UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

GUILHERME CARDOSO SOARES

Deconvolução não cega de Imagens em Dispositivos Móveis

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Manuel Menezes de Oliveira Neto

Porto Alegre
2015

AGRADECIMENTOS

Aos meus pais, Ana Izabel Pereira Cardoso Soares e Valdemar Silva Soares, por me proporcionarem todas as condições e apoio fundamentais para a realização deste sonho. A meu irmão, por sempre me auxiliar em momentos difíceis durante os últimos semestres.

Aos meus amigos, que me apoiaram e me acompanharam durante todos esses anos de graduação. Aos caros colegas Alexandre Wermann e Lucas Carraro pela grande ajuda para que este trabalho pudesse ser concluído.

Aos amigos da família, bem como os familiares mais próximos, que sempre deram o apoio necessário para que eu tivesse forças para a realização deste sonho.

RESUMO

A captura de uma imagem pode ser modelada através de uma convolução entre a cena, um *kernel* de convolução e um ruído. Por conta disso, a imagem resultante pode conter artefatos indesejados, borrões, etc. Para tentarmos recuperar a imagem de forma que ela seja o mais fiel a cena, podemos realizar a operação inversa a qual ocorreu durante a captura: uma deconvolução. A deconvolução trata-se de uma técnica que busca remover artefatos indesejados de uma imagem, ela pode ser cega, necessitando apenas da imagem, ou não cega, quando precisamos ter as informações da imagem e do *kernel* de convolução.

Neste trabalho utilizaremos uma solução para deconvolução não cega proposta por Fortunato e Oliveira [2014], modelada como um sistema linear e resolvida no domínio frequência. A deconvolução foi implementada em Java, para ser executada em um dispositivo móvel - *smartphone* - seguindo a proposta mencionada. Apresentaremos a biblioteca de visão computacional utilizada para a realização do método: a OpenCV. Abordaremos a estrutura do aplicativo desenvolvido, todas suas etapas de execução. Logo após, serão apresentados os resultados obtidos, comparando o algoritmo executado no *smartphone* com a versão MATLAB disponibilizada pelos autores.

Por fim, comentamos sobre possíveis técnicas de estimar o *kernel* de borramento associado a uma imagem, em tempo de captura, bem como nossa tentativa de estimativa.

Palavras-chave: Deconvolução, Processamento de Imagens, PSF.

ABSTRACT

The capture an image can be modeled by a convolution of the scene, a convolution kernel and a noise. Because of this, the resulting image can contain unwanted artifacts, blurriness, etc. To try to retrieve the image so that it is as faithful a scene, we can perform the reverse operation which occurred during capture: A deconvolution. The deconvolution it is a technique that seeks to remove unwanted artifacts from an image, it may be blind, requiring only image, or not blind when we need to have the information of the image and the convolution kernel.

In this paper we use a solution to not blind deconvolution proposed by Fortunato and Oliveira [2014], modeled as a linear system and resolved in the frequency domain. The deconvolution was implemented in Java, to run on a mobile device - smartphone - following the mentioned proposal. We present a computer vision library used for performing the method: the OpenCV. We discuss the structure of the developed application, all stages of execution. Soon after, the results will be presented, comparing the algorithm runs on the smartphone with the MATLAB version made available by the authors.

Finally, we will comment on the possible techniques for estimating the blurring kernel associated with an image in capture time, as well as our attempt to estimate.

Keywords: deconvolution, Image Processing, PSF.

LISTA DE FIGURAS

LISTA DE ABREVIATURAS E SIGLAS

AOT	Ahead of Time
ART	Android Runtime
JIT	Just In Time
JNI	Java Native Interface
JVM	Java Virtual Machine
MATLAB	Matrix Laboratory
OPENCV	Open Source Computer Vision Library
PSF	Point Spread Function
SDK	Software Development Kit
UFRGS	Universidade Federal do Rio Grande do Sul
XML	eXtensible Markup Language

SUMÁRIO

1 INTRODUÇÃO

Com a evolução da tecnologia, os smartphones estão cada vez mais presentes em nossas vidas, nos auxiliando em inúmeras funções. Uma destas funções, dentre as mais utilizadas, além do acesso a internet (redes sociais, aplicativos de mensagens, etc), é a câmera fotográfica que está evoluindo constantemente devido ao seu grande uso. Ao longo dos últimos anos, as câmeras digitais, limitadas apenas a esta função, foram sendo substituídas pelos smartphones, muito mais gerais em uso e presentes no nosso dia-a-dia.

A popularidade dos *smartphones* cresceu consideravelmente graças a integração do mesmo com redes sociais. Estamos conectados praticamente o tempo inteiro, acompanhando o Facebook, Twitter, Instagram - esta última voltada diretamente para postagem de fotos - sites de notícias e muitos outros. A grande evolução das câmera nos *smartphones* se deve ao aumento das exigências dos consumidores. Apesar desta evolução, outros fatores influenciam a qualidade das fotografias capturadas, como estabilidade do aparelho, luminosidade do ambiente, distância da cena, etc.

Nas câmeras digitais convencionais (i.e., DSRL e *point-and shoot*) diversos acessórios podem ser usados para tentar garantir uma maior qualidade das imagens capturadas, como suportes estabilizadores para evitar tremidos na foto. Porém, mesmo que alguns destes acessórios tenham versões para *smartphones*, não andamos com eles o tempo todo, e muitas vezes temos apenas uma única oportunidade para eternizar aquela paisagem ou aquele reencontro com os amigos. Por outro lado, *smartphones* são computadores portáteis e podem executar aplicações para melhorar a qualidade das fotos obtidas. Além disso, estes dispositivos incluem sensores de movimento, como acelerômetros, e sensores de rotação, como giroscópios. Estas ferramentas, aliadas às técnicas de filtragem de imagens, podem ser utilizadas para obter fotografias mais estáveis e satisfatórias.

Este trabalho descreve a implementação de um algoritmo de deconvolução não cega utilizando um *smartphone*. Idealmente, fazendo-se uso dos sensores presentes nos *smartphones*, pode-se estimar a PSF associada ao borramento introduzido durante a captura de uma fotografia. Isto permitiria a obtenção de deconvolução cega (*i.e.*, sem conhecimento prévio da PSF) diretamente no *smartphone*. Esta possibilidade porém, é deixada como trabalho futuro.

1.1 Estrutura do Texto

Este trabalho encontra-se organizado da seguinte forma: o Capítulo 2 descreve o algoritmo de deconvolução não cega proposto por Fortunato e Oliveira [2014] que foi utilizado como base no desenvolvimento deste trabalho. O Capítulo 3 apresenta a biblioteca OpenCV e suas funções necessárias para a realização deste trabalho. O Capítulo 4 apresenta a plataforma Android e aborda a implementação do algoritmo de deconvolução não cega realizada em um dispositivo móvel. O Capítulo 5 apresenta alguns dos resultados obtidos com a

implementação realizada. O Capítulo 6 discute uma tentativa de estimar PSFs utilizando os sensores de movimento e rotação presentes nos *smartphones*. Por fim, o Capítulo 7 apresenta as conclusões obtidas com a realização deste trabalho e sugere alternativas para trabalhos futuros..

2 DECONVOLUÇÃO NÃO CEGA

Neste capítulo será apresentada a técnica de deconvolução não cega usando *sparse adaptative priors* proposta por Fortunato e Oliveira [2014]. O objetivo desta técnica é dada uma imagem contendo borramento e o *kernel* que gerou tal borramento, tentar obter uma imagem próxima da imagem ideal (Fig. 2.1). O problema foi modelado como um sistema linear que é resolvido no domínio frequência.

Fig. 2.1 Exemplo de deconvolução não cega. (Esquerda) Imagem de entrada com o respectivo kernel que gerou o borramento. Extraído de Shan et al [2008] (Direita) Imagem resultante do processo de deconvolução não cega proposto por Fortunato e Oliveira [2014].

O processo de captura de imagem pode ser descrito como uma convolução entre a cena f e um kernel de convolução h, acrescidos de um ruído n.

$$g = h \otimes f + n \quad (1)$$

Para a abordagem proposta, a equação acima é expressa na forma matricial da seguinte forma:

$$g = hf + n \quad (2)$$

onde h_{mxn} é uma matriz quadrada e g, f e n são vetores coluna, de tamanho (m x n). De acordo com Fortunato e Oliveira [2014], o problema de deconvolução pode ser modelado por meio da seguinte função a ser minimizada:

$$\delta(f) = \|h * f - g\|^2 + \sum_{s=1}^{n} \lambda_s \|d_s f - w_s\|^2 \quad (3)$$

onde f, h e g têm o mesmo significado que na Equação (2). D_s, $s \in \{1,...,5\}$, representam as derivadas de primeira e segunda ordem. λ_s são pesos positivos e w_s são as derivadas esperadas para a imagem ideal f. Tendo τ como um limiar que representa algum nível de ruído, temos que para cada pixel, $w_s = 0$ se $| d_s f_{(i,j)} | < \tau$, caso contrário, $w_s = d_s f_{(i,j)}$.

Derivando a Equação (3) com relação a cada pixel e igualando-se as expressões resultantes a zero, chega-se a equação:

$$\left(h^T h + \sum_{s=1}^{5} \lambda_s d_s^T \right) \hat{f} = h^T g + \sum_{s=1}^{5} \lambda_s d_s^T w_s \quad (4)$$

que pode ser reescrita como:

$$a\hat{f} = b \quad (5)$$

onde:

$$a = h^T h + \sum_{s=1}^{5} \lambda_s d_s^T$$

$$b = h^T g \sum_{s=1}^{5} \lambda_s d_s^T w_s$$

Expressando essa equação, no domínio frequência, temos:

$$A \circ \hat{F} = B \quad (6)$$

onde:

$$A = H^* \circ H + \sum_{s=1}^{5} \lambda_s D_s^* \circ D_s \quad (7a)$$

$$B = H^* \circ G + \sum_{s=1}^{5} \lambda_s D_s^* \circ W_s \quad (7b)$$

Neste ponto, □* representa o complexo conjugado e ∘ é o operador de multiplicação ponto-a-ponto. A imagem f', que é nosso interesse, pode ser expressa através da fórmula

$$\hat{f} = FFT^{-1}(B./A) \quad (8)$$

sendo ./ o operador de divisão ponto-a-ponto.

2.1 O Algoritmo

O algoritmo tem como entrada a imagem *f*, o kernel *h* e um vetor (sigma) que são os pesos de regularização. Todo o processo é realizado no domínio frequência, desta forma, a operação de deconvolução - inversa a convolução - é uma divisão ponto-a-ponto. Para isso, precisamos que tanto a imagem, quanto o kernel tenham as mesmas dimensões. Para que o kernel tenha as mesmas dimensões da imagem, é feita uma expansão (*padding*) e uma normalização. Para expandí-lo é definida uma matriz do tamanho da imagem de entrada, inicialmente preenchida com zeros. Após isto, o kernel é colocado no centro dessa matriz, substituindo os zeros que ali estavam. Por fim, a matriz é normalizada, divindindo cada elemento pela soma de todos os elementos da matriz. O algoritmo é dividido em 4 passos: o primeiro, uma deconvolução com prior Gaussiana ($w_s = 0$) para obter uma aproximação inicial da imagem a qual queremos; o segundo, uma filtragem com preservação de bordas

para reduzir o ruído, nos dando uma segunda aproximação para a imagem; no terceiro passo, calcula-se novos valores de w_s, com base nas derivadas da imagem obtida no segundo passo; por fim, realiza-se a deconvolução da imagem obtida no segundo passo com w_s calculado no terceiro passo (Fig 2.2).

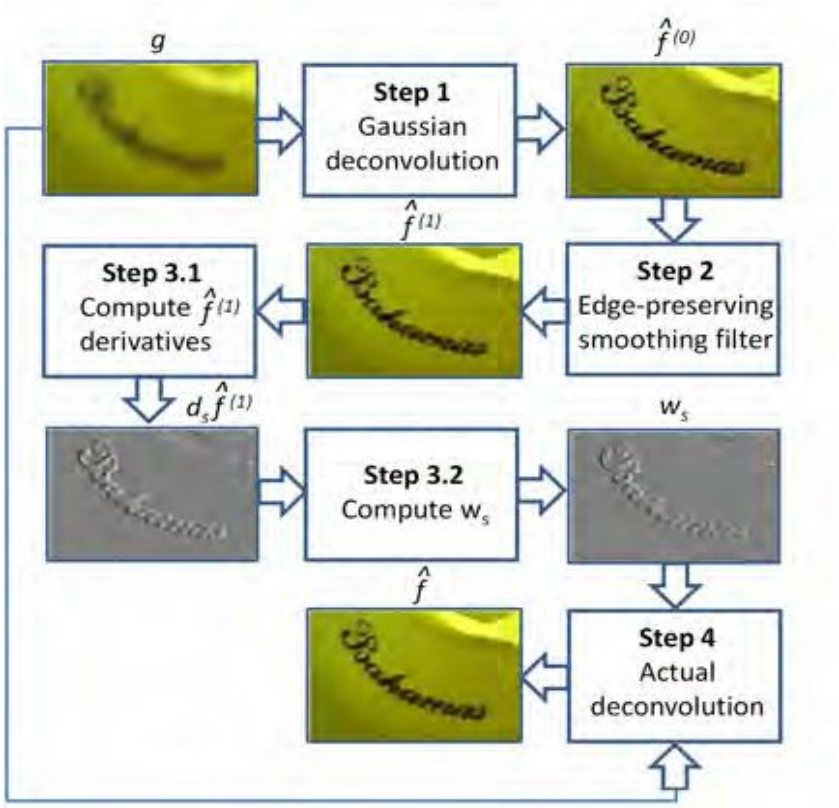

Fig. 2.2 Passo a passo do algoritmo: O primeiro passo obtem uma aproximação inicial para *f*, através de uma deconvolução utilizando $w_s = 0$ (deconvolução com prior Gaussiano). No segundo passo é realizada uma filtragem com preservação de bordas para redução do ruído preservando as principais bordas. No terceiro passo, calcula-se os valores de regularização (w_s) com base nas derivadas da imagem produzida no passo anterior. Por fim, no quarto passo, utiliza-se a deconvolução com os valores de w_s calculados no terceiro passo. Extraído de Fortunato e Oliveira [2014].

