Universidade Paulista

Ciência da Computação

PROCESSAMENTO DE IMAGEM

Aleksander Rocha - R.A.: C630IH-0

Daniel Jançanti - R.A.: C630IG-2

Ingrid Oliveira - R.A.: C51791-7

Rafaela Aranas - R.A.: C29CII-0

Araraquara – 2017

1 Introdução

Este trabalho tem por objetivo a implementação das operações de rotacionamento, redirecionamento e aplicação de filtros de imagens. A filtragem aplicada a uma imagem digital é uma operação local que modifica os valores dos níveis digitais de cada pixel da imagem considerando o contexto atual do pixel.

O processo de filtragem é feito utilizando matrizes denominadas máscaras, as quais são aplicadas sobre a imagem. Pela filtragem, o valor de cada pixel da imagem é modificado utilizando-se uma operação de vizinhança, ou seja, uma operação que leva em conta os níveis digitais dos pixels vizinhos e o próprio valor digital do pixel considerado. Nas definições de vizinhança começamos por considerar que: um pixel qualquer da imagem digital é vizinho dele mesmo.

2 Objetivos da filtragem

- Extração de ruídos da imagem;
- Homogeneização da imagem ou de alvos específicos;
- Melhorar na discriminação de alvos da imagem;
- Detecção de bordas entre alvos distintos presentes na imagem;
- Detecção de formas.

3 Descrição

Processamento de imagem é qualquer forma de processamento de dados no qual a entrada e saída são imagens tais como fotografias ou quadros de vídeo. Ao contrário do tratamento de imagens, que se preocupa somente na manipulação de figuras para sua representação final, o processamento de imagens é um estágio para novos processamentos de dados tais como aprendizagem de máquina ou reconhecimento de padrões. A maioria das técnicas envolve o tratamento da imagem como um sinal bidimensional, no qual são aplicados padrões de processamento de sinal.

3.1 Métodos de processamento

Algumas décadas atrás o processamento de imagem era feito majoritariamente de forma analógica, através de dispositivos ópticos. Apesar disso, devido ao grande aumento de velocidades dos computadores, tais técnicas foram gradualmente substituídas por métodos digitais.

O processamento digital de imagem é geralmente mais versátil, confiável e preciso, além de ser mais fácil de implementar que seus duais analógicos. Hardware especializado ainda é usado para o processamento digital de imagem, contando com arquiteturas de computador paralelas para tal, em sua maioria no processamento de vídeos. O processamento de imagens é, em sua maioria, feito por computadores pessoais.

3.2 Técnicas mais usadas

A maioria dos conceitos de processamento de sinais que se aplicam a sinais unidimensionais também podem ser estendidos para o processamento bidimensional de imagens. A transformada de Fourier é bastante usada nas operações de imagem envolvendo uma grande área de correlação.

- Resolução de imagem;
- Limite dinâmico;
- Largura de banda;
- Filtro: Permite a redução de ruídos da imagem para que mais padrões possam ser encontrados;

- Operador diferencial;
- Histograma: Consiste na frequência de um tom específico (seja escala de cinza ou colorido) em uma imagem. Permite a obtenção de informações como o brilho e o contraste da imagem e sua distribuição;
- Detecção de borda;
- Redução de ruído.

3.3 Problemas típicos

Além de imagens bidimensionais estáticas, o campo também abrange o processamento de sinais variados pelo tempo tais como vídeos ou a saída de um equipamento de tomografia. Tais técnicas são especificadas somente para imagens binárias ou em escala de cinza.

- Transformações geométricas tais como escala, rotação e inclinação;
- Correção de cor como ajustes de brilho e contraste, limiarização ou conversão de espaço de cor;
- Combinação de imagens por média, diferença ou composição;
- Interpolação e recuperação de imagem de um formato bruto tal como o filtro bayesiano;
- Segmentação de uma imagem em regiões;
- Edição de imagem e acabamento (retoque) digital;

4 Resultados

4.1 Histograma

O cálculo do histograma é um procedimento simples, onde um vetor que contém a intensidade (luminância) de cada pixel é indexado pelo valor do mesmo e incrementado a cada pixel.

