEHLER, A. **Learning opencv: computer vision with the opencv library.** 1. ed. Califórnia: O'Reilly, 2008. 555 p.

CHAPRA, S. C. **Applied numerical methods with matlab for engineers and scientists**. 3. ed. Boston: McGraw-Hill, 2008. 673 p.

DESAI, B. K.; PANDYA, M.; POTDAR, M. B. Comparison of various template matching for face recognition. **International journal of engineering research and development – IJSERD**, v. 8, p. 16-18, 2013.

GREWAL, M. S.; ANDREWS, A. P. **Kalman filtering: theory and practice with matlab**, 4. ed. New Jersey: John Wiley & Sons, 2015. 617 p.

KUGA, H. K. **Notas de aula de 2005 - noções práticas de técnicas de estimação.** São José dos Campos: INPE, 2005. Apostila para disciplina de pós-graduação da Divisão de Mecânica Espacial e Controle.