Quando uma imagem do domínio espacial é levada para o domínio frequência (através da transformada de Fourier, por exemplo), ela passa a ser um período de um sinal periódico. Ou seja, existem 'cópias' (outros períodos) da imagem ao lado da que está sendo analisada. Por conta disso, ao realizar a deconvolução no domínio frequência são introduzidos alguns artefatos (Fig 2.3) próximos as bordas da imagem.

Fig 2.3 Imagem deconvoluída sem moldura: Imagem deconvoluída sem o uso de moldura (*padding*), ilustrando a existência de anelamentos próximo as bordas das imagens. Extraída de Fortunato e Oliveira [2014].

A fim de minimizar os impactos destes artefatos é adicionada a imagem de entrada, antes do início do processo, uma moldura (Fig 2.4 esquerda). Essa moldura é criada através da introdução de 2*m* linhas antes da primeira e após a última linha da imagem, e de 2*m* colunas antes da primeira e após a última coluna da imagem. O valor de *m* é a maior dimensão do kernel. Após a realização desta operação de padding é feita uma suavização desta moldura, através da multiplicação da imagem emoldurada por uma máscara de suavização que tem uma transição gradual, de 1 pela região original da imagem, para quase 0 nas bordas da imagem emoldurada. A máscara tem as mesmas dimensões que a imagem possui após a emolduração e se assemelha a um filtro de Butterworth. Ela é definida pela equação:

$$mask(r,c) = \frac{1}{(1+(\frac{r-r_c}{R/2})^{2n_r})(1+(\frac{c-c_c}{C/2})^{2n_c})}$$

onde *r* e *c* são a linha e coluna de um determinado pixel na máscara, *rc* e *cc* são a linha e a coluna do centro da máscara. *R* e *C* são as dimensões da imagem de entrada, antes da emolduração e *nr* e *nc* são fatores para garantir a transição suave de 1 (por toda a imagem) para quase zero nas bordas da moldura:

$$n_r = [0.5log((1-\alpha)/\alpha)/log(r_c/(R/2))]$$
$$n_c = [0.5log((1-\alpha)/\alpha)/log(c_c/(C/2))]$$

onde α é o valor desejado para as bordas. O valor de α utilizado foi 0.01. Após a inserção da moldura a imagem, é feita a deconvolução e obtemos uma imagem mais limpa (Fig 2.5 centro). Porém, a moldura que foi anteriormente adicionada ainda está presente. Para a remoção da moldura, dividimos a imagem resultante da deconvolução pela máscara de suavização utilizada e recortando a imagem, removendo as linhas e colunas adicionadas pelo processo de emolduração, obtendo a imagem final (Fig 2.5 direita).

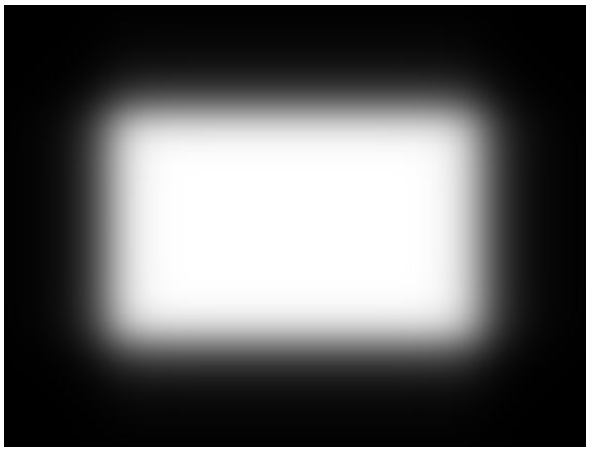

Fig 2.4 Exemplo de máscara de suavização: Exemplo de máscara de suavização que é utilizada no algoritmo de deconvolução.

Fig 2.5 Deconvolução utilizando moldura: (Esq.) Imagem de entrada após a introdução da moldura, para que a transição entre um período e outro ocorra de forma suave. (Centro) Imagem deconvoluída com a moldura. (Dir.) Imagem deconvoluída com a moldura retirada. Extraído de Fortunato e Oliveira [2014].

2.1.1 Primeiro Passo (Deconvolução utilizando prior Gaussiana)

Nesta etapa, são utilizadas as equações (7a, 7b e 8), com $w_s = 0$, gerando uma primeira aproximação, $f^{(0)}$ (Fig 3.6 centro-esquerda). Esta primeira estimativa, embora contenha artefatos indesejáveis, nos oferece uma boa noção das bordas da imagem;

2.1.2 Segundo Passo (Filtragem com preservação de bordas)

Neste passo, realizamos a filtragem com preservação de bordas na imagem obtida no primeiro passo ($f^{(0)}$), utilizando a *Domain Transform* proposta por Gastal e Oliveira [2011]. Com isso, reduzimos o efeito do ruído presente em $f^{(0)}$, preservando as bordas importantes, gerando uma nova aproximação $f^{(1)}$ (Fig 2.6 centro direita).

2.1.3 Terceiro Passo

Neste passo, calculamos novos valores para w_s, com base nas derivadas primeira e segunda da imagem $f^{(1)}$ obtida no passo anterior. Estes novos valores de w_s são obtidos através da equação $w_s = 0$ se $| d_s f_{(i,j)} | < \tau$, caso contrário, $w_s = d_s f_{(i,j)}$.

2.1.4 Quarto Passo (Deconvolução)

Neste último passo, realizamos a mesma convolução utilizada no primeiro passo. Porém, ao invés de utilizarmos $w_s = 0$, utilizamos os valores de w_s calculados no terceiro passo. Com isso obtemos a imagem deconvoluída f, que é o nosso objetivo (Fig 2.6 direita)

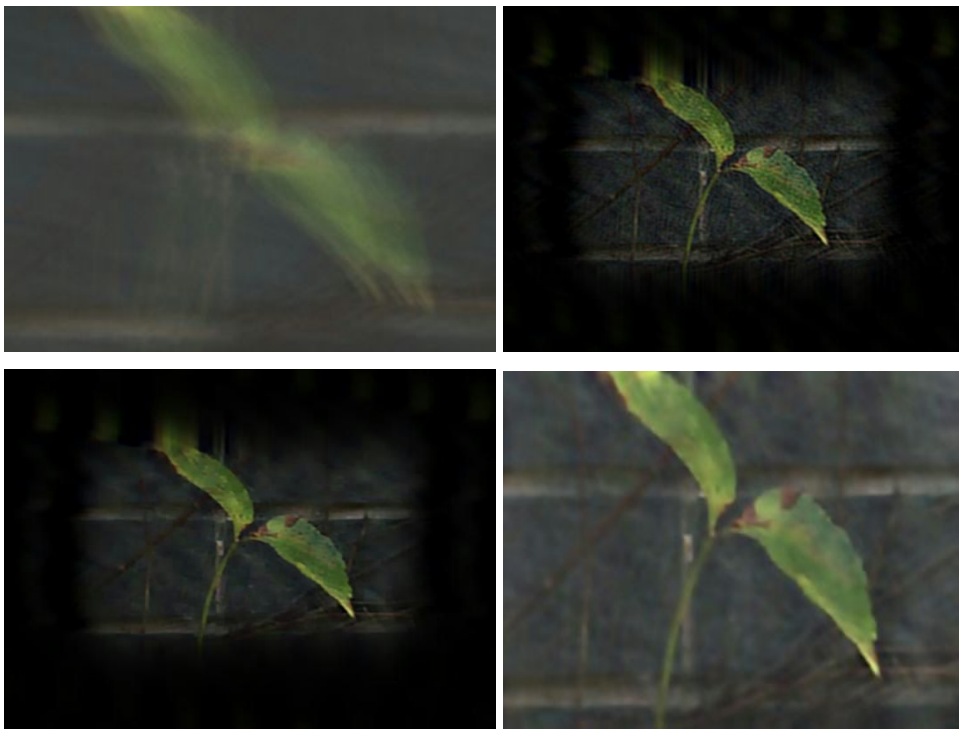

Fig 2.6 Ilustração do passo a passo do algoritmo: A primeira imagem (Superior, a esquerda) é a imagem borrada usada como entrada. A segunda imagem (Superior, a direita), é resultado do primeiro passo do algoritmo, uma deconvolução com prior Gaussiano. A terceira imagem (Inferior, a esquerda), é após a filtragem com preservação de bordas. Por fim, a quarta imagem (Inferior, a direita) é o resultado final do algoritmo. Imagem de entrada extraída de Shan et al [2008].

2.2 Resumo

Neste capítulo foram abordados as etapas do algoritmo de deconvolução proposto por Fortunato e Oliveira [2014] em cada um de seus passos. Também foi abordada a necessidade de criar uma moldura para a imagem e como ela é inserida na imagem. O uso de tal moldura é necessário devido as operações serem feitas no domínio frequência, para evitar a introdução de artefatos próximos às bordas da imagem.

3 OPENCV

Este capítulo apresenta uma breve introdução à biblioteca OpenCV. São descritos os principais pacotes e funções utilizados para a realização deste trabalho.

A OpenCV (*Open Source Computer Vision Library*) é uma biblioteca de código aberto, inicialmente desenvolvida pela Intel no ano 2000, que possui módulos para o processamento de imagem (nosso interesse), álgebra linear, entre outros. Foi desenvolvida em C/C++, mas possui suporte a outras linguagens como Java e Python. A integração entre OpenCV e o código para Android é feita graças ao suporte a Java que a biblioteca oferece, além de ter uma versão compilada para Android. Com ela, podemos calcular histogramas, manipular a câmera do *smartphone*, mudar o esquema de cores de imagens, etc. Para a realização deste trabalho foi utilizada a versão 2.4.3 para Android.

3.1 Pacotes da Biblioteca OpenCV

OpenCV encontra-se dividida em vários pacotes (bibliotecas menores), separados de acordo com suas finalidades. Os principais utilizados neste trabalho são detalhados nas Seções 3.1 a 3.4. Outros pacotes também estão disponíveis, como os pacotes *Photo* e *Video*, para algumas operações mais específicas no tratamento de fotos e vídeos. Além destes, também existem os pacotes de *Features* e de *Utilities*. Este último, possui uma série de conversores entre objetos da OpenCV e objetos do Java.

Para que seja possível utilizar código que faz uso da biblioteca em um dispositivo com Android, o mesmo deve contar com um aplicativo extra instalado: o *OpenCV Manager*. É através deste aplicativo que todos os pacotes, sistema de tipos e funções se tornam disponíveis para outras aplicações que utilizem funções da biblioteca OpenCV.

3.1.1 Pacote Android

Neste pacote encontram-se as funções de inicialização da biblioteca no dispositivo e de utilidades. Para que seja possível a utilização das funções e objetos da OpenCV, é preciso inicializá-la, e ter uma função de *callback* (função chamada quando da ocorrência de um determinado evento). Sem essa inicialização prévia, a aplicação não será capaz de reconhecer as chamadas de função e sistema de tipos que são nativos da biblioteca, impedindo o funcionamento correto da aplicação. Para isso, utiliza-se a classe OpenCVLoader, que carrega as informações da biblioteca, presentes no dispositivo, e com isso, seja possível identificar os objetos. A inicialização se dá através de uma chamada de função, que tem os seguintes parâmetros:

- Versão da OpenCV;
- Contexto da aplicação;
- Função de *callback* que foi criada pelo programador.

Um exemplo de chamada desta função é:

OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_3, this, mLoaderCallback);

Por sua vez, funções de utilidades presentes neste pacote são utilizadas para conversão de um objeto Java para um objeto OpenCV e vice-versa. Mais especificamente, de um *Bitmap* Java para um *Mat* OpenCV. Estas funções de conversão são importantes uma vez que o tratamento de imagens em Java é feito através de objetos *Bitmap*, enquanto OpenCV trata uma imagem como um objeto do tipo *Mat*.