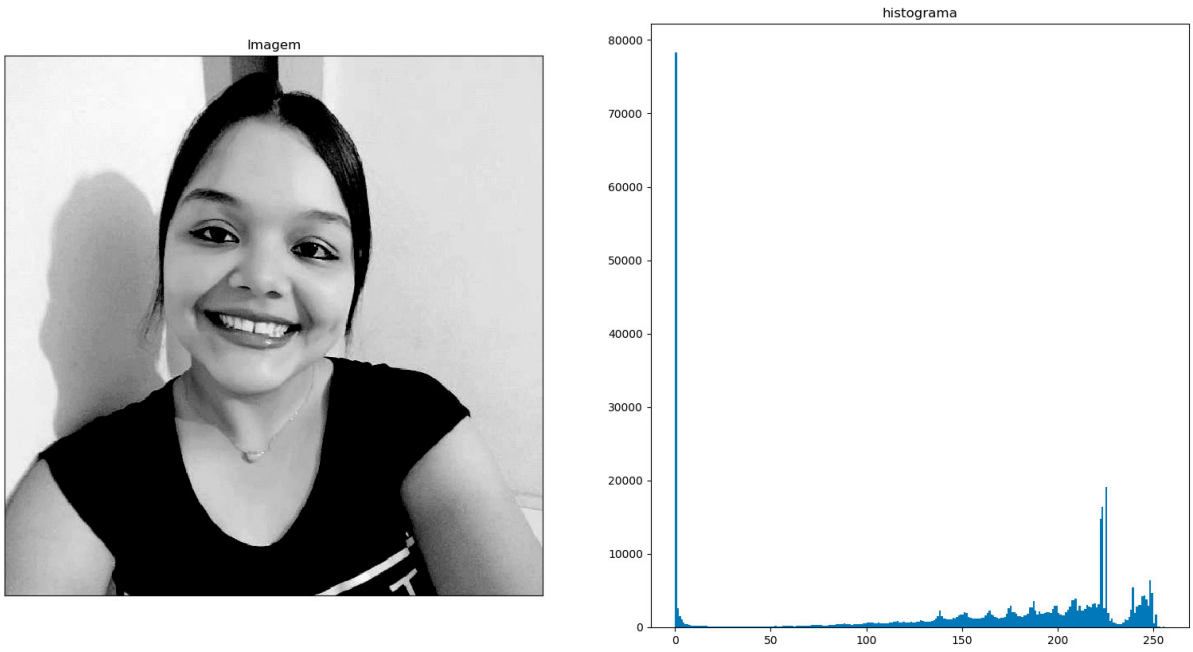

Figura 1: Histograma da imagem

```
void calcula_histograma(IplImage * img) {
  for (int i = 0; i < 256; i++) {
    histograma[i] = 0;
  }
  histMax = 0;
  foreachpixel(img, calc_hist, nothing, nothing);
}

void * calc_hist(unsigned char * w) {
  temp[ * w]++;
  if (temp[ * w] > histMax) {
    histMax = temp[ * w];
  }
}

IplImage * desenha_histograma(int HISTSIZE) {
  int HISTOFFSET;
  HISTOFFSET = HISTSIZE / 256;
  IplImage * ret;
  float a, b, c;
  b = histMax;
  int DrawHist[256];

  for (int i = 0; i < 256; i++) {
    a = (float) histograma[i];
    c = a / b;
    DrawHist[i] = (int)(c * HISTSIZE);
  }
  ret = cvCreateImage(cvSize(HISTSIZE, HISTSIZE), IPL_DEPTH_8U, 3);
```

```
  for (int i = 0; i < 256; i++) {
    cvRectangle(ret, cvPoint(i * HISTOFFSET + 1, HISTSIZE - DrawHist[i]), cvPoint(i * HISTOFFSET + HISTOFFSET, HISTSIZE), cvScalar(255, 127, 127),
  }
  return ret;
}
```

4.2 Equalização do Histograma

O cálculo do histograma é um procedimento simples, onde um vetor que contém a intensidade (luminância) de cada pixel é indexado pelo valor do mesmo e incrementado a cada pixel.

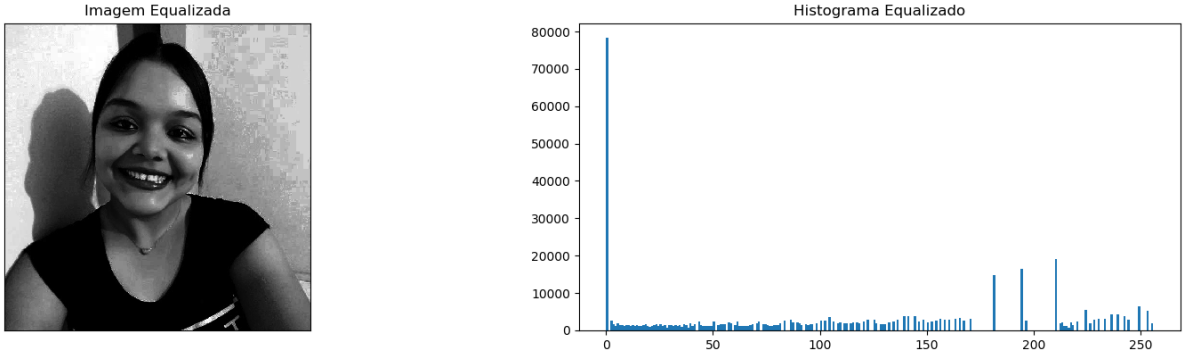

Figura 2: Histograma equalizado

```
void foreachpixel(IplImage * input, void * ( * red)(unsigned char * ), void * ( * green)(unsigned char * ), void * ( * blue)(unsigned char * )) {
  unsigned char * pixel;
  pixel = (unsigned char * ) input -> imageData;

  for (int i = 0; i < input -> width * input -> height * input -> nChannels; i += 3) {

    ( * red)(pixel);
    pixel += 3;
  }
  pixel = (unsigned char * ) input -> imageData + 1;
  for (int i = 0; i < input -> width * input -> height * input -> nChannels; i += 3) {

    ( * green)(pixel);
    pixel += 3;
  }

    pixel = (unsigned char * ) input -> imageData + 2;
  for (int i = 0; i < input -> width * input -> height * input -> nChannels; i += 3) {

    ( * blue)(pixel);
    pixel += 3;
  }
}
```

4.3 Conversão para escala cinza

Algoritmo de escala de cinza baseado na luminosidade do pixel pela visão humana usando a fórmula: `L = R*0.3 + B*0.59 + G*0.11`. Dado o resultado o algoritmo salva o pixel na forma LLL. Primeiro convertemos a imagem em JPEG para PPM

(formato simples e sem compressão, sendo mais fácil a manipulação), então obtemos um buffer dos pixels, na classe Image.

Figura 3: Imagem convertida para escala cinza

```
for i in range(img.width):
    for j in range(img.height):
        pix = image.Pix(img.getPixel(i, j))
        lum = int(pix[0]*0.3 + pix[1]*0.59 + pix[2]*0.11)

        img.setPixel(i, j, image.Pix((lum, lum, lum)))
```

4.4 Rotacionamento de imagens

O método de rotação consiste em usar a matriz de pixel e deslocá-las de lugar como na imagem abaixo praticamente especificado.

Figura 4: Imagem rotacionada em sentido anti-horário

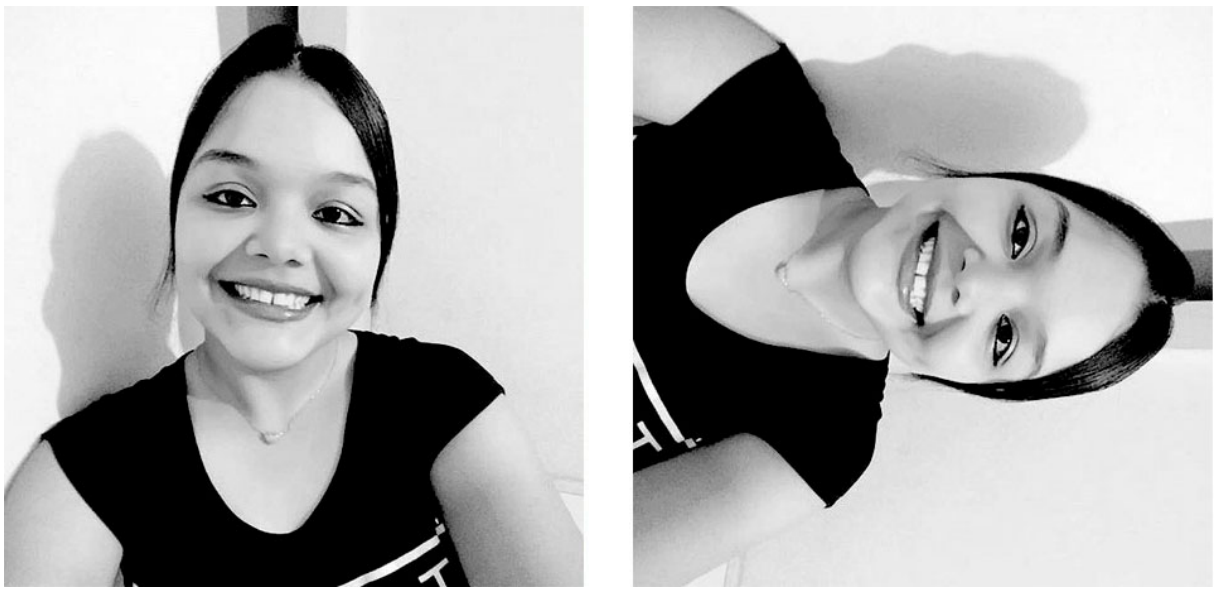

Figura 5: Imagem rotacionada em sentido horário

```
for x in xrange(xoffset,nw):
        for y in xrange(yoffset,nh):
            ox, oy = affine_t(x-cx, y-cy, *mrotate(-r, cx, cy))
            if ox > -1 and ox < ow and oy > -1 and oy < oh:
                pt = bilinear(bmp, ox, oy) if interpol else nn(bmp, ox, oy)
                draw.point([(x+mx,y+my),], fill=pt)
```

4.5 Redimensionamento

Ao dimensionar uma imagem gráfica, as primitivas gráficas que compõem a imagem podem ser dimensionadas usando transformações geométricas, sem perda de qualidade de imagem. Ao dimensionar uma imagem gráfica pixelada, uma nova imagem com um número maior ou menor de pixels deve ser gerada. No caso de diminuir o número de pixels (redução de escala), isso geralmente resulta em uma perda de qualidade visível. Do ponto de vista do processamento de sinal digital , o dimensionamento de gráficos de varredura é um exemplo bidimensional de conversão de taxa de amostragem, a conversão de um sinal discreto de uma taxa de amostragem (neste caso, a taxa de amostragem local) para outra.

Há vários algoritmos para se fazer esse processo, alguns deles são:

- Interpolação de vizinhança mais próxima
- Algoritmos bilineais e bicúbicos
- Reaprovação Sinc e Lanczos
- Amostragem de caixa

- Mipmap
- Métodos de transformada de Fourier
- Interpolação dirigida por borda
- HQX
- Vetorização

Figura 6: Imagem redimensionada

```
import numpy as np
from scipy.interpolate import griddata
import matplotlib.pyplot as plt

def func( x, y ):
    return np.sin(x*12.0)*np.sin(y*20.0)

points = np.random.rand(1000, 2)
values = func(points[:,0], points[:,1])

grid_x, grid_y = np.mgrid[0:1:100j, 0:1:200j]
grid_z = griddata(points, values, (grid_x, grid_y), method='cubic')

plt.imshow(grid_z.T, extent=(0,1,0,1), origin='lower')
plt.scatter(points[:,0], points[:,1], c='k')

plt.show()
```

4.6 Espelhamento

O espelhamento da imagem é a inversão dos pixel na matriz no sentido oposto, por exemplo, um espelhamento horizontal os pixel da direita são movidos para a esqueda e, no espelhamento vertical os pixels inferiores são movidos para cima.

Figura 7: Imagem espelhada horizontalmente

```
for (line = 0;line < height+gamb/*+1*/ ; line++)
        {//swap height -line with line
        linePt[line] = data;
        data += channels*width;
        buffer =          linePt[line];

        }

for (line = 0;line < height/2/*-1*/; line++) //paralelizar ?
        {

        memcpy(buffer,linePt[line],sizeof(unsigned char)*width*channels);

        memcpy(linePt[line],linePt[height -line -1],sizeof(unsigned char)*width*channels);

        memcpy(linePt[height - line-1 ],buffer,sizeof(unsigned char)*width*channels);
        }
```