3.1.2 Pacote Core

No pacote Core encontram-se as principais funções para manipulação de matrízes que foram utilizadas neste trabalho: as operações ponto a ponto de soma/subtração e de multiplicação/divisão. Além disso, este pacote contém a definição da classe do tipo *Mat*, que é como uma imagem é tratada no OpenCV. Além disso, temos as definições dos tipos dos valores que podem ser utilizados dentro de um objeto do tipo *Mat*, como por exemplo: *CV_64FC3*, onde:

- CV é apenas um prefixo;
- 64 representa o número de bits que serão utilizados para representar o valor armazenado dentro do objeto, podendo assumir outros valores como 8, 16 e 32;
- F indica o tipo que, neste caso, é um float;
- Cx, onde x é o número de canais da matriz. Por exemplo, com x = 3 teremos 3 canais por valor, representando R,G e B.

Neste trabalho, foram utilizadas matrízes do tipo *CV_64FC4* para a imagem (4 canais: R, G, B e *Alpha*) e *CV_64FC1* para o kernel (1 único canal: imagem em tons de cinza). Além disso, é neste pacote que encontra-se a transformada discreta de Fourier, bem como sua transformada inversa. Elas são fundamentais para este trabalho, uma vez que todas as operações são realizadas no domínio frequência. A transformada discreta de Fourier, assim como sua inversa, tem os parâmetros:

- A matriz de origem;
- A matriz de destino;
- Um parâmetro de opções, não obrigatório;
- Número de linhas não zeradas, não obrigatório.

Um exemplo de opções que podem ser utilizadas como parâmetro é *DFT_COMPLEX_OUTPUT*, que faz com que o resultado seja uma matriz complexa com simetria do complexo-conjugado. Para aplicar mais de uma opção, basta realizar a soma das opções escolhidas.

3.1.3 Pacote Highgui

Neste pacote são encontradas as funções de leitura e escrita de imagens em memória do dispositivo e também em *buffer*. As funções de entrada e saída que operam na memória do dispositivo possuem exatamente a mesma estrutura que no MATLAB:

- ***imread***: recebe como parâmetro uma *string* que representa o nome da imagem e, como retorno, teremos a imagem representada por um objeto do tipo *Mat* para ser utilizado nas funções de manipulação de imagens;
- ***imwrite***: tem como parâmetros uma *string* para o nome do arquivo e um objeto do tipo *Mat*, e devolve um booleano que indica se o arquivo foi escrito com sucesso.

As funções que operam sobre *buffer* funcionam de maneira análoga, só que ao invés de utilizar um arquivo na memória do dispositivo, utilizam um objeto do tipo *Mat* como *buffer*. A função *imdecode* lê as informações de um *buffer* e as salva em uma matriz (*i.e*., em outro objeto do tipo *Mat*). Por sua vez, a função *imencode* escreve as informações de uma matriz em um *buffer* retornando um booleano para indicar se a escrita ocorreu com sucesso.

3.1.4 Pacote Imgproc

Neste pacote temos as funções de manipulação de imagens como funções de filtragem, criação de bordas, operações sobre histogramas (cálculo, comparação, equalização, etc), transformações geométricas, entre outras. Para este trabalho, o pacote *Imgproc* foi utilizado para obter as funções de filtragem e de criação de bordas. A função de filtragem utilizada foi a ***filter2D*** que realiza uma convolução entre a matriz da imagem e o *kernel*, e utiliza os seguintes parâmetros:

- A matriz da imagem de origem;
- A matriz de destino, onde o resultado será armazenado;
- A profundidade da matriz resultante. Normalmente utiliza-se o valor -1, para indicar que a matriz de saída deve ter a mesma profundidade da matriz de entrada;
- O kernel que será utilizado na convolução;
- Um ponto (x,y) que indica a posição do pixel filtrado dentro do kernel, normalmente usa-se (-1,-1) para indicar que a posição é o centro do kernel;
- Um valor a ser somado ao valor do pixel, após o mesmo ser armazenado na matriz de destino;
- Tipo de borda. Utilizou-se borda constante.

Exemplo de uso:

```
Imgproc.filter2D(image, output, -1, kernel, new Point(-1, -1), 0, Imgproc.BORDER_CONSTANT);
```

Por sua vez, a função de criação de bordas tem como parâmetro:

- A matriz da imagem a qual deseja-se adicionar bordas;
- Uma matriz de saída, que tenha as dimensões da imagem já com as bordas;
- O tamanho de cada borda (topo, esquerda, direita e baixo);
- Como essa borda será preenchida, se será através de replicação de uma linha/coluna ou se será preenchida com um valor constante.

Exemplo de uso:

Imgproc.copyMakeBorder(image, paddingImage, padSizeTop, padSizeBottom, padSizeLeft, padSizeRight, Imgproc.BORDER_REPLICATE);

3.2 Resumo

Este capítulo apresentou uma breve introdução à biblioteca OpenCV, utilizada para realização deste trabalho. Esta escolha se deveu ao fato de que suas funções podem ser associadas quase diretamente com as respectivas funções do MATLAB. Foram apresentados, também, os principais pacotes, com as respectivas funções utilizadas.

4 SISTEMA DESENVOLVIDO

Este capítulo apresenta a plataforma de desenvolvimento utilizada, o sistema operacional Android, bem como descreve a aplicação desenvolvida. São discutidos os principais aspectos do sistema, como ambiente de programação, interação com o usuário, etc.

4.1 Android

O sistema operacional Android é baseado no *kernel* do Linux, e atualmente distribuído pela Google, voltado principalmente, para dispositivos móveis como *smartphones*, *tablets*, relógios de pulso, entre outros. Sua interação com o usuário é baseada em manipulação direta, através de telas sensíveis ao toque. Embora seja muito conhecido por estar presente em *smartphones*, existem versões do Android em videogames, câmeras fotográficas e até mesmo em computadores.

O Android executa uma máquina virtual baseada em registradores, a *Dalvik*. Projetada e escrita por Dan Bornstein em conjunto com outros engenheiros da própria Google, a Dalvik executa bytecodes. Muitos a consideram uma Máquina Virtual Java (JVM) como a que temos em nossos computadores, embora isso não seja totalmente verdade, uma vez que o bytecode executado pela Dalvik é diferente do executado pela JVM. Ou seja, o *bytecode* Java é traduzido para *bytecode* Dalvik, dentro do dispositivo. Em 2014, a Google anunciou a ART (*Android Runtime*), uma remodelagem da máquina virtual Dalvik, que chegou aos dispositivos como uma opção na versão 4.4 (KitKat) do sistema operacional e se tornou padrão a partir da versão 5.0 (Lollipop). A principal diferença entre as máquinas virtuais são que a Dalvik trabalha com o sistema *Just-in-Time* (JIT) que é a tradução e compilação do código em tempo de execução, enquanto a ART utiliza o sistema *Ahead-of-Time* (AOT). No sistema AOT os códigos da aplicação são pré-compilados em tempo de instalação na linguagem de execução, afim de se evitar que, a cada execução, o código do aplicativo tenha que ser recompilado.

O desenvolvimento de aplicativos para este sistema operacional é feito em Java, independentemente de qual máquina virtual o dispositivo esteja usando, com uso do *Software Development Kit* (SDK) que é distribuído gratuitamente, junto com uma IDE para desenvolvimento. Atualmente é possível escrever aplicações em Java que fazem uso de bibliotecas e/ou trechos de códigos em outras linguagens, como C/C++ por exemplo. Isto é possível graças a *Java Native Interface* (JNI) que permite que um código Java, sendo executado em uma máquina virtual, invoque ou seja invocado por aplicações/códigos nativo.

Neste trabalho foi utilizado um *smartphone* com a versão 5.0.2 do Android (Lollipop), que utiliza a máquina virtual ART, porém, sem utilização da JNI. A interface com o usuário é descrita através de um arquivo XML (*eXtensible Markup Language*) que pode ser editado de forma gráfica, escolhendo os componentes que farão parte da interface e os posicionando na tela. Para uso destes componentes, eles devem ser associados a objetos

dentro do código escrito em Java, onde são definidas as ações que irão acontecer ao se interagir com algum deles.

4.2 Aplicativo Desenvolvido

O aplicativo desenvolvido para este trabalho implementa o algoritmo de deconvolução não cega proposto por Fortunato e Oliveira [2014], descrito no Capítulo 2. Para isso, realizou-se a tradução do código disponibilizado pelos autores, originalmente em MATLAB, para Java com uso da biblioteca OpenCV (Capítulo 3). O fluxo de execução do aplicativo pode ser dividido em quatro etapas principais: a primeira, a inicialização do sistema e carregamento da OpenCV; a segunda, a obtenção da imagem e do *kernel* que corresponde à *point spread function* (PSF) associada ao borramento observado na imagem de entrada; a terceira, consiste na execução da deconvolução não cega propriamente dita; e por fim, a exibição do resultado.

4.2.1 Primeira Etapa: Inicialização

Esta etapa ocorre ao abrir o aplicativo, onde é criada a interface do aplicativo, juntamente com o carregamento da OpenCV, para que seja possível utilizar a biblioteca. Esse carregamento se faz através da classe *OpenCVLoader* descrita no capítulo anterior. Além disso, são definidos os métodos que serão disparados de acordo com a interação do usuário com a interface. Estes métodos são os *listeners*, funções que aguardam a ocorrência de determinados eventos, como por exemplo o *onClickListener* que aguarda um toque do usuário em um determinado objeto para ser disparado.

4.2.2 Segunda Etapa: Obtenção da imagem e do *kernel*

Para a obtenção da imagem existe um botão na interface. Ao clicar neste botão, será criado um *Intent* que irá abrir a galeria de fotos do dispositivo. Um *Intent* é uma descrição abstrata de uma ação a ser realizada. Ao se disparar um *Intent* para uma determinada ação (selecionar uma imagem, por exemplo), é exibida uma lista de aplicações que são capazes de realizar a ação solicitada, e cabe ao usuário escolher qual será utilizada. Após a imagem ter sido escolhida, ela é carregada para um objeto *Bitmap* para que esteja disponível para a etapa seguinte. De modo análogo, o *kernel* é escolhido e também carregado para um objeto *Bitmap*. Após seus carregamentos, tanto a imagem quanto o *kernel* são exibidos na interface, para que o usuário possa confirmar que aqueles são as imagens corretas. Tendo carregado a imagem e o *kernel*, libera-se o terceiro botão, que é o responsável por iniciar o processo de deconvolução.

4.2.3 Terceira Etapa: Deconvolução

Ao clicar no botão que inicia o processo de deconvolução, é chamada a função que executa o algoritmo descrito no Capítulo 3, com todas as suas etapas. Essa função tem como parâmetro dois objetos do tipo *Bitmap* (imagem e *kernel*) e retorna um outro *Bitmap* que é a imagem decnvoluída. Os *Bitmaps* providos como entrada são convertidos para objetos do tipo *Mat* do OpenCV e então iniciam-se as etapas do algoritmo, conforme apresentado anteriormente:

1. Criação de uma moldura (*padding*) e multiplicação da imagem de entrada por uma máscara;
2. Expansão do *kernel* para que tenha as mesmas dimensões da matriz da imagem, para que as operações possam ser realizadas no domínio frequência;
3. Aplicação de uma deconvolução à imagem resultante do Passo (1) utilizando prior Gaussiano (i.e., $w_s = 0$);
4. Filtragem da imagem obtida na etapa anterior utilizando um filtro passa-baixas com preservação de bordas baseado na Domain Transform proposta por Gastal e Oliveira [2011];
5. Obtenção dos valores de w_s com base nas derivadas da imagem resultante do passo anterior;
6. Aplicação de uma deconvolução à imagem resultante do Passo (1), desta vez, utilizando como priors os valores de w_s calculados na etapa anterior;
7. Remoção da moldura, *i.e.*, divisão pela máscara e recorte (*cropping*) da imagem resultante da deconvolução do passo anterior;

A matriz obtida no Passo (7) é convertida para um objeto do tipo *Bitmap*, o qual pode ser exibido e/ou salvo no dispositivo. A Figura 4.1 mostra a função que implementa o processo de deconvolução. A versão completa do código encontra-se no Anexo A.

```
/**
 * Run deconvolution of Mat attributes
 * @return deblurredImage bitmap
 */

public Bitmap run(Bitmap blurredImage, Bitmap kernelImage) {

    double wev[] = new double[] {0.001, 20, 0.033, 0.05};

    int KR = (int) Math.floor((this.smallKernel.rows() - 1) / 2);
    int KC = (int) Math.floor((this.smallKernel.cols() - 1) / 2);
    int padSize = 2 * Math.max(KR, KC);

    Mat maskPad = new Mat();

    this.setBlurredImage(blurredImage);
    this.setKernel(kernelImage);

    Mat paddingImage = this.padImage(this.blurredImage, padSize, maskPad);

    Mat bigKernel = this.getBigKernel(paddingImage.rows(), paddingImage.cols());

    Mat bifilterImage = this.bifilter(paddingImage, bigKernel, wev);

    bifilterImage = this.unpadImage(bifilterImage, maskPad, padSize);

    Mat aux = new Mat(paddingImage.size(),CvType.CV_8UC4);
    bifilterImage.convertTo(aux, CvType.CV_8UC4);
    Bitmap deblurredImage = Bitmap.createBitmap(aux.cols(), aux.rows(), Bitmap.Config.ARGB_8888);

    Utils.matToBitmap(aux, deblurredImage);

    return deblurredImage;

}
```

Fig 4.1 Ponto de entrada da deconvolução: A função mostrada na imagem é o ponto de entrada e de saída do método de deconvolução. A partir dela que são chamadas todas as demais funções necessárias para o processo.