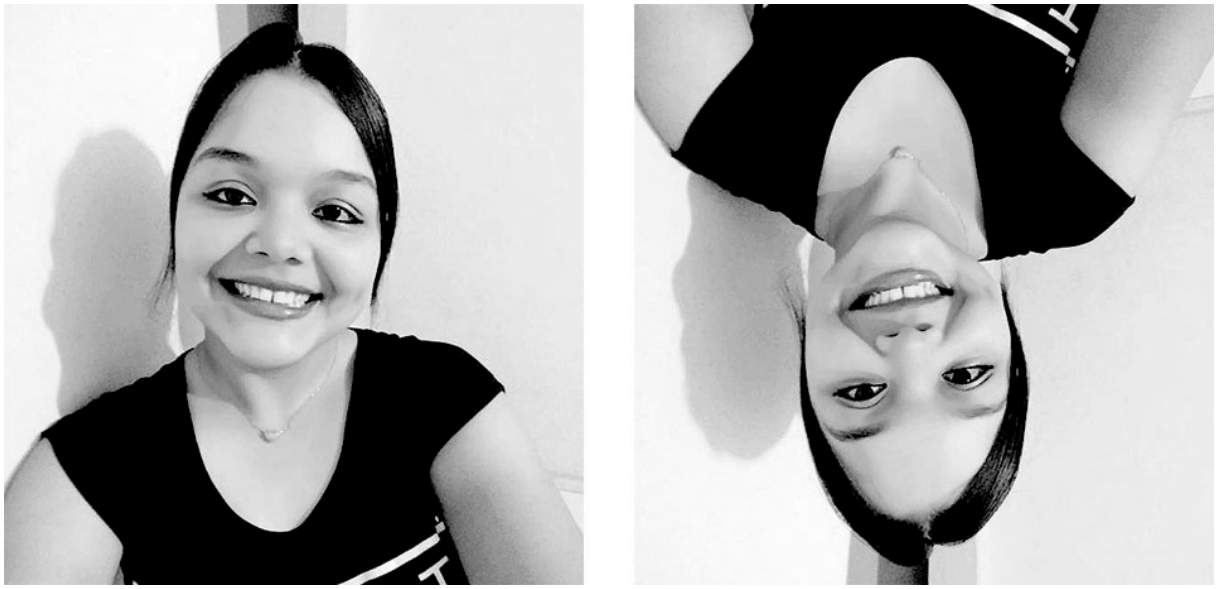

Figura 8: Imagem espelhada verticalmente

```
void SwapChar(unsigned char *A,unsigned char *B,int number)
```

```
{
unsigned char *C;
C = (unsigned char *) calloc(number, sizeof(unsigned char));
memcpy(C,A,sizeof(unsigned char)*number);
memcpy(A,B,sizeof(unsigned char)*number);
memcpy(B,C,sizeof(unsigned char)*number);
free(C);
}
```

```
for(column = 0;column < width/2; column++)
 {
 destPointer = data + lineSize*line + (column)*channels;
 sourcePointer = data + lineSize*line + (width-column)*channels;
 SwapChar(destPointer,sourcePointer,channels);
 }
```

4.7 Negativo

Para se gerar uma imagem negativa é necessário efetuar os seguintes passos:

1. Extrair o fator de luminância da imagem para a cor cinza
2. criar uma nova cor utilizando para os valores de RGB o fator de escala de cinza obtido

No caso, esta luminância em cinza é baseada em 30% de vermelho, 59% de verde e 11% de azul. Vale lembrar que como estamos trabalhando no padrão RGB o valor mínimo de uma cor é zero e o máximo e 255. Para realizar esta mudança em cada um dos pixels da imagem devemos percorrer todos os pixels da mesma, extrair a cor, gerar o fator de luminância para o cinza e alterar a cor do pixel.

Figura 9: Imagem negativa

```
public class pdi {
 public static void main(String args[] ) throws IOException{
    BufferedImage imagem = null;
    imagem = ImageIO.read(
     new File("C:\\Users\\ufc\\workspace\\PDI\\src\\tux.png")
    );

    int w = imagem.getWidth();
    int h = imagem.getHeight();
    //retorna um vetor inteiro representando os pixels da imagem
    int pixels[] = imagem.getRGB(0, 0, w, h, null, 0, w);

    for (int col = 0; col < w; col++) {
      for (int lin = 0; lin < h; lin++) {
  //Pega a cor em cada pixel
  Color c = new Color(pixels[lin*w + col]);
  pixels[lin*w + col] = new
  Color(255-c.getRed(),255-c.getBlue(),255-c.getGreen()).getRGB();
      }
 }
        //seta um vetor inteiro representando os pixels da imagem
 imagem.setRGB(0, 0, w, h, pixels, 0, w);
 ImageIO.write(imagem, "PNG",
        new File("C:\\Users\\ufc\\workspace\\PDI\\src\\tux2.png"));
 }
}
```

4.8 Brilho

Semelhantemente ao brilho, o contraste foi implementado usando-se um spin box para uma melhor precisão. O código implementado foi o seguinte:

Figura 10: Imagem com brilho

```
void MainWindow::on_brilho_BW_sliderMoved(int position) {

  brilho = position - brilho_anterior[0];
  brilho_anterior[0] = position;
  foreachpixel(output, brilho_pixel, brilho_pixel, brilho_pixel);
  cvShowImage("OUTPUT", output);
}
```

```
void MainWindow::on_brilho_R_sliderMoved(int position) {
  brilho = position - brilho_anterior[1];
  brilho_anterior[1] = position;
  foreachpixel(output, nothing, nothing, brilho_pixel);
  cvShowImage("OUTPUT", output);
}

  void * brilho_pixel(unsigned char * w) {
    int temp;
    temp = * w + brilho;
    if (temp > 255) {
      temp = 255;
    }
    if (temp < 0) {
      temp = 0;
    } * w = (unsigned char) temp;
  }
```

4.9 Contraste

O ajuste de brilho é implementado como uma operação pixel a pixel usando-se uma função. Como foi utilizada uma slide bar, o procedimento usado foi salvar o estado anterior da slide bar para incrementar ou decrementar a variação da barra. A rotina para tratamento de cada mudança de valor na slide bar é a seguinte:

Figura 11: Imagem com contraste

```
void MainWindow::on_Constrate_valueChanged(double value) {
  output = cvLoadImage(fn.toStdString().c_str());
  contraste = (float) value;
  switch (contraste_select) {
  case CONTRASTE_ALL:
    foreachpixel(output, contraste_pixel, contraste_pixel, contraste_pixel);

    break;
  case CONTRASTE_RED:
    foreachpixel(output, nothing, nothing, contraste_pixel);
    break;
  case CONTRASTE_BLUE:
    foreachpixel(output, nothing, contraste_pixel, nothing);
```

```
    break;
  case CONTRASTE_GREEN:
    foreachpixel(output, contraste_pixel, nothing, nothing);
    break;
  }
  cvShowImage("OUTPUT", output);
}

  void * contraste_pixel(unsigned char * w) {
    float temp;
    temp = ( * w) * contraste;
    if (temp > 255) {
      temp = 255;
    }
    if (temp < 0) {
      temp = 0;
    } * w = (unsigned char) temp;
  }
```

4.10 Filtro Sobel

Consiste num operador que calcula diferenças finitas, dando uma aproximação do gradiente da intensidade dos pixels da imagem. Em cada ponto da imagem, o resultado da aplicação do filtro Sobel devolve o gradiente ou a norma deste vector, aplicada sobretudo em algoritmos de detecção de contornos.

O filtro Sobel calcula o gradiente da intensidade da imagem em cada ponto, dando a direcção da maior variação de claro para escuro e a quantidade de variação nessa direcção. Assim, obtém-se uma noção de como varia a luminosidade em cada ponto, de forma mais suave ou abrupta.

Com isto consegue-se estimar a presença de uma transição claro-escuro e de qual a orientação desta. Como as variações claro-escuro intensas correspondem a fronteiras bem definidas entre objectos, consegue-se fazer a detecção de contornos.

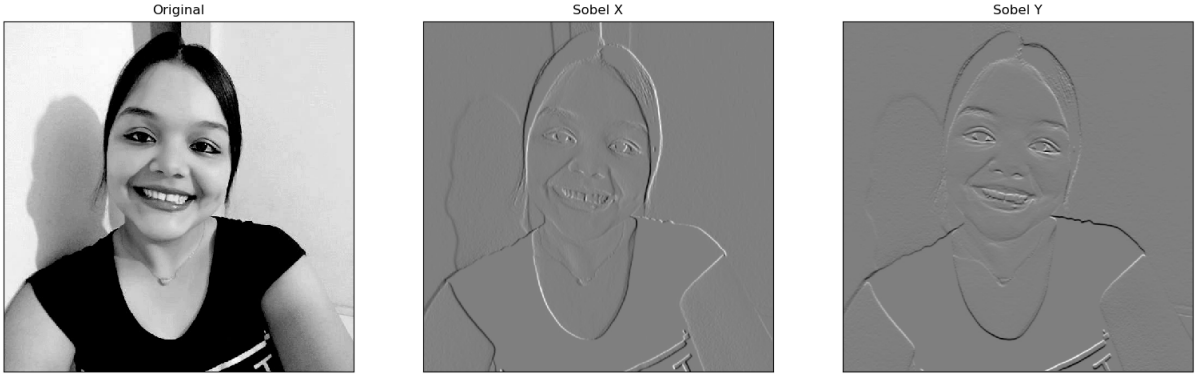

Figura 12: Filtro Sobel

```
from PIL import Image
import math
path = "ingrid.jpg"
img = Image.open(path)
```

```
newimg = Image.new("RGB", (width, height), "white")
for x in range(1, width-1):
    for y in range(1, height-1):
        Gx, Gy = 0,0
        p = img.getpixel((x-1, y-1))
        r,g,b = p[0],p[1],p[2]
        intensity = r + g + b
        Gx += -intensity
        Gy += -intensity
        p = img.getpixel((x-1, y))
        r,g,b = p[0],p[1],p[2]
        Gx += -2 * (r + g + b)
        p = img.getpixel((x-1, y+1))
        r,g,b = p[0],p[1],p[2]
        Gx += -(r + g + b)
        Gy += (r + g + b)
        p = img.getpixel((x, y-1))
        r,g,b = p[0],p[1],p[2]
        Gy += -2 * (r + g + b)
        p = img.getpixel((x, y+1))
        r,g,b = p[0],p[1],p[2]
        Gy += 2 * (r + g + b)
        p = img.getpixel((x+1, y-1))
        r,g,b = p[0],p[1],p[2]
        Gx += (r + g + b)
        Gy += -(r + g + b)
        p = img.getpixel((x+1, y))
        r,g,b = p[0],p[1],p[2]
        Gx += 2 * (r + g + b)
        p = img.getpixel((x+1, y+1))
        r,g,b = p[0],p[1],p[2]
        Gx += (r + g + b)
        Gy += (r + g + b)
        length = math.sqrt((Gx * Gx) + (Gy * Gy))
        length = length / 4328 * 255
        length = int(length)
        newimg.putpixel((x,y),(length,length,length))
```

4.11 Filtro Prewitt

Em termos simples, o operador calcula o gradiente da intensidade da imagem em cada ponto, dando a direção do maior aumento possível da luz para a escuridão e a taxa de mudança nessa direção. O resultado, portanto, mostra como "abruptamente"ou "suavemente"a imagem muda nesse ponto e, portanto, como é provável que essa parte da imagem represente uma vantagem , bem como a forma como essa borda provavelmente estará orientada. Na prática, o cálculo da magnitude (probabilidade de uma borda) é mais confiável e mais fácil de interpretar do que o cálculo da direção.

```
for (int i = 0; i < (height-prewitt.size+1)*(width-prewitt.size+1)*4; i++) {
    imgb[i] = 0;
}

infoheader.biHeight = height - prewitt.size + 1;
infoheader.biWidth = width - prewitt.size + 1;

picsize = (height-prewitt.size+1)*(width-prewitt.size+1);


cur_idx = bitmap.plot(blue_img, prewitt);
```