4.2.4 Quarta Etapa: Exibição do resultado e escrita na memória do dispositivo

Antes da exibição, efetua-se a escrita do *Bitmap* na memória do celular e então, cria-se um novo *Intent* para exibir o resultado na tela do dispositivo. Este resultado também fica disponível para o usuário através da galeria de fotos ou qualquer outro aplicativo que gerencie as fotos/imagens do dispositivo. Depois disso, é possível retornar ao aplicativo e iniciar uma nova operação de deconvolução, escolhendo uma nova imagem e/ou um novo *kernel*.

4.3 Resumo

Este capítulo apresentou uma visão geral do funcionamento do sistema operacional Android, e descreveu as etapas de funcionamento do aplicativo desenvolvido para executar a deconvolução não cega proposta por Fortunato e Oliveira [2014].

5 RESULTADOS

Neste capítulo serão apresentados os resultados obtidos com a implementação deste trabalho. Foram realizados testes com imagens e *PSFs* utilizadas por Fortunato e Oliveira [2014], do modo compararar os resultados obtidos. Também foram realizados testes adicionais com imagens obtidas na Internet. Em todos os casos, os resultados obtidos com o aplicativo desenvolvido foram comparadas com os resultados gerados pelo código no MATLAB fornecido pelos autores. Para os resultados aqui descritos foi utilizado um *smartphone* Motorola XT1069 (Moto G 2014) com sistema operacional Android 5.0.2 (*Lollipop*) com 1GB de memória. Os testes em MATLAB foram realizado em um notebook com CPU Intel Core i5-3337U 1.80 GHz e 8 GB de memória.

A Figura 5.1 mostra uma fotografia borrada e a PSF correspondente do processo de captura. A Figura 5.2 compara os resultados obtidos com a deconvolução da imagem mostrada na Figura 5.1 utilizando a implentação em *smartphone* descrita neste trabalho e com o código MATLAB disponbilizado por Fortunato e Oliveira [2014]. Como pode-se notar, de fato houve uma melhora significativa ao se executar a deconvolução no *smartphone*. Porém, um resultado bem mais satisfatório foi obtido com o código em MATLAB. As causas exatas das diferenças entre estes resultados não foi identificada até o momento da escrita desta monografia.

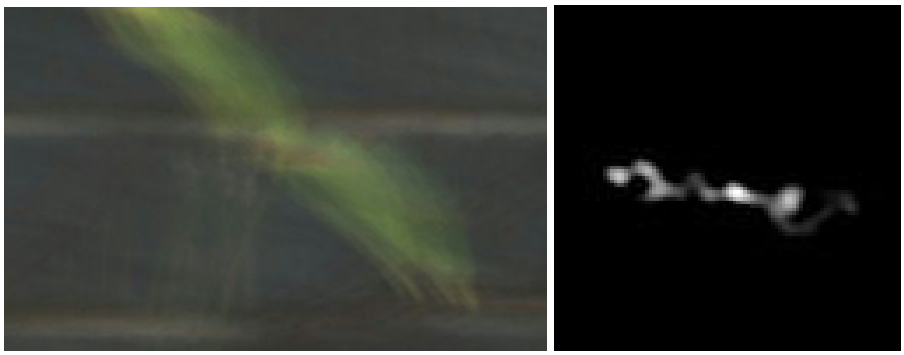

Fig 5.1 Entrada: Planta Imagem borrada de uma planta (esq.) com sua respectiva PSF (dir.). Extraída de Shan et al [2008].

Fig 5.2 Imagens de saída: Planta Imagem deconvoluída no smartphone (esq.) e utilizando o código MATLAB disponibilizado por Fortunato e Oliveira [2014] (dir.)

A Figura 5.3 mostra mostra outro par de fotografia borrada e PSF correspondente. De modo análogo, a Figura 5.4 compara os resultados de deconvolução obtidos utilizando a implentação em *smartphone* e utilizando o código MATLAB disponbilizado por Fortunato e Oliveira [2014]. Neste caso, também podemos notar que houve uma melhora significativa ao se executar a deconvolução no *smartphone*. Mas novamente, o resultado do MATLAB se mostrou mais satisfatório.

Fig 5.3 Entrada: Bonsai Imagem borrada de um Bonsai (esq.) com sua respectiva PSF (dir.). Extraída de Shan et al [2008].

Fig 5.4 Imagens de saída: Bonsai Imagem deconvoluída no celular (esq.) e no MATLAB (dir.)

As figuras 5.5 a 5.8 ilustram resultados de deconvolução aplicados à imagens borradas sinteticamente. Nestes exemplos, a fotografia da cena dos bonés foi borrada utilizando uma PSF gerada pelo autor (Figura 5.5). Já a foto da Lamborghini foi borrada utilizando uma PSF disponibilizada por Shan et al [2008]. No caso da Figura 5.6, fica visível a diferença entre os resultados da deconvolução feita no *smartphone* e utilizando o código MATLAB. Na imagem de saída obtida no *smartphone* é possível observar linhas verticais na imagem, que não estão presentes no resultado obtido através do MATLAB. As imagens mostradas na Figura 5.8 (Lamborghini) também exibem artefatos nos resultados obtidos com

a implmentação no *smartphone*. As causas precisas destes artefatos não foram identificadas até o momento da escrita da monografia.

Fig 5.5 Entrada: Bonés Imagem borrada de bonés pendurados em uma parede (esq.) com sua respectiva PSF (dir.). Imagem de entrada extraída de Kodak Lossless True Color Image Suite (disponível em http://r0k.us/graphics/kodak/), PSF gerada pelo autor.

Fig 5.6 Imagens de saída: Bonés Imagem deconvoluída no *smartphone* Android (esq.) e no MATLAB (dir.)

Fig 5.7 Entrada: Lamborghini Imagem borrada de um carro (esq.) com sua respectiva PSF (dir.). Imagem de entrada obtida no site Car and Drive, na reportagem sobre os 10 carros mais potentes do mundo, encontrada através de uma busca no Google por imagens de carros (disponível em

http://caranddriverbrasil.uol.com.br/noticias/mercado/veja-os-dez-carros-mais-potentes-do-mundo/6300), PSF extraída de Shan et al [2008].

Fig 5.8 Imagens de saída: Lamborghini Imagem deconvoluída no *smartphone* (esq.) e no MATLAB (dir.)

5.1 Discussão

Os artefatos observados nas imagens deconvoluídas no *smartphone* podem ser resultado de perda de informação ocorrida em virtude de conversões entre os tipos, representação interna incompatível ou erro durante o porte do algoritmo de MATLAB para Java.

Outro ponto que foi observado durante os testes, é uma limitação em relação ao tamanho das imagens que podem ser manipuladas no *smartphone* utilizado. Imagens muito grandes fazem com que o aplicativo não consiga realizar a deconvolução. Isto deve acontecer por limitação de *hardware* como, por exemplo, a quantidade de memória RAM do dispositivo utilizado nos experimentos (1 GB). Boa parte da RAM (aproximadamente 600 MB) é ocupada por processos em *background* e pelo sistema operacional, deixando pouca memória disponível para o aplicativo desenvolvido.

5.2 Resumo

Este capítulo apresentou alguns resultados obtidos com o aplicativo desenvolvido neste trabalho que realiza deconvolução não cega em um *smartphone*. Tais resultado foram comparados com os obtidos utilizando uma implmentação MATLAB do algoritmo fornecida pelos autores da técnica implementada. Os resultados obtidos com a versão MATLAB foram nitidamente superiores. As causas das diferenças não foram identificadas até o fechamento deste texto.

6 ESTIMATIVA DE PSF A PARTIR DE SENSORES DE MOVIMENTO E ROTAÇÃO

Este capítulo descreve uma tentativa de estimar diretamente uma PSF a partir de dados capturados pelos sensores de movimento (acelerômetro) e rotação (giroscópio) do *smartphone*. Isto permitiria a realização de deconvolução cega diretamente no dispositível móvel. O capítulo inicia com uma discussão dos trabalhos anteriores que utilizam sensores para estimar a PSF resultante do movimento da câmera durante a captura da foto (*camera shake*).

6.1 Camera Motion Tracking for Deblurring and Identification

Horstmeyer [2010] propôs uma solução para estimar a PSF associada a imagem capturada utilizando um microcontrolador Arduino acoplado a uma Canon Digital SLR Rebel XSI (Fig. 6.1). Ao Arduino foram conectados um acelerômetro de 3 eixos e um giroscópio de 2 eixos.

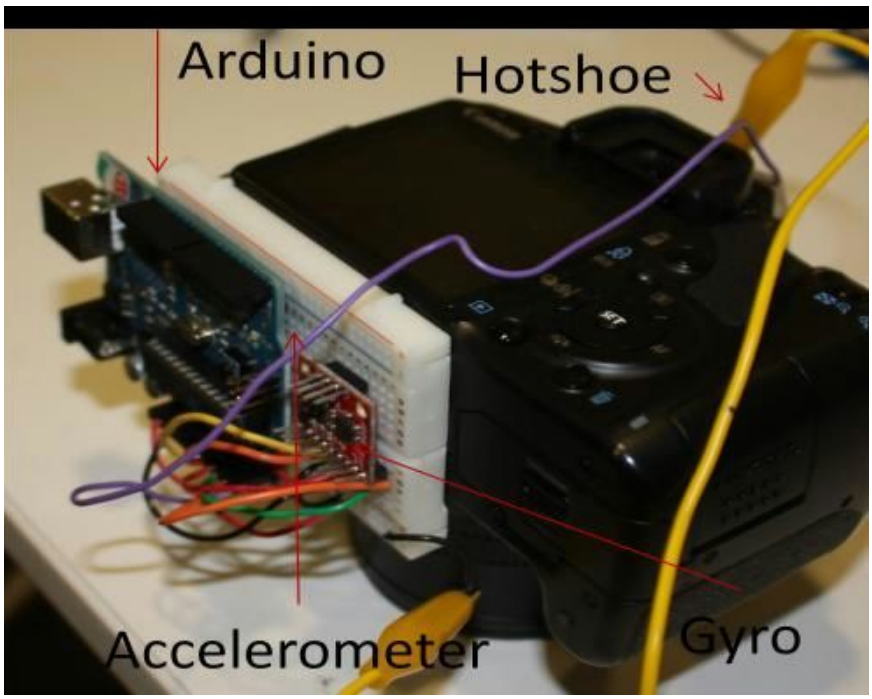

Fig. 6.1 *Setup* da câmera Na parte inferior da câmera, encontra-se a placa Arduino ligada aos sensores, conectados através de uma *protoboard*. Extraído de Horstmeyer [2010].

Um problema comum ao usar acelerômetros para medir posição é a amplificação de ruído, visto que para obter informações de posição a partir das informações do acelerômetro são necessárias duas integrações. Para minimizar o efeito deste ruído, Horstmeyer implementou um filtro passa baixas, em hardware, limitando a largura de banda passante em 50Hz. Em relação ao giroscópio, o ruído gerado por ele não é um problema tão significativo quanto o gerado pelo acelerômetro, uma vez, para o acelerômetro, realiza-se uma única integração. Para gerar a *Point Spread Function* (PSF - Fig 6.2), as informações dos sensores são combinadas através da função:

$$p = G_z(z, M, DOF) * (p_{mx}p_{my} + p_{rx}p_{ry})$$

onde:

- $G_z(z, M, DOF)$ é a rotação sobre o eixo óptico (z) dependente do *Depth of Field* (DOF);
- $[p_{mx}\, p_{my}]$ são os dados lidos do acelerômetro, nos eixos x e y;
- $[p_{r\theta x}\, p_{r\theta y}]$são os dados lidos do giroscópio, nos eixos x e y;

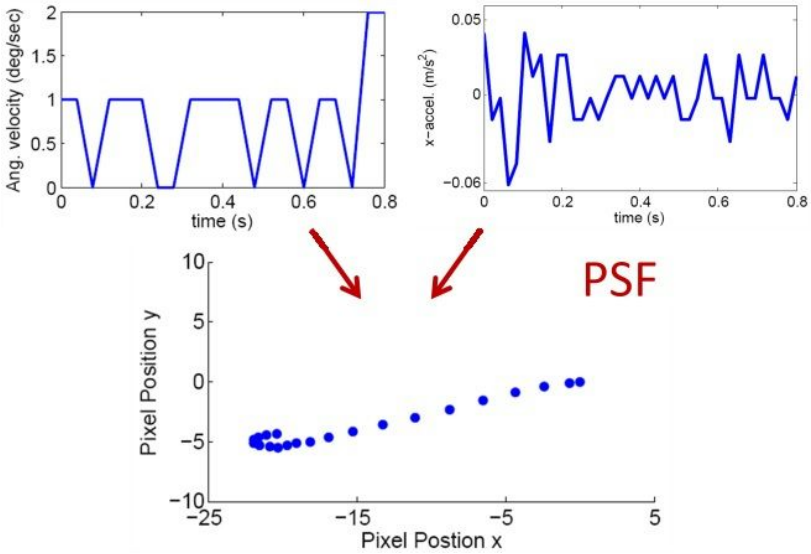

Fig 6.2 Geração da PSF Dados do giroscópio (topo, a esquerda), dados do acelerômetro (topo, a direita). Realizando a combinação destes dados, cria-se a estimativa de PSF (baixo). Extraído de Horstmeyer [2010].

Foram produzidas, em média, de 25 a 50 medições por exposição. Todas as medições realizadas pelos sensores são enviadas do Arduino para um computador através de um cabo USB, assim como as imagens capturadas. Todo o processamento das amostras e da imagem foi feito no MATLAB. Para realizar uma validação do *kernel* de borramento que foi estimado, foi incluído na cena um LED (Fig 6.3). Durante o processo de captura, o LED induz um rastro na imagem capturada pela câmera. Este rastro é comparado com o *kernel* que foi estimado.

Fig 6.3 Captura com um ponto de fonte de luz na cena Exemplo de imagem capturada com a inclusão de um LED para validar a estimativa feita para a PSF. Extraído de Horstmeyer [2010].

Para movimentos simples a estimativa gerada foi satisfatória, mas para movimentos mais complexos - tremidos, por exemplo - o movimento não foi mapeado corretamente (Fig. 6.4), dificultando o processo de deconvolução.

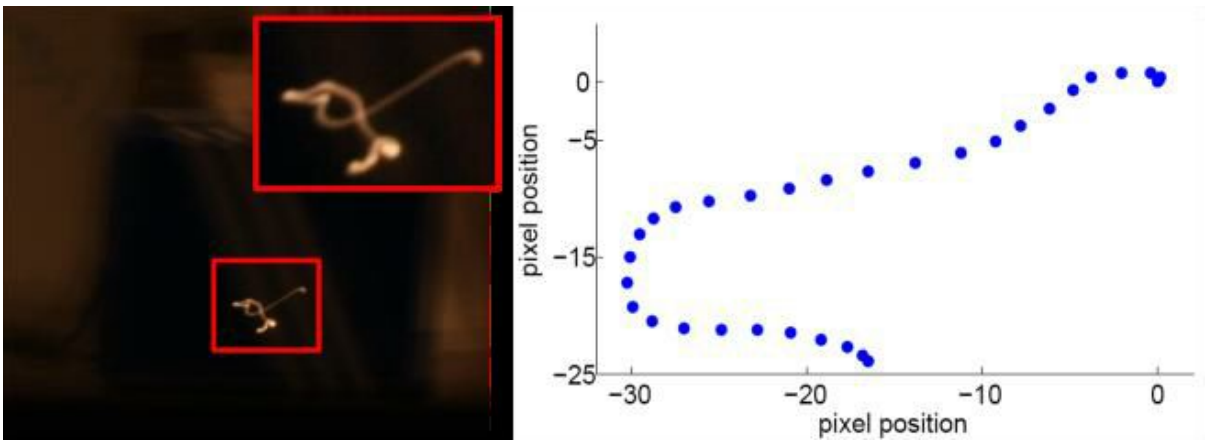

Fig 6.4 Movimentos complexos Exemplo de movimento mais complexo para o qual não foi possível estimar corretamente a PSF. Extraído de Horstmeyer [2010].

6.2 Image Deblurring using Inertial Measurement Sensors

Assim como Horstmeyer, Joshi et al. [2010] utilizaram uma solução em hardware para estimar a PSF que pode ser acoplada a qualquer câmera fotográfica. Foram utilizados 3 giroscópios de eixo único e um acelerômetro de 3 eixos, acoplados em uma placa Arduino (Fig. 6.5).

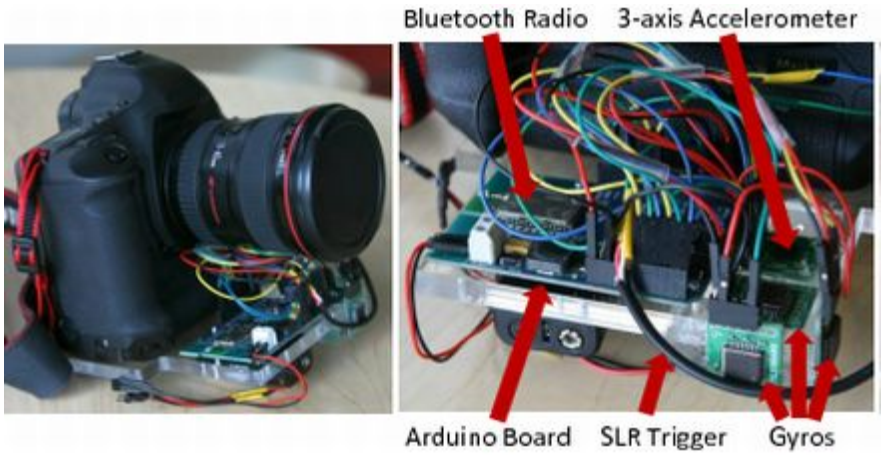

Fig. 6.5 *Setup* da câmera II À esquerda, uma visão da câmera com a placa e circuitos acoplados. À direita uma visão da placa Arduino com os sensores. Extraído de Joshi et al. [2010].

O que difere este trabalho do de Horstmeyer [2010], é que neste o borramento varia espacialmente, ou seja, uma única PSF não é suficiente para a imagem inteira, uma vez que cada região da imagem pode borrar de uma forma diferente após um movimento da câmera.

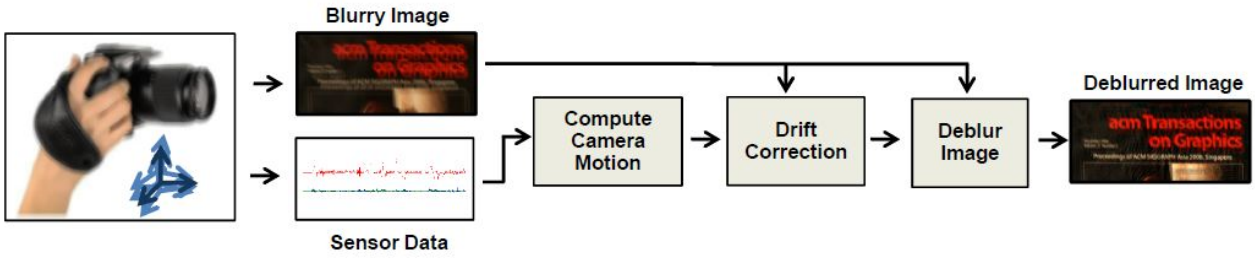

Fig. 6.6 Passo a Passo Os dados dos sensores são obtidos durante a captura da imagem. Logo após, é estimado o movimento da câmera, e realiza-se uma correção do ruído presente nos sensores. Após a correção é feita a deconvolução. Extraído de Joshi et al. [2010].

As informações de rotação e translação da câmera são recuperadas integrando a aceleração e velocidade angulares medidos em relação as coordenadas do início da captura. Isto nos fornece os valores ao longo do tempo, que são utilizados para calcular a PSF através da equação:

$$B = A(d)I + N$$

onde:

- B é a imagem borrada;
- I é a cena que está sendo capturada;
- N representa o ruído adicionado;
- A(d) é a matríz de borramento, em relação a distância d.

A matríz A(d) é obtida através da integral descrita abaixo:

$$\int_0^s A_t(d)dt,$$

onde

$$A_t = K(R_t + \frac{1}{d}T_t N^T)K^{-1}$$

e R_t é a componente de rotação, T_t a componente de translação, K é a matríz intrínseca a câmera e N^T e 1/d correspondem a profundidade. Depois de estimar a movimentação da câmera, pode-se realizar a deconvolução através da seguinte equação:

$$I = \arg min_I \left\| B - \sum A_t I \right\|^2 / \sigma^2 + \lambda \left| \bigtriangledown I \right|^{0.8}$$

O modelo elaborado por Joshi et al. [2010] utiliza o *Sparse Gradient Prior* proposto por Levin et al. [2009], além de utilizar pesos ajustados pelo método de mínimos quadrados. Alguns exemplos de resultados obtidos podem ser vistos abaixo (Fig 6.7).

Fig. 6.7 Resultado obtido Imagem de entrada, borrada (esq.), imagem de saída, após a execução do método, que se utiliza de *kernels* que variam espacialmente. Por conta disso, há a possibilidade de se encontrar mais de uma PSF para uma determinada imagem. Extraído de Joshi et al. [2010].

6.3 Estimativa de PSF Utilizando um *smartphone*

Diferentemente dos trabalhos mencionados, que utilizam um *hardware* adicional com sensores para estimar o movimento realizado pela câmera, a ideia é utilizar os sensores disponíveis no próprio *smartphone* para realizar a estimativa da PSF. Para isso, é necessário que o aplicativo seja capaz de utilizar a câmera do aparelho para captura de imagens, e ler os dados dos sensores - acelerômetro, por exemplo - em paralelo a captura. No momento em que a captura da imagem é iniciada, cria-se uma tarefa assíncrona, que monitora o acelerômetro (e quaisquer outros sensores). Ao fim da captura, esta tarefa realiza a estimativa de deslocamento do dispositivo durante o intervalo de tempo no qual esteve ativa. Para que seja possível ter uma estimativa mais precisa, é necessário remover a influência da aceleração da gravidade nos valores medidos do sensor. A gravidade influencia, de forma não uniforme o valor da aceleração nos 3 eixos do sistema de coordenadas (Fig 6.8), além de não garantir que os valores lidos do sensor sejam nulos quando o dispositivo estiver completamente parado. Após remover a influência da gravidade dos valores lidos do acelerômetro, podemos estimar o movimento realizado pelo *smartphone*, que pode ser descrito como:

$$s(t) = \iint_{t_0}^{t} a(t)dt$$

onde:

- a(t) é a aceleração medida no tempo t;
- s(t) é o deslocamento estimado no intervalo de tempo (t-t_0).

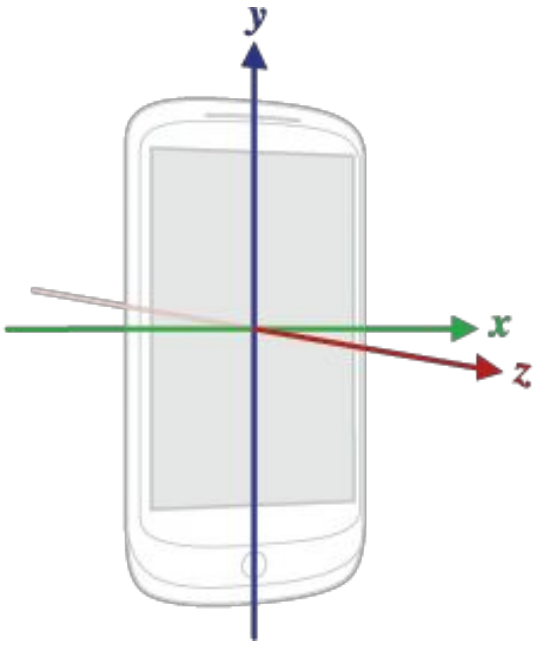

Fig 6.8 Sistema de coordenadas do acelerômetro: Mantendo o celular na vertical, conforme a ilustração, temos que o eixo Y tem valores positivos para cima, o eixo X para a direita e o eixo Z na direção do usuário.

Este cálculo é feito para os eixos *x* e *y*, resultando em uma lista de pontos (x,y) que é plotada em um gráfico. Este gráfico representa a estimativa de movimento realizado pelo dispositivo no plano XY (Fig. 6.9). Porém, este gráfico não pode ser utilizado diretamente como PSF, visto que o movimento realizado pelo dispositivo pode não representar diretamente o movimento realizado pelos *pixels* da imagem. Para compreender qual a relação entre o movimento do dispositivo e o fluxo ótico (movimento dos *pixels* da imagem) adicionamos à cena um LED e realizamos a captura da cena. Após isto, estimamos a movimentação com base nos dados do acelerômetro e a comparamos com a movidmentação/projeção do LED, para se estabelecer uma relação entre eles.

Porém, durante os experimentos, foi encontrado outro problema em nossas tentativas de gerar a estimativa da PSF: o tempo de amostragem dos sensores. Por vezes, não eram capturados dados do sensor, ou por que eles não estarem prontos para leitura e o *hardware* não permitiu o acesso aos dados, ou por que o tempo de captura da imagem foi rápido o suficiente para não dar tempo de realizar a leitura do sensor. Em outras tentativas, eram capturadas algumas amostras. Essa inconsistência verificada na leitura dos dados do sensor impossibilitou que fosse possível ter uma estimativa de PSF já neste trabalho. Como trabalho futuro, pretendemos obter uma boa estimativa de PSF. Quando esta estimativa estiver pronta, o código que estimará a PSF poderá ser integrado ao que realiza a deconvolução, se tornando um único aplicativo e dando origem a uma deconvolução cega sendo realizada em um dispositivo móvel.