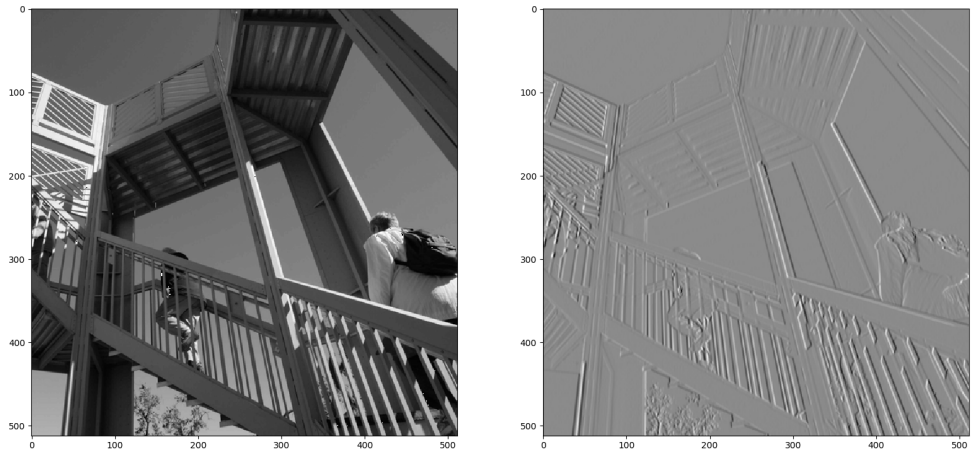

Figura 13: Filtro Prewitt

```
for (int i = 0; i < picsize; i++) {

    tpp = *cur_idx;
    imgb[i*4] = tpp;
    cur_idx++;

}
```

4.12 Filtro Roberts

O operador Roberts cross é usado no processamento de imagens e visão computacional para detecção de borda. Foi um dos primeiros detectores de borda e foi inicialmente proposto por Lawrence Roberts em 1963. Como operador diferencial, a idéia por trás do operador Roberts Cross é aproximar o gradiente de uma imagem através de diferenciação discreta *m* que é conseguida pela computação a soma dos quadrados das diferenças entre pixels diagonalmente adjacentes.

```
for (int i = 0; i < (height-roberts.size+1)*(width-roberts.size+1)*4; i++) {
    imga[i] = 0;
}

infoheader.biHeight = height - roberts.size + 1;
infoheader.biWidth = width - roberts.size + 1;

picsize = (height-roberts.size+1)*(width-roberts.size+1);


cur_idx = bitmap.plot(blue_img, roberts);

for (int i = 0; i < picsize; i++) {
    tpp = *cur_idx;
    imga[i*4] = (int) tpp;
    cur_idx++;
}
```

Figura 14: Filtro Roberts

4.13 Filtro Canny

Desenvolvido por John F. Canny em 1986 o detector de bordas de Canny utiliza um algoritmo multi-estágios para detectar uma ampla margem de bordas na imagem. Canny também desenvolveu uma teoria computacional sobre detecção de bordas explicando porque as técnicas funcionam. John Canny propôs que o detector de bordas ótimo deveria respeitar os seguintes parâmetros:

Boa detecção: O algoritmo deve ser capaz de identificar todas as bordas possíveis na imagem

Boa Localização: As bordas encontradas devem estar o mais próximo possível das bordas da imagem original.

Resposta Mínima: Cada borda da imagem deve ser marcada apenas uma vez. O ruído da imagem não deve criar falsas bordas.

Para satisfazer tais condições, Canny utilizou um cálculo de variações, visando encontrar uma função que otimizasse o funcional desejado. A função ideal para o detector de Canny é descrito pela soma de quatro termos de exponenciais, que pode ser aproximada pela primeira derivada de uma gaussiana.

```
void canny (Mat& A) {
    Mat B;
    if (sigma) {
        GaussianBlur (A, B, Size(0,0), sigma);
    } else
        A.copyTo(B);
    Gradient grad = get_gradient (B);
    Mat C;
```

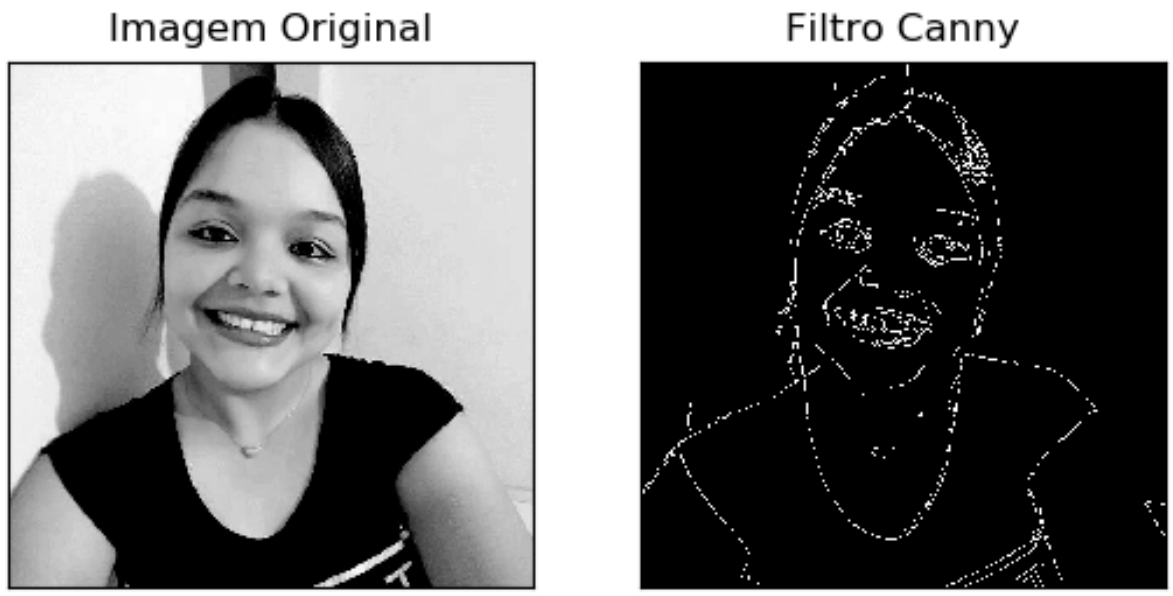

Figura 15: Filtro Canny

```
    thin_edge (grad.G, C, grad.theta);
    float max_grad = get_max_grad (C);
    high_seuil = max_grad * (max_slider/100.0);
    low_seuil = max_grad * (min_slider/100.0);
    Mat D (C.rows, C.cols, CV_8U);
    BFS_edge_tracking (C, D);
    imshow ("Canny␣Algorithm", D);
}
```

4.14 Filtro Laplaciano

É um filtro de detecção de bordas que gera uma borda fina de apenas um pixel de largura e é baseado numa derivada de 2ª ordem.

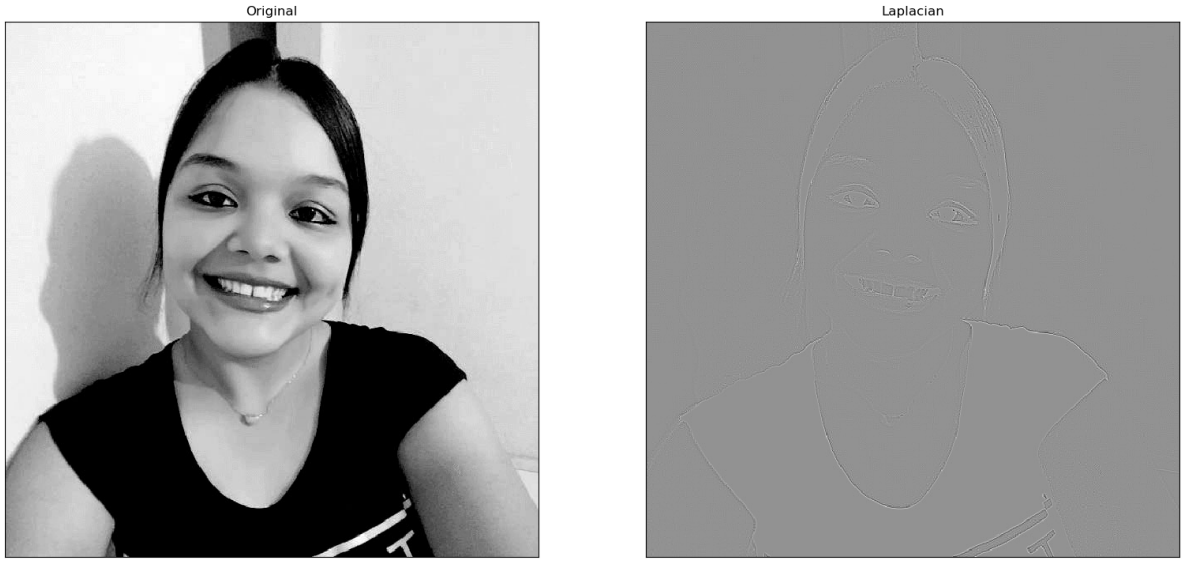

Figura 16: Filtro laplaciano

```
Mat LaplacianPyramid::Reconstruct() const {
  Mat base = pyramid_.back();
  Mat expanded;

  for (int i = pyramid_.size() - 2; i >= 0; i--) {
    vector<int> subwindow;
    GaussianPyramid::GetLevelSize(subwindow_, i, &subwindow);
    int row_offset = ((subwindow[0] % 2) == 0) ? 0 : 1;
    int col_offset = ((subwindow[2] % 2) == 0) ? 0 : 1;

    expanded.create(pyramid_[i].rows, pyramid_[i].cols, base.type());

    if (base.channels() == 1) {
      GaussianPyramid::Expand<double>(base, row_offset, col_offset, expanded);
    } else if (base.channels() == 3) {
      GaussianPyramid::Expand<Vec3d>(base, row_offset, col_offset, expanded);
    }
    base = expanded + pyramid_[i];
  }

  return base;
}
```

4.15 Filtro Média

Substitui o valor do pixel original pela média aritmética do pixel dos seus vizinhos. Quanto maior a máscara, maior o efeito de borramento. Pesos positivos. Soma dos pesos igual a 1 – não altera a média.

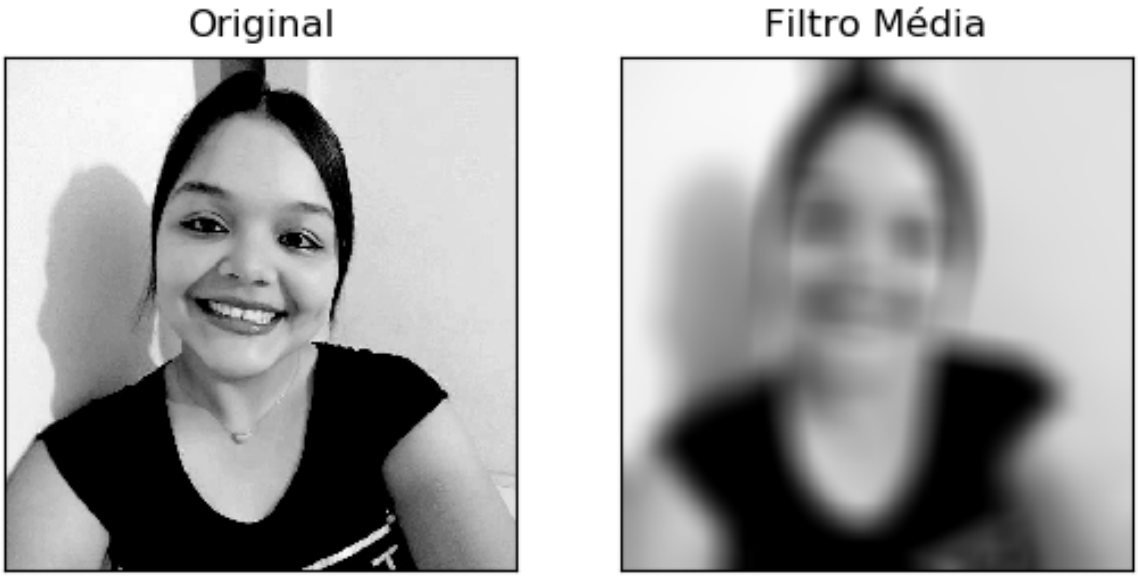

Figura 17: Filtro Média

```
def running_mean(l, N):
    sum = 0
    result = list( 0 for x in l)

    for i in range( 0, N ):
        sum = sum + l[i]
        result[i] = sum / (i+1)
```

```
    for i in range( N, len(l) ):
        sum = sum - l[i-N] + l[i]
        result[i] = sum / N

    return result
```

4.16 Filtro Mediana

Suaviza a imagem sem contudo diminuir sua resolução. os pontos da vizinhança de (x,y), dentro de uma janela na imagem, são ordenados e tomado como novo valor para (x,y) o valor mediano desta ordenação.