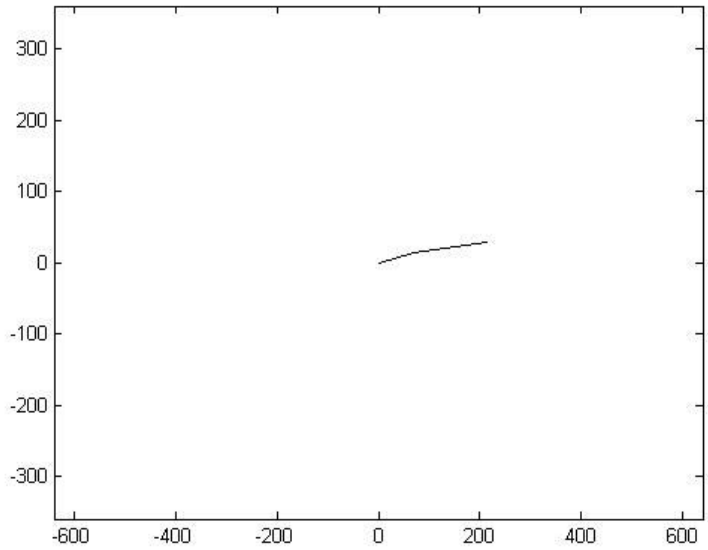

Fig 6.9 Estimativa de Movimento Gráfico gerado a partir das estimativas de movimento realizado pelo dispositivo durante a captura de uma cena. Eixo X na horizontal, eixo Y na vertical.

6.3 Resumo

Este capítulo descreveu uma tentativa de estimar a PSF a partir de dados obtidos dos sensores do *smartphone*. Também foram descritas duas propostas para estimar PSF conectando sensores a uma câmera DSLR através de um microcontrolador. Ao contrário destes trabalhos, a abordagem utilizada nesta monografia tentou fazer uso do acelerômetro disponível nos *smartphones*.

7 CONCLUSÃO

Neste trabalho foi realizada a implementação da abordagem proposta por Fortunato e Oliveira [2014] para uma deconvolução não cega de alta qualidade, em um *smartphone*. A partir dos resultados gerados e comparados com a execução da implementação original em MATLAB, é possível estabelecermos algumas considerações.

As imagens obtidas com a implementação feita para o *smartphone* tiveram uma qualidade inferior, mas ainda satisfatória. A possível causa dessa qualidade inferior pode estar em algum detalhe que tenha passado despercebido durante o pote do código de MATLAB para Java. Ao se executar a técnica no dispositivo, foram encontradas certas restrições, como por exemplo, limite de tamanho para a imagem de entrada. Embora os recursos presentes nos *smartphones* atuais sejam consideravelmente bons, ainda não são suficientes para nos oferecer condições para operar sobre imagens de tamanho médio e grande, pois ao tentar executar com uma imagem de tamanho médio (1280x720), por exemplo, o aplicativo cancela sua execução, ou mostra a mensagem de "Sem Resposta". Este problema acontece, normalmente, por falta de memória para a execução, ou por ter estourado a pilha de chamadas de função, sendo a primeira a mais provável.

Em relação a tentativa de estimar a *point spread function* associada a imagem durante a captura, foram encontradas dificuldades para interpretar corretamente como os valores do sensor influenciam o borramento na imagem. Além disso, também foram encontradas dificuldades na leitura dos sensores, pois em alguns casos, nenhum valor era lido. Estas dificuldades acabaram inviabilizando a implementação da estimativa, mas forneceram uma base para implementações futuras.

REFERÊNCIAS

LEVIN, A., FERGUS, R., DURAND, F., AND FREEMAN, W. T. 2007. Image and depth from a conventional camera with a coded aperture. **ACM Trans. Graph**. 26 (Julho), Article 70.

LEVIN, A., SAND, P., CHO, T. S., DURAND, F., AND FREEMAN, W. T. 2008. Motion-invariant photography. **ACM Trans. Graph.** 27 (Agosto), 71:1–71:9.

LEVIN, A., WEISS, Y., DURAND, F., AND FREEMAN, W. 2009. Understanding and evaluating blind deconvolution algorithms. **In Computer Vision and Pattern Recognition**, 2009. CVPR 2009. IEEE Conference on, IEEE Computer Society, 1964–1971.

JOSHI, N., KANG, S., ZITNICK, C., SZELISKI, R. Image Deblurring using Inertial Measurement Sensors, **ACM Trans. Graph.** (Julho/2010)

FORTUNATO, Horacio E., OLIVEIRA, Manuel M. Fast high-quality non-blind deconvolution using sparse adaptive priors. **The Visual Computer.** Volume 30 (2014), Numbers 6-8, pp. 661-671.

GASTAL, Eduardo S. L., OLIVEIRA, Manuel M. Domain Transform for Edge-Aware Image and Video Processing. **ACM Transactions on Graphics.** Volume 30 (2011), Number 4, Proceedings of SIGGRAPH 2011, Article 69.

HORSTMEYER, R., Camera Motion Tracking for De-blurring and Identification. MIT Media Lab MAS 863 Final Project.

GOOGLE, Android Programming Guide and Documentation. Disponível na Internet: <http://developer.android.com/reference/packages.html>.

OPENCV DEV TEAM, OpenCV Documentation and API Reference. Disponível na Internet: <http://docs.opencv.org>.

SHAN, Q., JIA, J., AGARWALA, A. 2008. High-quality motion deblurring from a single image. ACM TOG 27.