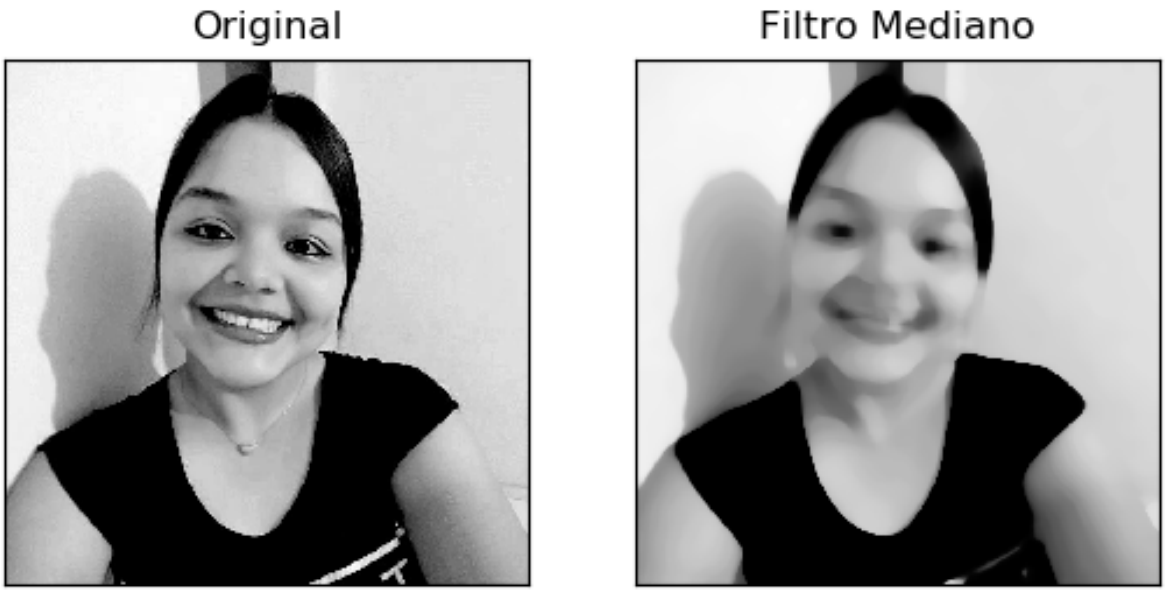

Figura 18: Filtro Mediana

```
def medianOfMedians(l, j):
    if len(l) < 10:
        l.sort()
        return l[j]
    s = []
    lIndex = 0
    while lIndex+5 < len(l)-1:
        s.append(l[lIndex:lIndex+5])
        lIndex += 5
    s.append(l[lIndex:])
    meds = []
    for subList in s:
        meds.append(medianOfMedians(subList, int((len(subList)-1)/2)))
    med = medianOfMedians(meds, int((len(meds)-1)/2))
    l1 = []
    l2 = []
    l3 = []
    for i in l:
        if i < med:
            l1.append(i)
        elif i > med:
            l3.append(i)
        else:
            l2.append(i)
    if j < len(l1):
```

```
        return medianOfMedians(l1, int(j))
    elif j < len(l2) + len(l1):
        return l2[0]
    else:
        return medianOfMedians(l3, int(j-len(l1)-len(l2)))
```

4.17 Moda

Este filtro é usado para homogeneizar imagens temáticas, ou para reduzir ruídos mantendo o máximo de informação na imagem.

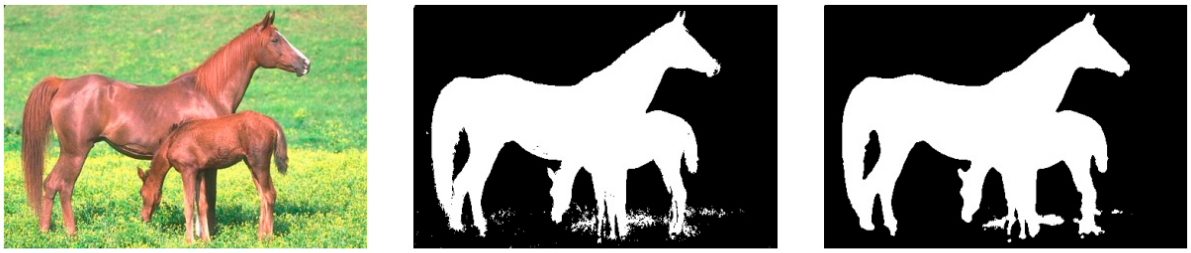

Figura 19: Filtro Moda

4.18 Filtro Gaussiano

Um filtro gaussiano é utilizado para borrar ou desfocar a imagem na qual ele é aplicado com o objetivo de reduzir os ruídos presentes na imagem. O resultado desta operação é a suavização da imagem, relembrando a visualização da mesma através de uma tela translúcida ou como se tivesse sendo vista através de uma lente fora de foco. A suavização gaussiana é largamente utilizada no estágio de pré-processamento da imagem a fim de enaltecer a estrutura da imagem em diferentes escalas.

```
private double getBlurColor(int x, int y,int whichColor){

    double blurGray = 0;
    double[][] colorMat = getColorMatrix(x,y,whichColor);

    int length = blurRadius*2+1;
    for (int i = 0;i<length;i++){
        for (int j=0; j<length; j++ ){
            blurGray += weightArr[i][j]*colorMat[i][j];
        }
    }

    return blurGray;
}
```

Figura 20: Filtro Gaussiano

5 Conclusão

Concluímos o estudo de processamento de imagem com quase todas as soluções apresentadas, dentre as possibilidades tiramos proveito do funcionamento dos filtros e dos algoritmos, podendo assim ter uma visão mais ampla do conceito de processamento de imagem.