ANEXO A <CÓDIGO CRIADO PARA A DECONVOLUÇÃO NÃO CEGA>

```
package com.example.deblur;

import android.graphics.Bitmap;
import android.graphics.Color;
import android.media.Image;
import android.renderscript.Sampler;
import android.util.Base64;

import org.opencv.core.Core;
import org.opencv.core.CvType;
import org.opencv.core.Mat;
import org.opencv.android.Utils;
import org.opencv.core.Point;
import org.opencv.core.Rect;
import org.opencv.core.Scalar;
import org.opencv.core.Size;
import org.opencv.imgproc.Imgproc;

import java.lang.reflect.Array;
import java.sql.SQLOutput;
import java.util.ArrayList;
import java.util.List;

public class Deconvolution {

    /**
     * Blured image
     */

    private Mat blurredImage = null;

    /**
     * Small kernel
     */

    private Mat smallKernel = null;

    /**
     * Big kernel
     */

    private Mat bigKernel = null;

    /**
     * Deblured image
     */

    private Mat deblurredImage = null;
```

```
    /**
     * Real and imaginary flags
     */

    private static int REAL_PART = 0;

    private static int IMAGINARY_PART = 1;

    private static int IMAGE_FORMAT = CvType.CV_64FC4;

    private static int KERNEL_FORMAT = CvType.CV_64FC1;

    private enum CONV2_TYPE {
        // Return the full convolution, including borders
        FULL,
        // Return only the part that corresponds to the original image
        SAME,
        // Return only the submatrix containing elements that were not influenced by the border
        VALID;
    }


    /**
     * Import blurred bitmap to Deconvolution class
     */
    public void setBlurredImage(Bitmap blurredImage) {


        //System.out.println("SETAR BLURRED");

        if(this.blurredImage == null)
            this.blurredImage = new Mat(blurredImage.getHeight(),blurredImage.getWidth(),IMAGE_FO
RMAT);

        Mat aux = new Mat(blurredImage.getHeight(),blurredImage.getWidth(),CvType.CV_8SC4);
        Utils.bitmapToMat(blurredImage, aux);
        aux.convertTo(this.blurredImage, IMAGE_FORMAT);

        //System.out.println("THIS.BLURREDIMAGE type = " + this.blurredImage.type());

        //System.out.println("BLURRED SETADO!");
    }

    /**
     * Export blurred bitmap to Deconvolution class
     */
    public Bitmap getBlurredImage() {

        Bitmap output = null;
```

```
        if(this.blurredImage != null) {
            output = Bitmap.createBitmap(this.blurredImage.width(),this.blurredImage.height(), Bitmap.C
onfig.ARGB_8888);
            Utils.matToBitmap(this.blurredImage, output);
        }

        return output;

    }


    /**
     * Import kernel bitmap to Deconvolution class
     */

    public void setKernel(Bitmap smallKernel) {

        //System.out.println("SETAR KERNEL");

        if(this.smallKernel == null)
            this.smallKernel = new Mat(smallKernel.getHeight(),smallKernel.getWidth(),KERNEL_FOR
MAT);

        Mat aux = new Mat(smallKernel.getHeight(),smallKernel.getWidth(),CvType.CV_8SC4);
        Mat smallKernelAllChannels = new Mat(smallKernel.getHeight(),smallKernel.getWidth(),IMA
GE_FORMAT);

        Utils.bitmapToMat(smallKernel, aux);
        aux.convertTo(smallKernelAllChannels, IMAGE_FORMAT);

        List<Mat> allChannels = new ArrayList<>();
        Core.split(smallKernelAllChannels, allChannels);

        allChannels.get(0).copyTo(this.smallKernel);

        //System.out.println("THIS.kernel type = " + this.smallKernel.type());

        //System.out.println("KERNEL SETADO!");

    }

    /**
     * Export kernel bitmap to Deconvolution class
     */
    public Bitmap getSmallKernel() {

        Bitmap output = null;

        if(this.smallKernel != null) {
```

```
            output = Bitmap.createBitmap(this.smallKernel.width(),this.smallKernel.height(), Bitmap.Con
fig.ARGB_8888);
            Utils.matToBitmap(this.smallKernel, output);
        }

        return output;

    }


    /**
     * Export deblurred bitmap image from Deconvolution class
     */

    public Bitmap getDeblurredImage() {
        Bitmap deblurredImage = null;

        Utils.matToBitmap(this.deblurredImage, deblurredImage);

        return deblurredImage;
    }


    /**
     * Devonvolve image
     */

    public Bitmap deconvolve(Bitmap blurredImage, Bitmap kernelImage) {

        this.setBlurredImage(blurredImage);
        this.setKernel(kernelImage);

        return this.run();

    }

    public Bitmap deconv(Mat im_in, Bitmap kernel){
        this.blurredImage = im_in;
        return this.run();
    }

            /*
        PRINT TESTS
        Mat aux = new Mat(bigKernel.size(),CvType.CV_8UC3);
        bigKernel.convertTo(aux, CvType.CV_8UC3);
        Bitmap deblurredImage = Bitmap.createBitmap(aux.cols(), aux.rows(), Bitmap.Config.RGB_565
);
        System.out.println("SAI 5!");
        Utils.matToBitmap(aux, deblurredImage);
        System.out.println("SAI 3!");
        */
```

```
    /**
     * Run deconvolution of Mat attributes
     * @return deblurredImage bitmap
     */

    public Bitmap run() {

        double wev[] = new double[] {0.001, 20, 0.033, 0.05};

        int KR = (int) Math.floor((this.smallKernel.rows() - 1) / 2);
        int KC = (int) Math.floor((this.smallKernel.cols() - 1) / 2);
        int padSize = 2 * Math.max(KR, KC);


        //System.out.println("Passei aqui!");
        // Generate padding image and big kernel
        Mat maskPad = new Mat();
        Mat paddingImage = this.padImage(this.blurredImage, padSize, maskPad);

        //Bitmap deblurredImage = Bitmap.createBitmap(paddingImage.width(), paddingImage.height(),
Bitmap.Config.RGB_565);
        //Utils.matToBitmap(paddingImage, deblurredImage);

        Mat bigKernel = this.getBigKernel(paddingImage.rows(), paddingImage.cols());

        Mat bifilterImage = this.bifilter(paddingImage, bigKernel, wev);

        bifilterImage = this.unpadImage(bifilterImage, maskPad, padSize);

        Mat aux = new Mat(paddingImage.size(),CvType.CV_8UC4);
        bifilterImage.convertTo(aux, CvType.CV_8UC4);
        Bitmap deblurredImage = Bitmap.createBitmap(aux.cols(), aux.rows(), Bitmap.Config.ARGB_8
888);
        //System.out.println("SAI 5!");
        Utils.matToBitmap(aux, deblurredImage);
        //System.out.println("SAI 3!");

        return deblurredImage;

    }

    /**
     * Pad image to remove border ringing atifacts (see section 4.1 of our paper)
     * @param padSize
     * @return padImage
     */

    private Mat padImage(Mat image, int padSize, Mat mask) {
```

```
        Mat paddingImage = new Mat();

        Imgproc.copyMakeBorder(image, paddingImage, padSize, padSize, padSize, padSize, Imgproc.
BORDER_REPLICATE);

        // Pensar em outra forma de fazer o gradiente
        int nRows = paddingImage.rows();
        int nColumns = paddingImage.cols();

        Mat X = new Mat(nRows,nColumns,KERNEL_FORMAT,new Scalar(0));
        Mat Y = new Mat(nRows,nColumns,KERNEL_FORMAT,new Scalar(0));
        Mat DX = new Mat();
        Mat DY = new Mat();

        // Meshgrid (Parallel programming)
        for(int j = 0; j < nColumns ; j++)
            X.put(0,j,j+1);
        for(int j = 0; j < nRows ; j++)
            Y.put(j,0,j+1);

        for(int i = 1; i < nRows ; i++)
            X.row(0).copyTo(X.row(i));
        for(int j = 0; j < nColumns ; j++)
            Y.col(0).copyTo(Y.col(j));

        // Derived
        Scalar X0 = new Scalar(1 + Math.floor(nColumns/2));
        Scalar Y0 = new Scalar(1 + Math.floor(nRows/2));

        Core.absdiff(X, X0, DX);
        Core.absdiff(Y, Y0, DY);

        Scalar C0 = new Scalar(X0.val[0] - padSize);
        Scalar R0 = new Scalar(Y0.val[0] - padSize);

        double alpha = 0.01;

        // force mask value at the borders aprox equal to alpha
        // this makes the transition smoother for large kernels
        int nx = (int) (Math.ceil(0.5 * Math.log((1-alpha)/alpha)) / Math.log(X0.val[0] / C0.val[0]));
        int ny = (int) (Math.ceil(0.5 * Math.log((1-alpha)/alpha)) / Math.log(Y0.val[0] / R0.val[0]));

        // Calc mX
        Mat mX = calcMx(DX, C0, nx);

        // Calc mY
        Mat mY = calcMy(DY, R0, ny);

        Mat mask0 = new Mat();
        Core.multiply(mX, mY, mask0);
```

```
    List<Mat> maskRGB = new ArrayList<>();

    for(int i = 0; i < paddingImage.channels(); i++)
        maskRGB.add(i, mask0);

    Core.merge(maskRGB, mask);
    Mat outPadding = new Mat();
    Core.multiply(paddingImage, mask, outPadding);

    return outPadding;


}


private Mat calcMx(Mat DX, Scalar C0, int nx) {

    Mat mX = new Mat();
    Core.divide(DX, C0, mX);
    //System.out.println("PASSEI AQUI 3!");
    Core.pow(mX, 2 * nx, mX);
    //System.out.println("PASSEI AQUI 4!");
    Core.add(mX, new Scalar(1), mX);
    //System.out.println("PASSEI AQUI 5!");
    Core.divide(1, mX, mX);
    //System.out.println("PASSEI AQUI 6!");

    return mX;

}

private Mat calcMy(Mat DY, Scalar R0, int ny) {

    Mat mY = new Mat();
    Core.divide(DY, R0, mY);
    //System.out.println("PASSEI AQUI 7!");
    Core.pow(mY, 2 * ny, mY);
    //System.out.println("PASSEI AQUI 8!");
    Core.add(mY, new Scalar(1), mY);
    //System.out.println("PASSEI AQUI 9!");
    Core.divide(1, mY, mY);
    //System.out.println("PASSEI AQUI 10!");


    return mY;
}


/**
 * Remove padding (see section 4.1 of our paper)
 * @param image
```

```
     * @return unpaddingImage
     */
    private Mat unpadImage(Mat image, Mat mask, int padSize) {

        Mat unpaddingImage = new Mat();

        Core.divide(image, mask, unpaddingImage);

        List<Mat> unpaddingList = new ArrayList<>();
        List<Mat> unpaddingList2 = new ArrayList<>();
        Core.split(unpaddingImage, unpaddingList);

        for(int i = 0; i < unpaddingList.size(); i++) {
            unpaddingList2.add(i,
                unpaddingList.get(i).submat(padSize,image.rows()-padSize-1,padSize,image.cols()-padSiz
e-1));
        }

        unpaddingImage = new Mat();
        Core.merge(unpaddingList2, unpaddingImage);

        return unpaddingImage;

    }

    /**
     * Generate big kernel
     * @param padRows
     * @param padColumns
     * @return bigKernel
     */
    private Mat getBigKernel(int padRows, int padColumns) {

        int RC = (int) Math.floor(padRows/2);
        int CC = (int) Math.floor(padColumns/2);

        int RF = this.smallKernel.rows();
        int CF = this.smallKernel.cols();

        int RCF = (int) Math.floor(RF/2);
        int CCF = (int) Math.floor(CF/2);

        //Mat bigKernel = new Mat(this.smallKernel.size(),KERNEL_FORMAT, new Scalar(0));
        Mat bigKernel = new Mat(padRows,padColumns,KERNEL_FORMAT, new Scalar(0));
        // Put the small_kernel in the center of big_kernel
        //this.smallKernel.convertTo(bigKernel, KERNEL_FORMAT);
        // top, buttom, left, right
        // row start, row end, column start, column end
        //Imgproc.copyMakeBorder(bigKernel, bigKernel, RC - RCF, RC - RCF, CC - CCF, CC - CCF,
Imgproc.BORDER_CONSTANT, new Scalar(0));
```

```
        this.smallKernel.copyTo(bigKernel.rowRange(RC - RCF, RC - RCF + RF).colRange(CC - CCF,
 CC - CCF + CF));

        this.ifftshift(bigKernel);

        Core.divide(bigKernel, Core.sumElems(bigKernel), bigKernel);

        return bigKernel;

    }

    /**
     * Apply the IFFTSHIFT - Inverse FFT shift.
     * For matrices, IFFTSHIFT(X) swaps the first and third quadrants and the second and fourth quad
rants.
     * @param image
     *
     *
     * http://docs.opencv.org/doc/tutorials/core/discrete_fourier_transform/discrete_fourier_transform.
html
     *
     */
    private void ifftshift(Mat image) {

        // If odd number of rows or columns
        image = new Mat(image, new Rect(0,0, image.cols() & -2, image.rows() & -2));

        int centerX = image.cols()/2;
        int centerY = image.rows()/2;

        Mat q1 = new Mat(image, new Rect(0,0,centerX,centerY));
        Mat q2 = new Mat(image, new Rect(centerX,0,centerX,centerY));
        Mat q3 = new Mat(image, new Rect(0,centerY,centerX,centerY));
        Mat q4 = new Mat(image, new Rect(centerX,centerY,centerX,centerY));

        Mat tmp = new Mat();
        q1.copyTo(tmp);
        q4.copyTo(q1);
        tmp.copyTo(q4);

        q2.copyTo(tmp);
        q3.copyTo(q2);
        tmp.copyTo(q3);

    }


    private Mat bifilter(Mat blurredImage, Mat kernel, double web[]) {

        int imageRows = blurredImage.rows();
        int imageCols = blurredImage.cols();
```

```
int imageNumChannels = blurredImage.channels();


// Get the complex pairs
Mat fft2Kernel = this.dft2(kernel);

Mat A0 = new Mat();
Mat conjFft2Kernel = this.complexConjugate(fft2Kernel);

 Core.mulSpectrums(conjFft2Kernel, fft2Kernel, A0, 0, false);

// Get real part of A0
A0 = this.splitComplex(A0,REAL_PART);

Mat A1 = getA1(imageRows,imageCols);
Mat A10 = new Mat(A1.size(),A1.type());
Core.multiply(A1, new Scalar(web[0]), A10);
Core.add(A0, A10, A10);

List<Mat> imageChannels = new ArrayList<>();

Core.split(blurredImage, imageChannels);

List<Mat> B0Channels = new ArrayList<>();

Mat aux;
List<Mat> imOutTempList = new ArrayList<>();


// Gaussian step
for(int ch = 0; ch < imageNumChannels ; ch++) {   // for each image channel

   aux = new Mat();

   Core.mulSpectrums(conjFft2Kernel, this.dft2(imageChannels.get(ch)), aux, 0);

   B0Channels.add(ch, aux);

   aux = this.divideB0A10(B0Channels.get(ch), A10);

   imOutTempList.add(ch, this.splitComplex(this.idft2(aux), REAL_PART));


}

double sigmaS = web[1];
double sigmaR = web[2];

List<Mat> imOutBi = this.dtrepf(imOutTempList,sigmaS,sigmaR);
```

```
    // Actual deconvolution

    Mat B1;
    Mat B;
    List<Mat> outputList = new ArrayList<>();
    Mat A12 = new Mat(A1.size(),A1.type());
    Core.multiply(A1, new Scalar(web[3]), A12);
    Core.add(A0, A12, A12);


    for(int i = 0; i < imageNumChannels; i++) {
        B = new Mat();

        B1 = this.getB1(imOutBi.get(i));

        Core.multiply(this.dft2(B1), this.initComplexMat(B1.size(), web[3], web[3]), B);
        aux = this.createComplexPart(B0Channels.get(i), 0);
        Core.add(aux, B, B);
        aux = new Mat();

        Core.divide(B, this.joinRealComplex(A12,A12), aux);

        outputList.add(i,this.splitComplex(this.idft2(aux),REAL_PART));
    }

    Mat output = new Mat(outputList.get(0).size(),IMAGE_FORMAT);
    Core.merge(outputList,output);

    return output;

}

private Mat divideB0A10(Mat B0, Mat A10) {

    Mat output = new Mat();

    if(B0.channels() == A10.channels())
        Core.divide(B0,A10,output);
    else if(B0.channels() == 2 && A10.channels() == 1)
    {
        List<Mat> listAux = new ArrayList<>();
        Mat aux = new Mat();
        Core.split(A10,listAux);
        A10.copyTo(aux);
        listAux.add(1, aux);
        aux = new Mat();
        Core.merge(listAux,aux);
        Core.divide(B0,aux,output);
    }

    return output;
```

```
}


private Mat initComplexMat(Size size, double realValue, double imaginaryValue) {

    Mat realPart = new Mat(size,KERNEL_FORMAT,new Scalar(realValue));
    Mat imaginaryPart = new Mat(size,KERNEL_FORMAT,new Scalar(imaginaryValue));

    List<Mat> outputList = new ArrayList<>();
    Mat output = new Mat();

    outputList.add(0, realPart);
    outputList.add(1, imaginaryPart);

    Core.merge(outputList, output);

    return output;

}

private Mat joinRealComplex(Mat realPart, Mat complexPart) {

    List<Mat> outputList = new ArrayList<>();

    outputList.add(0,realPart);
    outputList.add(1, complexPart);

    Mat output = new Mat();

    Core.merge(outputList, output);

    return output;

}


private Mat createComplexPart(Mat realPart, double initialValue) {

    if(realPart.channels() != 1)
        return realPart;

    Mat complexPart = new Mat(realPart.size(),KERNEL_FORMAT, new Scalar(initialValue));
    List<Mat> outputList = new ArrayList<>();
    Mat output = new Mat();
    outputList.add(0, realPart);
    outputList.add(1, complexPart);

    Core.merge(outputList, output);
```

```
    return output;

}


/**
 * RF  Domain transform recursive edge-preserving filter.
 */
private List<Mat> dtrepf(List<Mat> image, double sigmaS, double sigmaR) {


    int imageRows = image.get(0).rows();
    int imageCols = image.get(0).cols();
    int imageChannelsNum = image.size(); // The size of the list

    // Compute the domain transform (Equation 11 of our paper).
    // Estimate horizontal and vertical partial derivatives using finite
    // differences.

    Mat dIdx = new Mat(imageRows,imageCols,KERNEL_FORMAT,new Scalar(0));
    Mat dIdy = new Mat(imageRows,imageCols,KERNEL_FORMAT,new Scalar(0));

    double[] data = new double[1];
    Mat dx = new Mat(3,3,KERNEL_FORMAT, new Scalar(0));
    data[0] = (double) -1/8;
    dx.put(2,0,data);
    dx.put(0,0,data);
    data[0] = (double) 1/8;
    dx.put(0,2,data);
    dx.put(2,2,data);
    data[0] = (double) 2/8;
    dx.put(1,2,data);
    data[0] = (double) -2/8;
    dx.put(1, 0, data);

    Mat dy = new Mat(3,3,KERNEL_FORMAT, new Scalar(0));
    data[0] = (double) -1/8;
    dy.put(0,0,data);
    dy.put(0,2,data);
    data[0] = (double) 1/8;
    dy.put(2,0,data);
    dy.put(2,2,data);
    data[0] = (double) 2/8;
    dy.put(2,1,data);
    data[0] = (double) -2/ 8;
    dy.put(0, 1, data);

    List<Mat> dIcdx = new ArrayList<>();
    List<Mat> dIcdy = new ArrayList<>();

    Mat auxX;
```

```
        Mat auxY;
        for(int i = 0; i < imageChannelsNum ; i++) {

            auxX = this.convolve2D(image.get(i),dx,CONV2_TYPE.SAME);

            dIcdx.add(i, auxX);

            auxY = this.convolve2D(image.get(i),dy,CONV2_TYPE.SAME);
            dIcdy.add(i, auxY);

            Core.add(dIdx, this.absolute(auxX), dIdx);

            Core.add(dIdy,this.absolute(auxY),dIdy);

        }

        Core.divide(dIdx, new Scalar(imageChannelsNum), dIdx);
        Core.divide(dIdy, new Scalar(imageChannelsNum), dIdy);
        Mat dHdx = new Mat();
        Mat dVdy = new Mat();

        auxX = new Mat();
        data[0] = sigmaS / sigmaR;
        Core.multiply(dIdx, new Scalar(data[0]), auxX);
        Core.add(auxX, new Scalar(1), dHdx);

        auxY = new Mat();
        Core.multiply(dIdy, new Scalar(sigmaS / sigmaR), auxY);
        Core.add(auxY, new Scalar(1), dVdy);

         /*** Perform the filtering ***/

        // Transpose all channels
        for (int i = 0; i < imageChannelsNum ; i++)
            image.get(i).t().copyTo(image.get(i));

        List<Mat> output = this.transformedDomainRecursiveFilterHorizontal(image, dVdy.t(), sigmaS)
;


         // Transpose all channels
        for (int i = 0; i < imageChannelsNum ; i++)
            output.get(i).t().copyTo(output.get(i));

         output = this.transformedDomainRecursiveFilterHorizontal(output,dHdx,sigmaS);

        return output;

    }
```

```
    private List<Mat> transformedDomainRecursiveFilterHorizontal(List<Mat> image, Mat D, double
sigmaS) {

        int imageRows = image.get(0).rows();
        int imageCols = image.get(0).cols();
        int imageChannels = image.size();

        double a = Math.exp(( -(Math.sqrt(2)) / sigmaS));

        Mat V = this.doublePowMat(a, D);
        Mat aux;

        // Left -> Right filter
        for(int j = 1; j < imageCols; j++) {
            for(int ch = 0; ch < imageChannels; ch++) {
                aux = new Mat();
                Core.subtract(image.get(ch).col(j-1),image.get(ch).col(j),aux);
                Core.multiply(V.col(j),aux,aux);
                Core.add(image.get(ch).col(j),aux,image.get(ch).col(j));
            }
        }
        // Right -> Left filter
        for(int j = imageCols-2; j >= 0; j--) {
            for(int ch = 0; ch < imageChannels; ch++) {
                aux = new Mat();
                Core.subtract(image.get(ch).col(j+1),image.get(ch).col(j),aux);
                Core.multiply(V.col(j+1), aux, aux);
                Core.add(image.get(ch).col(j),aux,image.get(ch).col(j));
            }
        }

        return image;

    }


    private Mat doublePowMat(double a, Mat mat) {

        if(mat.channels() != 1)
            return null;

        Mat output = new Mat(mat.size(),mat.type(),new Scalar(0));

        int rows = mat.rows();
        int cols = mat.cols();
        double[] dAux = new double[1];

        for(int i = 0; i < rows; i++)
            for(int j = 0; j < cols; j++) {
                mat.get(i,j,dAux);
                output.put(i, j, Math.pow(a, dAux[0]));
```

```
        }

    return output;

}


private Mat absolute(Mat input) {

    Mat output = new Mat(input.size(),KERNEL_FORMAT);
    Mat angles = new Mat();

    // Has only real part, need to add imaginary
    if(input.channels() == 1) {
        Mat imaginaryPart = new Mat(input.size(), KERNEL_FORMAT, new Scalar(0));
        Core.cartToPolar(input,imaginaryPart,output,angles);
    }
    else if(input.channels() == 2){
        List<Mat> matList = new ArrayList<>();
        Core.split(input,matList);
        Core.cartToPolar(matList.get(0),matList.get(1),output,angles);
    }
    else
    {
        return null;
    }

    return output;

}


/**
 * Split real and complex parts of Mat
 * If param part is smaller than or equal to 0 return real part
 * If param part is greater than 0 return imaginary part
 *
 * @param input
 * @param part
 * @return
 */
private Mat splitComplex(Mat input, int part) {

    List<Mat> output = new ArrayList<>();

    Core.split(input, output);

    if(part > 0)
        return output.get(1);
    else
        return output.get(0);
```

```
}

private Mat complexConjugate(Mat input) {

    List<Mat> outputList = new ArrayList<>();
    Mat aux = new Mat();
    Mat output = new Mat();

    Core.split(input, outputList);

    Core.multiply(outputList.get(1), new Scalar(-1), aux);

    aux.copyTo(outputList.get(1));

    Core.merge(outputList, output);

    return output;

}

/**
 * Discrete Fourier Transform
 * @param input
 * @return
 */
private Mat dft2(Mat input) {

    if(input.channels() > 2)
        return null;

    Mat output = new Mat();

    Core.dft(input, output, Core.DFT_COMPLEX_OUTPUT, 0);

    return output;

}

/**
 * Inverse Discrete Fourier Transform
 * @param input
 * @return output
 */

private Mat idft2(Mat input) {

    if(input.channels() > 2)
        return null;

    Mat output = new Mat();
```

```
    // Has only real values
    if(input.channels() == 1) {
        List<Mat> join = new ArrayList<>();
        Mat zeros = Mat.zeros(input.size(),KERNEL_FORMAT);
        join.add(0,input);
        join.add(1,zeros);

        Core.merge(join,output);

        Core.idft(output, output, Core.DFT_COMPLEX_OUTPUT + Core.DFT_SCALE, 0);

    }
    else {

        Core.idft(input, output, Core.DFT_COMPLEX_OUTPUT + Core.DFT_SCALE, 0);
    }

    return output;

}



public Mat getA1(int R, int C) {

    Mat A1 = new Mat(R,C, KERNEL_FORMAT);

    Mat a1 = new Mat(R,C, KERNEL_FORMAT, new Scalar(0));

    // Use dx
    a1.put(0,0,1);
    a1.put(0,2,-0.25);      a1.put(0,C-2,-0.25);
    a1.put(R-2,0,-0.25); a1.put(2, 0, -0.25);

    double[] data = new double[1];

    // Use dxx
    a1.get(0, 0, data);
    data[0] = data[0] + 6;
    a1.put(0, 0, data);

    a1.get(0, 1, data);
    data[0] = data[0] - 2;
    a1.put(0, 1, data);

    a1.get(0, C - 1, data);
    data[0] = data[0] - 2;
    a1.put(0, C - 1, data);
```

```
a1.get(R - 1, 0, data);
data[0] = data[0] - 2;
a1.put(R - 1, 0, data);

a1.get(1, 0, data);
data[0] = data[0] - 2;
a1.put(1, 0, data);

a1.get(0, 2, data);
data[0] = data[0] + 0.5;
a1.put(0, 2, data);

a1.get(0, C - 2, data);
data[0] = data[0] + 0.5;
a1.put(0, C - 2, data);

a1.get(R - 2, 0, data);
data[0] = data[0] + 0.5;
a1.put(R - 2, 0, data);

a1.get(2, 0, data);
data[0] = data[0] + 0.5;
a1.put(2, 0, data);

// Use dxy
a1.get(0, 0, data);
data[0] = data[0] + 8;
a1.put(0, 0, data);

a1.get(0, 1, data);
data[0] = data[0] - 4;
a1.put(0, 1, data);

a1.get(0, C - 1, data);
data[0] = data[0] - 4;
a1.put(0, C - 1, data);

a1.get(R - 1, 0, data);
data[0] = data[0] - 4;
a1.put(R - 1, 0, data);

a1.get(1, 0, data);
data[0] = data[0] - 4;
a1.put(1, 0, data);

a1.get(1, 1, data);
data[0] = data[0] + 2;
a1.put(1, 1, data);

a1.get(R - 1, C - 1, data);
data[0] = data[0] + 2;
```

```
    a1.put(R - 1, C - 1, data);

    a1.get(R - 1, 1, data);
    data[0] = data[0] + 2;
    a1.put(R - 1, 1, data);

    a1.get(1, C - 1, data);
    data[0] = data[0] + 2;
    a1.put(1, C - 1, data);

    // Get the complex pairs
    Core.dft(a1, A1, Core.DFT_COMPLEX_OUTPUT, 0);
    // Get real part of A1
    A1 = this.splitComplex(A1,REAL_PART);

    return A1;

}

public Mat getB1(Mat image) {

    if(image.type() != KERNEL_FORMAT)
        return null;

    Mat conjDx = new Mat(1,3,image.type());
    conjDx.put(0,0,-0.5);
    conjDx.put(0, 1, 0);
    conjDx.put(0, 2, 0.5);

    Mat conjDy = new Mat(3,1,image.type());
    conjDy.put(0,0,-0.5);
    conjDy.put(1,0, 0);
    conjDy.put(2, 0, 0.5);

    Mat conjDxx = new Mat(1,3,image.type());
    conjDxx.put(0,0, -1/1.4142);
    conjDxx.put(0, 1, 2/1.4142);
    conjDxx.put(0, 2, -1 / 1.4142);

    Mat conjDyy = new Mat(3,1,image.type());
    conjDyy.put(0,0, -1/1.4142);
    conjDyy.put(1,0, 2/1.4142);
    conjDyy.put(2, 0, -1 / 1.4142);

    Mat conjDxy = new Mat(3,3,image.type());
    conjDxy.put(0,0,-1.4142);
    conjDxy.put(0,1,1.4142);
    conjDxy.put(0, 2, 0);
    conjDxy.put(1,0, 1.4142);
    conjDxy.put(1,1,-1.4142);
    conjDxy.put(1, 2, 0);
```

```
        conjDxy.put(2,0, 0);
        conjDxy.put(2,1, 0);
        conjDxy.put(2, 2, 0);

        // Rotate 180 degres
        Mat dx = this.rot180(conjDx);
        Mat dy = this.rot180(conjDy);
        Mat dxx = this.rot180(conjDxx);
        Mat dyy = this.rot180(conjDyy);
        Mat dxy = this.rot180(conjDxy);

        Mat w;
        Mat B1output;

        // Use dx
        double lambda = 0.065;


        w = this.convolve2D(image, dx, CONV2_TYPE.SAME);
        w = this.sparse(w, lambda);
        B1output = this.convolve2D(w, conjDx, CONV2_TYPE.SAME);

        w = this.convolve2D(image, dy);
        w = this.sparse(w, lambda);
        Core.add(B1output, this.convolve2D(w, conjDy, CONV2_TYPE.SAME), B1output);

        // Use dxx
        lambda = 0.5 * lambda;

        w = this.convolve2D(image, dxx, CONV2_TYPE.SAME);
        w = this.sparse(w, lambda);
        Core.add(B1output, this.convolve2D(w, conjDxx, CONV2_TYPE.SAME), B1output);

        //System.out.println("B1output 3" + B1output.dump());

        w = this.convolve2D(image, dyy, CONV2_TYPE.SAME);
        w = this.sparse(w, lambda);
        Core.add(B1output, this.convolve2D(w, conjDyy, CONV2_TYPE.SAME), B1output);

        // Use dxy
        w = this.convolve2D(image, dxy, CONV2_TYPE.SAME);
        w = this.sparse(w, lambda);
        Core.add(B1output, this.convolve2D(w, conjDxy, CONV2_TYPE.SAME), B1output);

        return B1output;

    }

    /**
     * Rotate image in 180 degrees
     * @param image
```

```
     * @return rotated 180 degrees image
     */

    private Mat rot180(Mat image) {

        //Core.transpose(image,image);
        //Core.flip(image, image, 0);
        //Core.transpose(image, image);
        //Core.flip(image, image, 0);

        Mat output = new Mat();
        Core.flip(image,output,-1);

        return output;

    }


    private Mat convolve2D(Mat image, Mat kernel) {

        Mat output = new Mat();

        Imgproc.filter2D(image, output, -1, kernel, new Point(-1, -1), 0, Imgproc.BORDER_CONSTAN
T);
        Core.multiply(output, new Scalar(-1), output);

        return output;

    }

    private Mat convolve2D(Mat image, Mat kernel, CONV2_TYPE conv2Type) {

        Mat source = image;
        Mat output = new Mat();
        Mat flippedKernel = new Mat();

        if(conv2Type == CONV2_TYPE.FULL) {
            source = new Mat();
            int additionalRows = kernel.rows()-1;
            int additionalCols = kernel.cols()-1;
            Imgproc.copyMakeBorder(image,source,(additionalRows+1)/2,additionalRows/2,
                    (additionalCols+1)/2,additionalCols/2,Imgproc.BORDER_CONSTANT, new Scalar(0));
        }

        Point anchor = new Point(kernel.cols()-kernel.cols()/2-1,kernel.rows()-kernel.rows()/2-1);
        Core.flip(kernel, flippedKernel, -1);
        Imgproc.filter2D(source,output,image.depth(),flippedKernel,anchor,0,Imgproc.BORDER_CONS
TANT);

        if(conv2Type == CONV2_TYPE.VALID)
            output.colRange((kernel.cols()-1)/2,output.cols()-kernel.cols()/2).rowRange(
```

```
                (kernel.rows()-1)/2,output.rows()-kernel.rows()/2);

        return output;

    }



    private Mat sparse(Mat x, double lambda) {

        Mat output = new Mat();
        Core.divide(lambda,x,output);
        Core.pow(output, 4,output);
        Core.add(output, new Scalar(1), output);

        Core.divide(x, output, output);

        return output;


    }


    private void printMat(Mat mat, String matName) {

        System.out.print(matName);
        Mat aux = new Mat(mat.size(),KERNEL_FORMAT);
        mat.copyTo(aux);
        System.out.println(aux.dump());

    }


    private void printListMat(List<Mat> list, String matName) {

        for(int i = 0; i < list.size(); i++) {
            this.printMat(list.get(i),matName + " " + i + " ");
        }

    }
}
```